Django Documentation

1.10

Django Software Foundation

November 27, 2016

Contents

1 Django documentation **1**
- 1.1 Getting help . . . 1
- 1.2 How the documentation is organized . . . 1
- 1.3 First steps . . . 1
- 1.4 The model layer . . . 2
- 1.5 The view layer . . . 2
- 1.6 The template layer . . . 2
- 1.7 Forms . . . 3
- 1.8 The development process . . . 3
- 1.9 The admin . . . 3
- 1.10 Security . . . 3
- 1.11 Internationalization and localization . . . 4
- 1.12 Performance and optimization . . . 4
- 1.13 Python compatibility . . . 4
- 1.14 Geographic framework . . . 4
- 1.15 Common Web application tools . . . 4
- 1.16 Other core functionalities . . . 5
- 1.17 The Django open-source project . . . 5

2 Getting started **7**
- 2.1 Django at a glance . . . 7
- 2.2 Quick install guide . . . 12
- 2.3 Writing your first Django app, part 1 . . . 13
- 2.4 Writing your first Django app, part 2 . . . 19
- 2.5 Writing your first Django app, part 3 . . . 31
- 2.6 Writing your first Django app, part 4 . . . 37
- 2.7 Writing your first Django app, part 5 . . . 42
- 2.8 Writing your first Django app, part 6 . . . 51
- 2.9 Writing your first Django app, part 7 . . . 53
- 2.10 Advanced tutorial: How to write reusable apps . . . 62
- 2.11 What to read next . . . 67
- 2.12 Writing your first patch for Django . . . 70

3 Using Django **81**
- 3.1 How to install Django . . . 81
- 3.2 Models and databases . . . 84
- 3.3 Handling HTTP requests . . . 176
- 3.4 Working with forms . . . 216

3.5 Templates . . . 259
3.6 Class-based views . . . 270
3.7 Migrations . . . 294
3.8 Managing files . . . 306
3.9 Testing in Django . . . 308
3.10 User authentication in Django . . . 346
3.11 Django's cache framework . . . 394
3.12 Conditional View Processing . . . 411
3.13 Cryptographic signing . . . 414
3.14 Sending email . . . 416
3.15 Internationalization and localization . . . 426
3.16 Logging . . . 470
3.17 Pagination . . . 480
3.18 Porting to Python 3 . . . 484
3.19 Security in Django . . . 490
3.20 Performance and optimization . . . 493
3.21 Serializing Django objects . . . 499
3.22 Django settings . . . 507
3.23 Signals . . . 511
3.24 System check framework . . . 515
3.25 External packages . . . 518

4 "How-to" guides **521**
4.1 Authentication using `REMOTE_USER` . . . 521
4.2 Writing custom `django-admin` commands . . . 523
4.3 Writing custom model fields . . . 528
4.4 Custom Lookups . . . 538
4.5 Custom template tags and filters . . . 543
4.6 Writing a custom storage system . . . 559
4.7 Deploying Django . . . 561
4.8 Upgrading Django to a newer version . . . 573
4.9 Error reporting . . . 575
4.10 Providing initial data for models . . . 579
4.11 Running Django on Jython . . . 580
4.12 Integrating Django with a legacy database . . . 581
4.13 Outputting CSV with Django . . . 582
4.14 Outputting PDFs with Django . . . 584
4.15 Managing static files (e.g. images, JavaScript, CSS) . . . 587
4.16 Deploying static files . . . 589
4.17 How to install Django on Windows . . . 591
4.18 Writing database migrations . . . 593

5 Django FAQ **599**
5.1 FAQ: General . . . 599
5.2 FAQ: Installation . . . 602
5.3 FAQ: Using Django . . . 603
5.4 FAQ: Getting Help . . . 604
5.5 FAQ: Databases and models . . . 605
5.6 FAQ: The admin . . . 606
5.7 FAQ: Contributing code . . . 608
5.8 Troubleshooting . . . 609

6 API Reference **611**
6.1 Applications . . . 611

6.2 System check framework . . . 617
6.3 Built-in class-based views API . . . 627
6.4 Clickjacking Protection . . . 675
6.5 `contrib` packages . . . 677
6.6 Cross Site Request Forgery protection . . . 925
6.7 Databases . . . 933
6.8 `django-admin` and `manage.py` . . . 945
6.9 Running management commands from your code . . . 969
6.10 Django Exceptions . . . 970
6.11 File handling . . . 974
6.12 Forms . . . 981
6.13 Middleware . . . 1036
6.14 Migration Operations . . . 1042
6.15 Models . . . 1049
6.16 Request and response objects . . . 1168
6.17 `SchemaEditor` . . . 1182
6.18 Settings . . . 1185
6.19 Signals . . . 1229
6.20 Templates . . . 1236
6.21 `TemplateResponse` and `SimpleTemplateResponse` . . . 1299
6.22 Unicode data . . . 1303
6.23 `django.urls` utility functions . . . 1309
6.24 `django.conf.urls` utility functions . . . 1312
6.25 Django Utils . . . 1314
6.26 Validators . . . 1328
6.27 Built-in Views . . . 1332

7 Meta-documentation and miscellany 1335
7.1 API stability . . . 1335
7.2 Design philosophies . . . 1336
7.3 Third-party distributions of Django . . . 1340

8 Glossary 1343

9 Release notes 1345
9.1 Final releases . . . 1345
9.2 Security releases . . . 1581

10 Django internals 1597
10.1 Contributing to Django . . . 1597
10.2 Mailing lists . . . 1639
10.3 Organization of the Django Project . . . 1641
10.4 Django team . . . 1644
10.5 Roles . . . 1650
10.6 Django's security policies . . . 1651
10.7 Django's release process . . . 1654
10.8 Django Deprecation Timeline . . . 1657
10.9 The Django source code repository . . . 1670
10.10 How is Django Formed? . . . 1673

11 Indices, glossary and tables 1681

Python 1683

CHAPTER 1

Django documentation

Everything you need to know about Django.

1.1 Getting help

Having trouble? We'd like to help!

- Try the FAQ – it's got answers to many common questions.
- Looking for specific information? Try the genindex, modindex or the detailed table of contents.
- Search for information in the archives of the *django-users* mailing list, or post a question.
- Ask a question in the #django IRC channel, or search the IRC logs to see if it's been asked before.
- Report bugs with Django in our ticket tracker.

1.2 How the documentation is organized

Django has a lot of documentation. A high-level overview of how it's organized will help you know where to look for certain things:

- Tutorials take you by the hand through a series of steps to create a Web application. Start here if you're new to Django or Web application development. Also look at the "*First steps*" below.
- Topic guides discuss key topics and concepts at a fairly high level and provide useful background information and explanation.
- Reference guides contain technical reference for APIs and other aspects of Django's machinery. They describe how it works and how to use it but assume that you have a basic understanding of key concepts.
- How-to guides are recipes. They guide you through the steps involved in addressing key problems and use-cases. They are more advanced than tutorials and assume some knowledge of how Django works.

1.3 First steps

Are you new to Django or to programming? This is the place to start!

- **From scratch:** Overview | Installation

- **Tutorial:** Part 1: Requests and responses | Part 2: Models and the admin site | Part 3: Views and templates | Part 4: Forms and generic views | Part 5: Testing | Part 6: Static files | Part 7: Customizing the admin site
- **Advanced Tutorials:** How to write reusable apps | Writing your first patch for Django

1.4 The model layer

Django provides an abstraction layer (the "models") for structuring and manipulating the data of your Web application. Learn more about it below:

- **Models:** Introduction to models | Field types | Meta options | Model class
- **QuerySets:** Executing queries | QuerySet method reference | Lookup expressions
- **Model instances:** Instance methods | Accessing related objects
- **Migrations:** Introduction to Migrations | Operations reference | SchemaEditor | Writing migrations
- **Advanced:** Managers | Raw SQL | Transactions | Aggregation | Search | Custom fields | Multiple databases | Custom lookups | Query Expressions | Conditional Expressions | Database Functions
- **Other:** Supported databases | Legacy databases | Providing initial data | Optimize database access | PostgreSQL specific features

1.5 The view layer

Django has the concept of "views" to encapsulate the logic responsible for processing a user's request and for returning the response. Find all you need to know about views via the links below:

- **The basics:** URLconfs | View functions | Shortcuts | Decorators
- **Reference:** Built-in Views | Request/response objects | TemplateResponse objects
- **File uploads:** Overview | File objects | Storage API | Managing files | Custom storage
- **Class-based views:** Overview | Built-in display views | Built-in editing views | Using mixins | API reference | Flattened index
- **Advanced:** Generating CSV | Generating PDF
- **Middleware:** Overview | Built-in middleware classes

1.6 The template layer

The template layer provides a designer-friendly syntax for rendering the information to be presented to the user. Learn how this syntax can be used by designers and how it can be extended by programmers:

- **The basics:** Overview
- **For designers:** Language overview | Built-in tags and filters | Humanization
- **For programmers:** Template API | Custom tags and filters

1.7 Forms

Django provides a rich framework to facilitate the creation of forms and the manipulation of form data.

- **The basics:** Overview | Form API | Built-in fields | Built-in widgets
- **Advanced:** Forms for models | Integrating media | Formsets | Customizing validation

1.8 The development process

Learn about the various components and tools to help you in the development and testing of Django applications:

- **Settings:** Overview | Full list of settings
- **Applications:** Overview
- **Exceptions:** Overview
- **django-admin and manage.py:** Overview | Adding custom commands
- **Testing:** Introduction | Writing and running tests | Included testing tools | Advanced topics
- **Deployment:** Overview | WSGI servers | Deploying static files | Tracking code errors by email

1.9 The admin

Find all you need to know about the automated admin interface, one of Django's most popular features:

- Admin site
- Admin actions
- Admin documentation generator

1.10 Security

Security is a topic of paramount importance in the development of Web applications and Django provides multiple protection tools and mechanisms:

- Security overview
- Disclosed security issues in Django
- Clickjacking protection
- Cross Site Request Forgery protection
- Cryptographic signing
- *Security Middleware*

1.11 Internationalization and localization

Django offers a robust internationalization and localization framework to assist you in the development of applications for multiple languages and world regions:

- Overview | Internationalization | *Localization* | Localized Web UI formatting and form input
- Time zones

1.12 Performance and optimization

There are a variety of techniques and tools that can help get your code running more efficiently - faster, and using fewer system resources.

- Performance and optimization overview

1.13 Python compatibility

Django aims to be compatible with multiple different flavors and versions of Python:

- Jython support
- Python 3 compatibility

1.14 Geographic framework

GeoDjango intends to be a world-class geographic Web framework. Its goal is to make it as easy as possible to build GIS Web applications and harness the power of spatially enabled data.

1.15 Common Web application tools

Django offers multiple tools commonly needed in the development of Web applications:

- **Authentication:** Overview | Using the authentication system | Password management | Customizing authentication | API Reference
- Caching
- Logging
- Sending emails
- Syndication feeds (RSS/Atom)
- Pagination
- Messages framework
- Serialization
- Sessions
- Sitemaps
- Static files management

- Data validation

1.16 Other core functionalities

Learn about some other core functionalities of the Django framework:

- Conditional content processing
- Content types and generic relations
- Flatpages
- Redirects
- Signals
- System check framework
- The sites framework
- Unicode in Django

1.17 The Django open-source project

Learn about the development process for the Django project itself and about how you can contribute:

- **Community:** How to get involved | The release process | Team organization | Meet the team | Current roles | The Django source code repository | Security policies | Mailing lists
- **Design philosophies:** Overview
- **Documentation:** About this documentation
- **Third-party distributions:** Overview
- **Django over time:** API stability | Release notes and upgrading instructions | Deprecation Timeline

CHAPTER 2

Getting started

New to Django? Or to Web development in general? Well, you came to the right place: read this material to quickly get up and running.

2.1 Django at a glance

Because Django was developed in a fast-paced newsroom environment, it was designed to make common Web-development tasks fast and easy. Here's an informal overview of how to write a database-driven Web app with Django.

The goal of this document is to give you enough technical specifics to understand how Django works, but this isn't intended to be a tutorial or reference – but we've got both! When you're ready to start a project, you can start with the tutorial or dive right into more detailed documentation.

2.1.1 Design your model

Although you can use Django without a database, it comes with an object-relational mapper in which you describe your database layout in Python code.

The data-model syntax offers many rich ways of representing your models – so far, it's been solving many years' worth of database-schema problems. Here's a quick example:

mysite/news/models.py

```
from django.db import models

class Reporter(models.Model):
    full_name = models.CharField(max_length=70)

    def __str__(self):              # __unicode__ on Python 2
        return self.full_name

class Article(models.Model):
    pub_date = models.DateField()
    headline = models.CharField(max_length=200)
    content = models.TextField()
    reporter = models.ForeignKey(Reporter, on_delete=models.CASCADE)

    def __str__(self):              # __unicode__ on Python 2
        return self.headline
```

2.1.2 Install it

Next, run the Django command-line utility to create the database tables automatically:

```
$ python manage.py migrate
```

The *migrate* command looks at all your available models and creates tables in your database for whichever tables don't already exist, as well as optionally providing much richer schema control.

2.1.3 Enjoy the free API

With that, you've got a free, and rich, Python API to access your data. The API is created on the fly, no code generation necessary:

```
# Import the models we created from our "news" app
>>> from news.models import Reporter, Article

# No reporters are in the system yet.
>>> Reporter.objects.all()
<QuerySet []>

# Create a new Reporter.
>>> r = Reporter(full_name='John Smith')

# Save the object into the database. You have to call save() explicitly.
>>> r.save()

# Now it has an ID.
>>> r.id
1

# Now the new reporter is in the database.
>>> Reporter.objects.all()
<QuerySet [<Reporter: John Smith>]>

# Fields are represented as attributes on the Python object.
>>> r.full_name
'John Smith'

# Django provides a rich database lookup API.
>>> Reporter.objects.get(id=1)
<Reporter: John Smith>
>>> Reporter.objects.get(full_name__startswith='John')
<Reporter: John Smith>
>>> Reporter.objects.get(full_name__contains='mith')
<Reporter: John Smith>
>>> Reporter.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Reporter matching query does not exist.

# Create an article.
>>> from datetime import date
>>> a = Article(pub_date=date.today(), headline='Django is cool',
...     content='Yeah.', reporter=r)
>>> a.save()
```

```
# Now the article is in the database.
>>> Article.objects.all()
<QuerySet [<Article: Django is cool>]>

# Article objects get API access to related Reporter objects.
>>> r = a.reporter
>>> r.full_name
'John Smith'

# And vice versa: Reporter objects get API access to Article objects.
>>> r.article_set.all()
<QuerySet [<Article: Django is cool>]>

# The API follows relationships as far as you need, performing efficient
# JOINs for you behind the scenes.
# This finds all articles by a reporter whose name starts with "John".
>>> Article.objects.filter(reporter__full_name__startswith='John')
<QuerySet [<Article: Django is cool>]>

# Change an object by altering its attributes and calling save().
>>> r.full_name = 'Billy Goat'
>>> r.save()

# Delete an object with delete().
>>> r.delete()
```

2.1.4 A dynamic admin interface: it's not just scaffolding – it's the whole house

Once your models are defined, Django can automatically create a professional, production ready administrative interface – a website that lets authenticated users add, change and delete objects. It's as easy as registering your model in the admin site:

mysite/news/models.py

```
from django.db import models

class Article(models.Model):
    pub_date = models.DateField()
    headline = models.CharField(max_length=200)
    content = models.TextField()
    reporter = models.ForeignKey(Reporter, on_delete=models.CASCADE)
```

mysite/news/admin.py

```
from django.contrib import admin

from . import models

admin.site.register(models.Article)
```

The philosophy here is that your site is edited by a staff, or a client, or maybe just you – and you don't want to have to deal with creating backend interfaces just to manage content.

One typical workflow in creating Django apps is to create models and get the admin sites up and running as fast as possible, so your staff (or clients) can start populating data. Then, develop the way data is presented to the public.

2.1.5 Design your URLs

A clean, elegant URL scheme is an important detail in a high-quality Web application. Django encourages beautiful URL design and doesn't put any cruft in URLs, like `.php` or `.asp`.

To design URLs for an app, you create a Python module called a URLconf. A table of contents for your app, it contains a simple mapping between URL patterns and Python callback functions. URLconfs also serve to decouple URLs from Python code.

Here's what a URLconf might look like for the `Reporter`/`Article` example above:

mysite/news/urls.py

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^articles/([0-9]{4})/$', views.year_archive),
    url(r'^articles/([0-9]{4})/([0-9]{2})/$', views.month_archive),
    url(r'^articles/([0-9]{4})/([0-9]{2})/([0-9]+)/$', views.article_detail),
]
```

The code above maps URLs, as simple regular expressions, to the location of Python callback functions ("views"). The regular expressions use parenthesis to "capture" values from the URLs. When a user requests a page, Django runs through each pattern, in order, and stops at the first one that matches the requested URL. (If none of them matches, Django calls a special-case 404 view.) This is blazingly fast, because the regular expressions are compiled at load time.

Once one of the regexes matches, Django imports and calls the given view, which is a simple Python function. Each view gets passed a request object – which contains request metadata – and the values captured in the regex.

For example, if a user requested the URL "/articles/2005/05/39323/", Django would call the function `news.views.article_detail(request, '2005', '05', '39323')`.

2.1.6 Write your views

Each view is responsible for doing one of two things: Returning an *`HttpResponse`* object containing the content for the requested page, or raising an exception such as *`Http404`*. The rest is up to you.

Generally, a view retrieves data according to the parameters, loads a template and renders the template with the retrieved data. Here's an example view for `year_archive` from above:

mysite/news/views.py

```
from django.shortcuts import render

from .models import Article

def year_archive(request, year):
    a_list = Article.objects.filter(pub_date__year=year)
    context = {'year': year, 'article_list': a_list}
    return render(request, 'news/year_archive.html', context)
```

This example uses Django's template system, which has several powerful features but strives to stay simple enough for non-programmers to use.

2.1.7 Design your templates

The code above loads the `news/year_archive.html` template.

Django has a template search path, which allows you to minimize redundancy among templates. In your Django settings, you specify a list of directories to check for templates with *DIRS*. If a template doesn't exist in the first directory, it checks the second, and so on.

Let's say the `news/year_archive.html` template was found. Here's what that might look like:

`mysite/news/templates/news/year_archive.html`

```
{% extends "base.html" %}

{% block title %}Articles for {{ year }}{% endblock %}

{% block content %}
<h1>Articles for {{ year }}</h1>

{% for article in article_list %}
    <p>{{ article.headline }}</p>
    <p>By {{ article.reporter.full_name }}</p>
    <p>Published {{ article.pub_date|date:"F j, Y" }}</p>
{% endfor %}
{% endblock %}
```

Variables are surrounded by double-curly braces. `{{ article.headline }}` means "Output the value of the article's headline attribute." But dots aren't used only for attribute lookup. They also can do dictionary-key lookup, index lookup and function calls.

Note `{{ article.pub_date|date:"F j, Y" }}` uses a Unix-style "pipe" (the "|" character). This is called a template filter, and it's a way to filter the value of a variable. In this case, the date filter formats a Python datetime object in the given format (as found in PHP's date function).

You can chain together as many filters as you'd like. You can write *custom template filters*. You can write custom template tags, which run custom Python code behind the scenes.

Finally, Django uses the concept of "template inheritance". That's what the `{% extends "base.html" %}` does. It means "First load the template called 'base', which has defined a bunch of blocks, and fill the blocks with the following blocks." In short, that lets you dramatically cut down on redundancy in templates: each template has to define only what's unique to that template.

Here's what the "base.html" template, including the use of static files, might look like:

`mysite/templates/base.html`

```
{% load static %}
<html>
<head>
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    <img src="{% static "images/sitelogo.png" %}" alt="Logo" />
    {% block content %}{% endblock %}
</body>
</html>
```

Simplistically, it defines the look-and-feel of the site (with the site's logo), and provides "holes" for child templates to fill. This makes a site redesign as easy as changing a single file – the base template.

It also lets you create multiple versions of a site, with different base templates, while reusing child templates. Django's creators have used this technique to create strikingly different mobile versions of sites – simply by creating a new base template.

Note that you don't have to use Django's template system if you prefer another system. While Django's template system is particularly well-integrated with Django's model layer, nothing forces you to use it. For that matter, you don't have to use Django's database API, either. You can use another database abstraction layer, you can read XML files, you can read files off disk, or anything you want. Each piece of Django – models, views, templates – is decoupled from the next.

2.1.8 This is just the surface

This has been only a quick overview of Django's functionality. Some more useful features:

- A caching framework that integrates with memcached or other backends.
- A syndication framework that makes creating RSS and Atom feeds as easy as writing a small Python class.
- More sexy automatically-generated admin features – this overview barely scratched the surface.

The next obvious steps are for you to download Django, read the tutorial and join the community. Thanks for your interest!

2.2 Quick install guide

Before you can use Django, you'll need to get it installed. We have a complete installation guide that covers all the possibilities; this guide will guide you to a simple, minimal installation that'll work while you walk through the introduction.

2.2.1 Install Python

Being a Python Web framework, Django requires Python. See *What Python version can I use with Django?* for details. Python includes a lightweight database called SQLite so you won't need to set up a database just yet.

Get the latest version of Python at https://www.python.org/download/ or with your operating system's package manager.

Django on Jython

If you use Jython (a Python implementation for the Java platform), you'll need to follow a few additional steps. See Running Django on Jython for details.

You can verify that Python is installed by typing `python` from your shell; you should see something like:

```
Python 3.4.x
[GCC 4.x] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

2.2.2 Set up a database

This step is only necessary if you'd like to work with a "large" database engine like PostgreSQL, MySQL, or Oracle. To install such a database, consult the *database installation information*.

2.2.3 Remove any old versions of Django

If you are upgrading your installation of Django from a previous version, you will need to *uninstall the old Django version before installing the new version*.

2.2.4 Install Django

You've got three easy options to install Django:

- Install a version of Django provided by your operating system distribution. This is the quickest option for those who have operating systems that distribute Django.
- *Install an official release*. This is the best approach for most users.
- *Install the latest development version*. This option is for enthusiasts who want the latest-and-greatest features and aren't afraid of running brand new code. You might encounter new bugs in the development version, but reporting them helps the development of Django. Also, releases of third-party packages are less likely to be compatible with the development version than with the latest stable release.

Always refer to the documentation that corresponds to the version of Django you're using!

If you do either of the first two steps, keep an eye out for parts of the documentation marked **new in development version**. That phrase flags features that are only available in development versions of Django, and they likely won't work with an official release.

2.2.5 Verifying

To verify that Django can be seen by Python, type `python` from your shell. Then at the Python prompt, try to import Django:

```
>>> import django
>>> print(django.get_version())
1.10
```

You may have another version of Django installed.

2.2.6 That's it!

That's it – you can now move onto the tutorial.

2.3 Writing your first Django app, part 1

Let's learn by example.

Throughout this tutorial, we'll walk you through the creation of a basic poll application.

It'll consist of two parts:

- A public site that lets people view polls and vote in them.
- An admin site that lets you add, change, and delete polls.

We'll assume you have Django installed already. You can tell Django is installed and which version by running the following command:

```
$ python -m django --version
```

If Django is installed, you should see the version of your installation. If it isn't, you'll get an error telling "No module named django".

This tutorial is written for Django 1.10 and Python 3.4 or later. If the Django version doesn't match, you can refer to the tutorial for your version of Django by using the version switcher at the bottom right corner of this page, or update Django to the newest version. If you are still using Python 2.7, you will need to adjust the code samples slightly, as described in comments.

See How to install Django for advice on how to remove older versions of Django and install a newer one.

Where to get help:

If you're having trouble going through this tutorial, please post a message to *django-users* or drop by #django on irc.freenode.net to chat with other Django users who might be able to help.

2.3.1 Creating a project

If this is your first time using Django, you'll have to take care of some initial setup. Namely, you'll need to auto-generate some code that establishes a Django *project* – a collection of settings for an instance of Django, including database configuration, Django-specific options and application-specific settings.

From the command line, `cd` into a directory where you'd like to store your code, then run the following command:

```
$ django-admin startproject mysite
```

This will create a `mysite` directory in your current directory. If it didn't work, see *Problems running django-admin*.

: You'll need to avoid naming projects after built-in Python or Django components. In particular, this means you should avoid using names like `django` (which will conflict with Django itself) or `test` (which conflicts with a built-in Python package).

Where should this code live?

If your background is in plain old PHP (with no use of modern frameworks), you're probably used to putting code under the Web server's document root (in a place such as `/var/www`). With Django, you don't do that. It's not a good idea to put any of this Python code within your Web server's document root, because it risks the possibility that people may be able to view your code over the Web. That's not good for security.

Put your code in some directory **outside** of the document root, such as `/home/mycode`.

Let's look at what `startproject` created:

```
mysite/
    manage.py
    mysite/
        __init__.py
        settings.py
        urls.py
        wsgi.py
```

These files are:

- The outer `mysite/` root directory is just a container for your project. Its name doesn't matter to Django; you can rename it to anything you like.
- `manage.py`: A command-line utility that lets you interact with this Django project in various ways. You can read all the details about `manage.py` in django-admin and manage.py.
- The inner `mysite/` directory is the actual Python package for your project. Its name is the Python package name you'll need to use to import anything inside it (e.g. `mysite.urls`).
- `mysite/__init__.py`: An empty file that tells Python that this directory should be considered a Python package. If you're a Python beginner, read more about packages in the official Python docs.
- `mysite/settings.py`: Settings/configuration for this Django project. Django settings will tell you all about how settings work.
- `mysite/urls.py`: The URL declarations for this Django project; a "table of contents" of your Django-powered site. You can read more about URLs in URL dispatcher.
- `mysite/wsgi.py`: An entry-point for WSGI-compatible web servers to serve your project. See How to deploy with WSGI for more details.

2.3.2 The development server

Let's verify your Django project works. Change into the outer `mysite` directory, if you haven't already, and run the following commands:

```
$ python manage.py runserver
```

You'll see the following output on the command line:

```
Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may not work properly until they are applied.
Run `python manage.py migrate' to apply them.

November 27, 2016 - 15:50:53
Django version 1.10, using settings `mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

: Ignore the warning about unapplied database migrations for now; we'll deal with the database shortly.

You've started the Django development server, a lightweight Web server written purely in Python. We've included this with Django so you can develop things rapidly, without having to deal with configuring a production server – such as Apache – until you're ready for production.

Now's a good time to note: **don't** use this server in anything resembling a production environment. It's intended only for use while developing. (We're in the business of making Web frameworks, not Web servers.)

Now that the server's running, visit http://127.0.0.1:8000/ with your Web browser. You'll see a "Welcome to Django" page, in pleasant, light-blue pastel. It worked!

Changing the port

By default, the *runserver* command starts the development server on the internal IP at port 8000.

If you want to change the server's port, pass it as a command-line argument. For instance, this command starts the server on port 8080:

```
$ python manage.py runserver 8080
```

If you want to change the server's IP, pass it along with the port. So to listen on all public IPs (useful if you want to show off your work on other computers on your network), use:

```
$ python manage.py runserver 0.0.0.0:8000
```

Full docs for the development server can be found in the *runserver* reference.

Automatic reloading of `runserver`

The development server automatically reloads Python code for each request as needed. You don't need to restart the server for code changes to take effect. However, some actions like adding files don't trigger a restart, so you'll have to restart the server in these cases.

2.3.3 Creating the Polls app

Now that your environment – a "project" – is set up, you're set to start doing work.

Each application you write in Django consists of a Python package that follows a certain convention. Django comes with a utility that automatically generates the basic directory structure of an app, so you can focus on writing code rather than creating directories.

Projects vs. apps

What's the difference between a project and an app? An app is a Web application that does something – e.g., a Weblog system, a database of public records or a simple poll app. A project is a collection of configuration and apps for a particular website. A project can contain multiple apps. An app can be in multiple projects.

Your apps can live anywhere on your Python path. In this tutorial, we'll create our poll app right next to your `manage.py` file so that it can be imported as its own top-level module, rather than a submodule of `mysite`.

To create your app, make sure you're in the same directory as `manage.py` and type this command:

```
$ python manage.py startapp polls
```

That'll create a directory `polls`, which is laid out like this:

```
polls/
    __init__.py
    admin.py
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    views.py
```

This directory structure will house the poll application.

2.3.4 Write your first view

Let's write the first view. Open the file `polls/views.py` and put the following Python code in it:

polls/views.py

```
from django.http import HttpResponse


def index(request):
    return HttpResponse("Hello, world. You're at the polls index.")
```

This is the simplest view possible in Django. To call the view, we need to map it to a URL - and for this we need a URLconf.

To create a URLconf in the polls directory, create a file called `urls.py`. Your app directory should now look like:

```
polls/
    __init__.py
    admin.py
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    urls.py
    views.py
```

In the `polls/urls.py` file include the following code:

polls/urls.py

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
]
```

The next step is to point the root URLconf at the `polls.urls` module. In `mysite/urls.py`, add an import for `django.conf.urls.include` and insert an *include()* in the `urlpatterns` list, so you have:

mysite/urls.py

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^polls/', include('polls.urls')),
    url(r'^admin/', admin.site.urls),
]
```

The *include()* function allows referencing other URLconfs. Note that the regular expressions for the *include()* function doesn't have a `$` (end-of-string match character) but rather a trailing slash. Whenever Django encounters *include()*, it chops off whatever part of the URL matched up to that point and sends the remaining string to the included URLconf for further processing.

The idea behind *include()* is to make it easy to plug-and-play URLs. Since polls are in their own URLconf (`polls/urls.py`), they can be placed under "/polls/", or under "/fun_polls/", or under "/content/polls/", or any other path root, and the app will still work.

When to use `include()`

You should always use `include()` when you include other URL patterns. `admin.site.urls` is the only exception to this.

Doesn't match what you see?

If you're seeing `include(admin.site.urls)` instead of just `admin.site.urls`, you're probably using a version of Django that doesn't match this tutorial version. You'll want to either switch to the older tutorial or the newer Django version.

You have now wired an `index` view into the URLconf. Lets verify it's working, run the following command:

```
$ python manage.py runserver
```

Go to http://localhost:8000/polls/ in your browser, and you should see the text "*Hello, world. You're at the polls index.*", which you defined in the `index` view.

The *`url()`* function is passed four arguments, two required: `regex` and `view`, and two optional: `kwargs`, and `name`. At this point, it's worth reviewing what these arguments are for.

`url()` argument: regex

The term "regex" is a commonly used short form meaning "regular expression", which is a syntax for matching patterns in strings, or in this case, url patterns. Django starts at the first regular expression and makes its way down the list, comparing the requested URL against each regular expression until it finds one that matches.

Note that these regular expressions do not search GET and POST parameters, or the domain name. For example, in a request to `https://www.example.com/myapp/`, the URLconf will look for `myapp/`. In a request to `https://www.example.com/myapp/?page=3`, the URLconf will also look for `myapp/`.

If you need help with regular expressions, see Wikipedia's entry and the documentation of the `re` module. Also, the O'Reilly book "Mastering Regular Expressions" by Jeffrey Friedl is fantastic. In practice, however, you don't need to be an expert on regular expressions, as you really only need to know how to capture simple patterns. In fact, complex regexes can have poor lookup performance, so you probably shouldn't rely on the full power of regexes.

Finally, a performance note: these regular expressions are compiled the first time the URLconf module is loaded. They're super fast (as long as the lookups aren't too complex as noted above).

`url()` argument: view

When Django finds a regular expression match, Django calls the specified view function, with an *`HttpRequest`* object as the first argument and any "captured" values from the regular expression as other arguments. If the regex uses simple captures, values are passed as positional arguments; if it uses named captures, values are passed as keyword arguments. We'll give an example of this in a bit.

`url()` argument: kwargs

Arbitrary keyword arguments can be passed in a dictionary to the target view. We aren't going to use this feature of Django in the tutorial.

`url()` argument: name

Naming your URL lets you refer to it unambiguously from elsewhere in Django, especially from within templates. This powerful feature allows you to make global changes to the URL patterns of your project while only touching a single file.

When you're comfortable with the basic request and response flow, read part 2 of this tutorial to start working with the database.

2.4 Writing your first Django app, part 2

This tutorial begins where Tutorial 1 left off. We'll setup the database, create your first model, and get a quick introduction to Django's automatically-generated admin site.

2.4.1 Database setup

Now, open up `mysite/settings.py`. It's a normal Python module with module-level variables representing Django settings.

By default, the configuration uses SQLite. If you're new to databases, or you're just interested in trying Django, this is the easiest choice. SQLite is included in Python, so you won't need to install anything else to support your database. When starting your first real project, however, you may want to use a more scalable database like PostgreSQL, to avoid database-switching headaches down the road.

If you wish to use another database, install the appropriate *database bindings* and change the following keys in the *DATABASES* `'default'` item to match your database connection settings:

- *ENGINE* – Either `'django.db.backends.sqlite3'`, `'django.db.backends.postgresql'`, `'django.db.backends.mysql'`, or `'django.db.backends.oracle'`. Other backends are *also available*.
- *NAME* – The name of your database. If you're using SQLite, the database will be a file on your computer; in that case, *NAME* should be the full absolute path, including filename, of that file. The default value, `os.path.join(BASE_DIR, 'db.sqlite3')`, will store the file in your project directory.

If you are not using SQLite as your database, additional settings such as *USER*, *PASSWORD*, and *HOST* must be added. For more details, see the reference documentation for *DATABASES*.

For databases other than SQLite

If you're using a database besides SQLite, make sure you've created a database by this point. Do that with "`CREATE DATABASE database_name;`" within your database's interactive prompt.

Also make sure that the database user provided in `mysite/settings.py` has "create database" privileges. This allows automatic creation of a *test database* which will be needed in a later tutorial.

If you're using SQLite, you don't need to create anything beforehand - the database file will be created automatically when it is needed.

While you're editing `mysite/settings.py`, set *TIME_ZONE* to your time zone.

Also, note the *INSTALLED_APPS* setting at the top of the file. That holds the names of all Django applications that are activated in this Django instance. Apps can be used in multiple projects, and you can package and distribute them for use by others in their projects.

By default, *INSTALLED_APPS* contains the following apps, all of which come with Django:

- *django.contrib.admin* – The admin site. You'll use it shortly.
- *django.contrib.auth* – An authentication system.
- *django.contrib.contenttypes* – A framework for content types.
- *django.contrib.sessions* – A session framework.
- *django.contrib.messages* – A messaging framework.
- *django.contrib.staticfiles* – A framework for managing static files.

These applications are included by default as a convenience for the common case.

Some of these applications make use of at least one database table, though, so we need to create the tables in the database before we can use them. To do that, run the following command:

```
$ python manage.py migrate
```

The *migrate* command looks at the *INSTALLED_APPS* setting and creates any necessary database tables according to the database settings in your `mysite/settings.py` file and the database migrations shipped with the app (we'll cover those later). You'll see a message for each migration it applies. If you're interested, run the command-line client for your database and type `\dt` (PostgreSQL), `SHOW TABLES;` (MySQL), `.schema` (SQLite), or `SELECT TABLE_NAME FROM USER_TABLES;` (Oracle) to display the tables Django created.

For the minimalists

Like we said above, the default applications are included for the common case, but not everybody needs them. If you don't need any or all of them, feel free to comment-out or delete the appropriate line(s) from *INSTALLED_APPS* before running *migrate*. The *migrate* command will only run migrations for apps in *INSTALLED_APPS*.

2.4.2 Creating models

Now we'll define your models – essentially, your database layout, with additional metadata.

Philosophy

A model is the single, definitive source of truth about your data. It contains the essential fields and behaviors of the data you're storing. Django follows the *DRY Principle*. The goal is to define your data model in one place and automatically derive things from it.

This includes the migrations - unlike in Ruby On Rails, for example, migrations are entirely derived from your models file, and are essentially just a history that Django can roll through to update your database schema to match your current models.

In our simple poll app, we'll create two models: `Question` and `Choice`. A `Question` has a question and a publication date. A `Choice` has two fields: the text of the choice and a vote tally. Each `Choice` is associated with a `Question`.

These concepts are represented by simple Python classes. Edit the `polls/models.py` file so it looks like this:

polls/models.py

```
from django.db import models


class Question(models.Model):
    question_text = models.CharField(max_length=200)
```

```
    pub_date = models.DateTimeField('date published')


class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

The code is straightforward. Each model is represented by a class that subclasses *django.db.models.Model*. Each model has a number of class variables, each of which represents a database field in the model.

Each field is represented by an instance of a *Field* class – e.g., *CharField* for character fields and *DateTimeField* for datetimes. This tells Django what type of data each field holds.

The name of each *Field* instance (e.g. `question_text` or `pub_date`) is the field's name, in machine-friendly format. You'll use this value in your Python code, and your database will use it as the column name.

You can use an optional first positional argument to a *Field* to designate a human-readable name. That's used in a couple of introspective parts of Django, and it doubles as documentation. If this field isn't provided, Django will use the machine-readable name. In this example, we've only defined a human-readable name for `Question.pub_date`. For all other fields in this model, the field's machine-readable name will suffice as its human-readable name.

Some *Field* classes have required arguments. *CharField*, for example, requires that you give it a *max_length*. That's used not only in the database schema, but in validation, as we'll soon see.

A *Field* can also have various optional arguments; in this case, we've set the *default* value of `votes` to 0.

Finally, note a relationship is defined, using *ForeignKey*. That tells Django each `Choice` is related to a single `Question`. Django supports all the common database relationships: many-to-one, many-to-many, and one-to-one.

2.4.3 Activating models

That small bit of model code gives Django a lot of information. With it, Django is able to:

- Create a database schema (`CREATE TABLE` statements) for this app.
- Create a Python database-access API for accessing `Question` and `Choice` objects.

But first we need to tell our project that the `polls` app is installed.

Philosophy

Django apps are "pluggable": You can use an app in multiple projects, and you can distribute apps, because they don't have to be tied to a given Django installation.

To include the app in our project, we need to add a reference to its configuration class in the *INSTALLED_APPS* setting. The `PollsConfig` class is in the `polls/apps.py` file, so its dotted path is `'polls.apps.PollsConfig'`. Edit the `mysite/settings.py` file and add that dotted path to the *INSTALLED_APPS* setting. It'll look like this:

`mysite/settings.py`

```
INSTALLED_APPS = [
    'polls.apps.PollsConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
```

```
    'django.contrib.staticfiles',
]
```

Now Django knows to include the `polls` app. Let's run another command:

```
$ python manage.py makemigrations polls
```

You should see something similar to the following:

```
Migrations for 'polls':
  polls/migrations/0001_initial.py:
    - Create model Choice
    - Create model Question
    - Add field question to choice
```

By running `makemigrations`, you're telling Django that you've made some changes to your models (in this case, you've made new ones) and that you'd like the changes to be stored as a *migration*.

Migrations are how Django stores changes to your models (and thus your database schema) - they're just files on disk. You can read the migration for your new model if you like; it's the file `polls/migrations/0001_initial.py`. Don't worry, you're not expected to read them every time Django makes one, but they're designed to be human-editable in case you want to manually tweak how Django changes things.

There's a command that will run the migrations for you and manage your database schema automatically - that's called *migrate*, and we'll come to it in a moment - but first, let's see what SQL that migration would run. The *sqlmigrate* command takes migration names and returns their SQL:

```
$ python manage.py sqlmigrate polls 0001
```

You should see something similar to the following (we've reformatted it for readability):

```
BEGIN;
--
-- Create model Choice
--
CREATE TABLE "polls_choice" (
    "id" serial NOT NULL PRIMARY KEY,
    "choice_text" varchar(200) NOT NULL,
    "votes" integer NOT NULL
);
--
-- Create model Question
--
CREATE TABLE "polls_question" (
    "id" serial NOT NULL PRIMARY KEY,
    "question_text" varchar(200) NOT NULL,
    "pub_date" timestamp with time zone NOT NULL
);
--
-- Add field question to choice
--
ALTER TABLE "polls_choice" ADD COLUMN "question_id" integer NOT NULL;
ALTER TABLE "polls_choice" ALTER COLUMN "question_id" DROP DEFAULT;
CREATE INDEX "polls_choice_7aa0f6ee" ON "polls_choice" ("question_id");
ALTER TABLE "polls_choice"
  ADD CONSTRAINT "polls_choice_question_id_246c99a640fbbd72_fk_polls_question_id"
    FOREIGN KEY ("question_id")
    REFERENCES "polls_question" ("id")
    DEFERRABLE INITIALLY DEFERRED;
```

```
COMMIT;
```

Note the following:

- The exact output will vary depending on the database you are using. The example above is generated for PostgreSQL.
- Table names are automatically generated by combining the name of the app (`polls`) and the lowercase name of the model – `question` and `choice`. (You can override this behavior.)
- Primary keys (IDs) are added automatically. (You can override this, too.)
- By convention, Django appends `"_id"` to the foreign key field name. (Yes, you can override this, as well.)
- The foreign key relationship is made explicit by a `FOREIGN KEY` constraint. Don't worry about the `DEFERRABLE` parts; that's just telling PostgreSQL to not enforce the foreign key until the end of the transaction.
- It's tailored to the database you're using, so database-specific field types such as `auto_increment` (MySQL), `serial` (PostgreSQL), or `integer primary key autoincrement` (SQLite) are handled for you automatically. Same goes for the quoting of field names – e.g., using double quotes or single quotes.
- The *`sqlmigrate`* command doesn't actually run the migration on your database - it just prints it to the screen so that you can see what SQL Django thinks is required. It's useful for checking what Django is going to do or if you have database administrators who require SQL scripts for changes.

If you're interested, you can also run *`python manage.py check`*; this checks for any problems in your project without making migrations or touching the database.

Now, run *`migrate`* again to create those model tables in your database:

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, polls, sessions
Running migrations:
  Rendering model states... DONE
  Applying polls.0001_initial... OK
```

The *`migrate`* command takes all the migrations that haven't been applied (Django tracks which ones are applied using a special table in your database called `django_migrations`) and runs them against your database - essentially, synchronizing the changes you made to your models with the schema in the database.

Migrations are very powerful and let you change your models over time, as you develop your project, without the need to delete your database or tables and make new ones - it specializes in upgrading your database live, without losing data. We'll cover them in more depth in a later part of the tutorial, but for now, remember the three-step guide to making model changes:

- Change your models (in `models.py`).
- Run *`python manage.py makemigrations`* to create migrations for those changes
- Run *`python manage.py migrate`* to apply those changes to the database.

The reason that there are separate commands to make and apply migrations is because you'll commit migrations to your version control system and ship them with your app; they not only make your development easier, they're also useable by other developers and in production.

Read the django-admin documentation for full information on what the `manage.py` utility can do.

2.4.4 Playing with the API

Now, let's hop into the interactive Python shell and play around with the free API Django gives you. To invoke the Python shell, use this command:

```
$ python manage.py shell
```

We're using this instead of simply typing "python", because `manage.py` sets the `DJANGO_SETTINGS_MODULE` environment variable, which gives Django the Python import path to your `mysite/settings.py` file.

Bypassing manage.py

If you'd rather not use `manage.py`, no problem. Just set the *DJANGO_SETTINGS_MODULE* environment variable to `mysite.settings`, start a plain Python shell, and set up Django:

```
>>> import django
>>> django.setup()
```

If this raises an `AttributeError`, you're probably using a version of Django that doesn't match this tutorial version. You'll want to either switch to the older tutorial or the newer Django version.

You must run `python` from the same directory `manage.py` is in, or ensure that directory is on the Python path, so that `import mysite` works.

For more information on all of this, see the django-admin documentation.

Once you're in the shell, explore the database API:

```
>>> from polls.models import Question, Choice   # Import the model classes we just wrote.

# No questions are in the system yet.
>>> Question.objects.all()
<QuerySet []>

# Create a new Question.
# Support for time zones is enabled in the default settings file, so
# Django expects a datetime with tzinfo for pub_date. Use timezone.now()
# instead of datetime.datetime.now() and it will do the right thing.
>>> from django.utils import timezone
>>> q = Question(question_text="What's new?", pub_date=timezone.now())

# Save the object into the database. You have to call save() explicitly.
>>> q.save()

# Now it has an ID. Note that this might say "1L" instead of "1", depending
# on which database you're using. That's no biggie; it just means your
# database backend prefers to return integers as Python long integer
# objects.
>>> q.id
1

# Access model field values via Python attributes.
>>> q.question_text
"What's new?"
>>> q.pub_date
datetime.datetime(2012, 2, 26, 13, 0, 0, 775217, tzinfo=<UTC>)

# Change values by changing the attributes, then calling save().
```

```
>>> q.question_text = "What's up?"
>>> q.save()

# objects.all() displays all the questions in the database.
>>> Question.objects.all()
<QuerySet [<Question: Question object>]>
```

Wait a minute. `<Question: Question object>` is, utterly, an unhelpful representation of this object. Let's fix that by editing the `Question` model (in the `polls/models.py` file) and adding a *`__str__()`* method to both `Question` and `Choice`:

`polls/models.py`

```
from django.db import models
from django.utils.encoding import python_2_unicode_compatible

@python_2_unicode_compatible  # only if you need to support Python 2
class Question(models.Model):
    # ...
    def __str__(self):
        return self.question_text

@python_2_unicode_compatible  # only if you need to support Python 2
class Choice(models.Model):
    # ...
    def __str__(self):
        return self.choice_text
```

It's important to add *`__str__()`* methods to your models, not only for your own convenience when dealing with the interactive prompt, but also because objects' representations are used throughout Django's automatically-generated admin.

Note these are normal Python methods. Let's add a custom method, just for demonstration:

`polls/models.py`

```
import datetime

from django.db import models
from django.utils import timezone


class Question(models.Model):
    # ...
    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)
```

Note the addition of `import datetime` and `from django.utils import timezone`, to reference Python's standard `datetime` module and Django's time-zone-related utilities in *`django.utils.timezone`*, respectively. If you aren't familiar with time zone handling in Python, you can learn more in the time zone support docs.

Save these changes and start a new Python interactive shell by running `python manage.py shell` again:

```
>>> from polls.models import Question, Choice

# Make sure our __str__() addition worked.
>>> Question.objects.all()
<QuerySet [<Question: What's up?>]>
```

```
# Django provides a rich database lookup API that's entirely driven by
# keyword arguments.
>>> Question.objects.filter(id=1)
<QuerySet [<Question: What's up?>]>
>>> Question.objects.filter(question_text__startswith='What')
<QuerySet [<Question: What's up?>]>

# Get the question that was published this year.
>>> from django.utils import timezone
>>> current_year = timezone.now().year
>>> Question.objects.get(pub_date__year=current_year)
<Question: What's up?>

# Request an ID that doesn't exist, this will raise an exception.
>>> Question.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Question matching query does not exist.

# Lookup by a primary key is the most common case, so Django provides a
# shortcut for primary-key exact lookups.
# The following is identical to Question.objects.get(id=1).
>>> Question.objects.get(pk=1)
<Question: What's up?>

# Make sure our custom method worked.
>>> q = Question.objects.get(pk=1)
>>> q.was_published_recently()
True

# Give the Question a couple of Choices. The create call constructs a new
# Choice object, does the INSERT statement, adds the choice to the set
# of available choices and returns the new Choice object. Django creates
# a set to hold the "other side" of a ForeignKey relation
# (e.g. a question's choice) which can be accessed via the API.
>>> q = Question.objects.get(pk=1)

# Display any choices from the related object set -- none so far.
>>> q.choice_set.all()
<QuerySet []>

# Create three choices.
>>> q.choice_set.create(choice_text='Not much', votes=0)
<Choice: Not much>
>>> q.choice_set.create(choice_text='The sky', votes=0)
<Choice: The sky>
>>> c = q.choice_set.create(choice_text='Just hacking again', votes=0)

# Choice objects have API access to their related Question objects.
>>> c.question
<Question: What's up?>

# And vice versa: Question objects get access to Choice objects.
>>> q.choice_set.all()
<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]>
>>> q.choice_set.count()
3
```

```
# The API automatically follows relationships as far as you need.
# Use double underscores to separate relationships.
# This works as many levels deep as you want; there's no limit.
# Find all Choices for any question whose pub_date is in this year
# (reusing the 'current_year' variable we created above).
>>> Choice.objects.filter(question__pub_date__year=current_year)
<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]>

# Let's delete one of the choices. Use delete() for that.
>>> c = q.choice_set.filter(choice_text__startswith='Just hacking')
>>> c.delete()
```

For more information on model relations, see Accessing related objects. For more on how to use double underscores to perform field lookups via the API, see *Field lookups*. For full details on the database API, see our Database API reference.

2.4.5 Introducing the Django Admin

Philosophy

Generating admin sites for your staff or clients to add, change, and delete content is tedious work that doesn't require much creativity. For that reason, Django entirely automates creation of admin interfaces for models.

Django was written in a newsroom environment, with a very clear separation between "content publishers" and the "public" site. Site managers use the system to add news stories, events, sports scores, etc., and that content is displayed on the public site. Django solves the problem of creating a unified interface for site administrators to edit content.

The admin isn't intended to be used by site visitors. It's for site managers.

Creating an admin user

First we'll need to create a user who can login to the admin site. Run the following command:

```
$ python manage.py createsuperuser
```

Enter your desired username and press enter.

```
Username: admin
```

You will then be prompted for your desired email address:

```
Email address: admin@example.com
```

The final step is to enter your password. You will be asked to enter your password twice, the second time as a confirmation of the first.

```
Password: **********
Password (again): *********
Superuser created successfully.
```

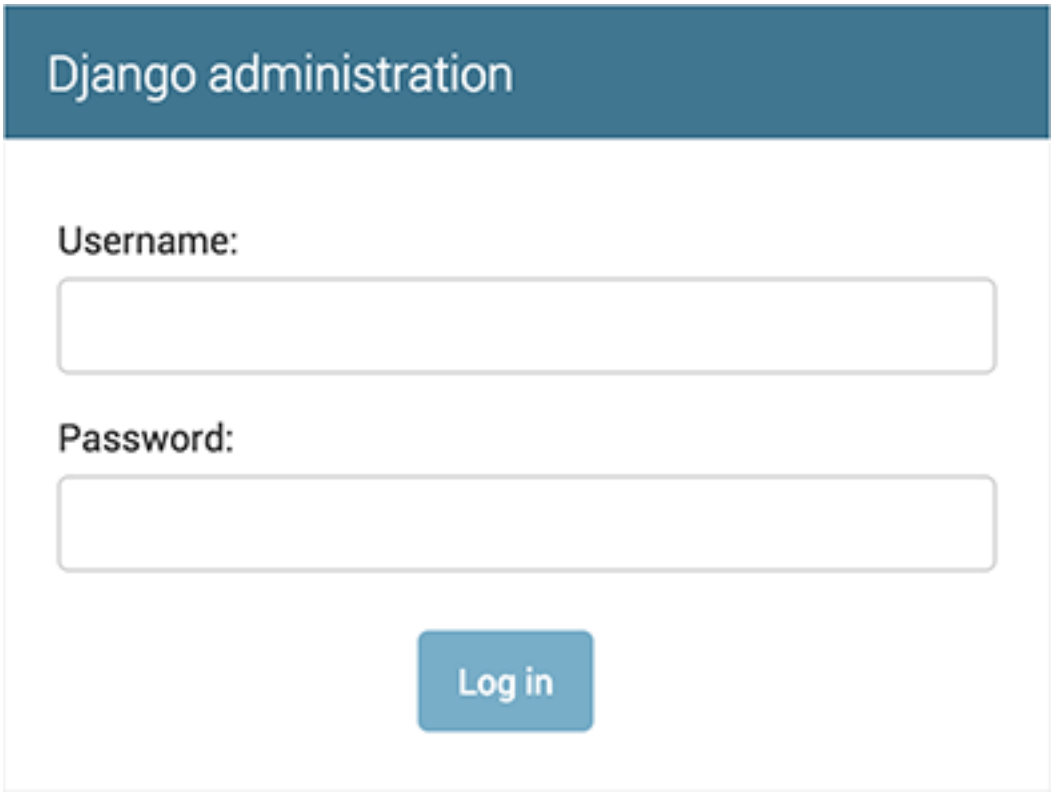
Start the development server

The Django admin site is activated by default. Let's start the development server and explore it.

If the server is not running start it like so:

```
$ python manage.py runserver
```

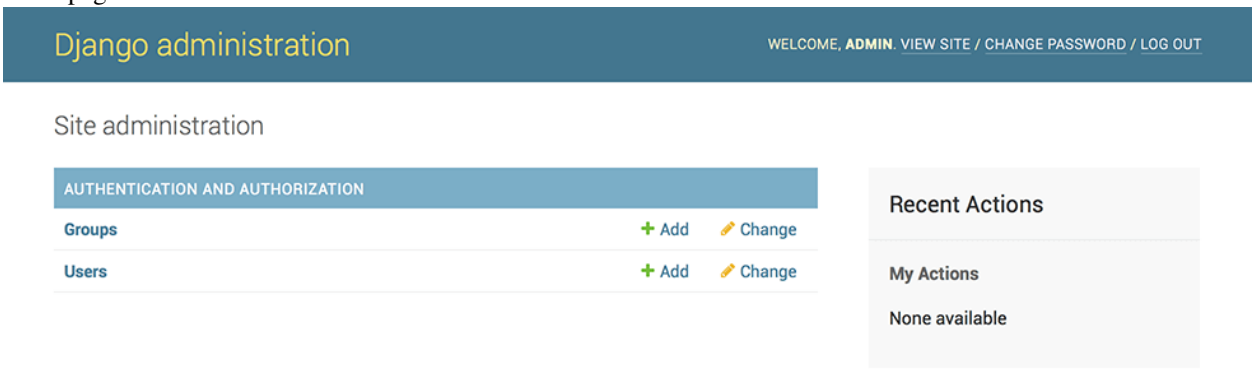
Now, open a Web browser and go to "/admin/" on your local domain – e.g., http://127.0.0.1:8000/admin/. You should see the admin's login screen:



Since translation is turned on by default, the login screen may be displayed in your own language, depending on your browser's settings and if Django has a translation for this language.

Enter the admin site

Now, try logging in with the superuser account you created in the previous step. You should see the Django admin index page:



You should see a few types of editable content: groups and users. They are provided by `django.contrib.auth`, the authentication framework shipped by Django.

Make the poll app modifiable in the admin

But where's our poll app? It's not displayed on the admin index page.

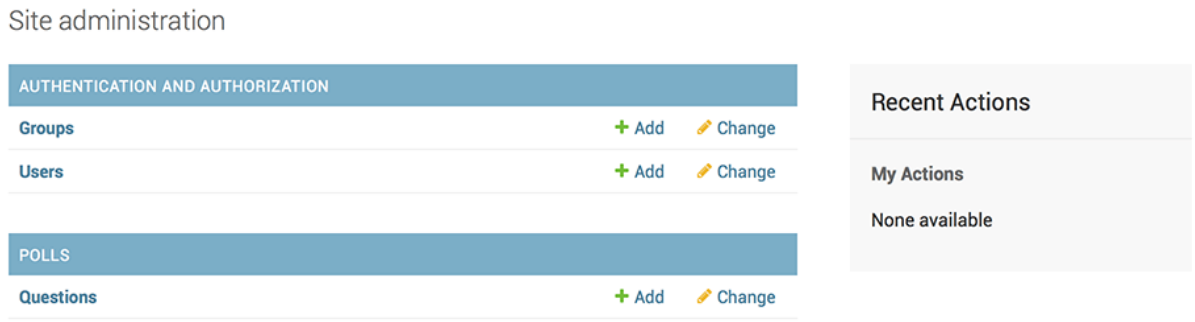
Just one thing to do: we need to tell the admin that `Question` objects have an admin interface. To do this, open the `polls/admin.py` file, and edit it to look like this:

`polls/admin.py`

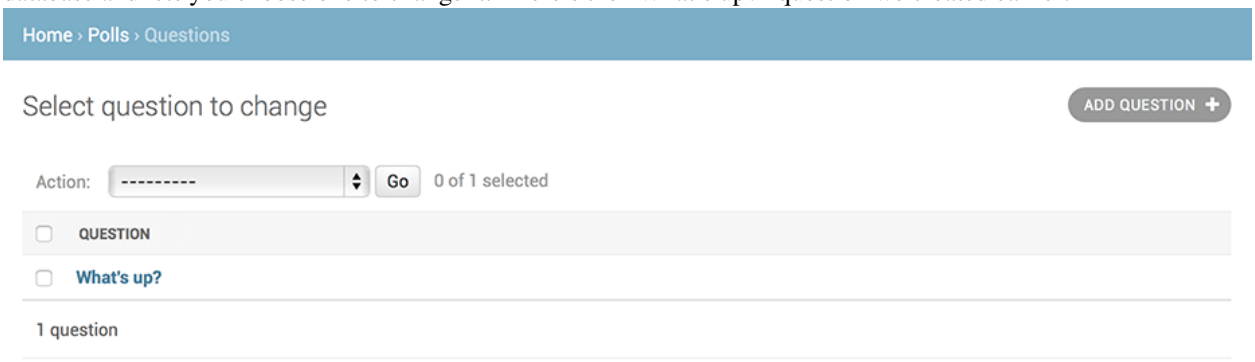
```
from django.contrib import admin

from .models import Question

admin.site.register(Question)
```

Explore the free admin functionality

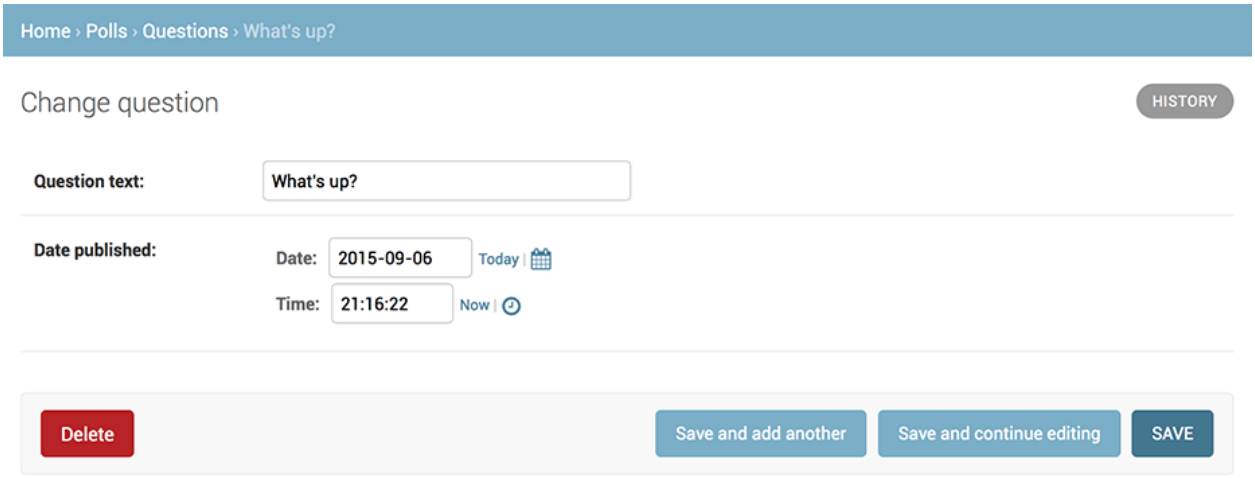
Now that we've registered `Question`, Django knows that it should be displayed on the admin index page:



Click "Questions". Now you're at the "change list" page for questions. This page displays all the questions in the database and lets you choose one to change it. There's the "What's up?" question we created earlier:



Click the "What's up?" question to edit it:



Things to note here:

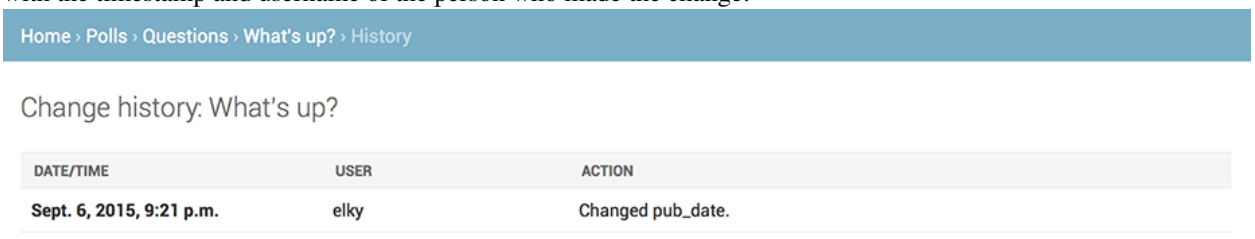
- The form is automatically generated from the `Question` model.
- The different model field types (*`DateTimeField`*, *`CharField`*) correspond to the appropriate HTML input widget. Each type of field knows how to display itself in the Django admin.
- Each *`DateTimeField`* gets free JavaScript shortcuts. Dates get a "Today" shortcut and calendar popup, and times get a "Now" shortcut and a convenient popup that lists commonly entered times.

The bottom part of the page gives you a couple of options:

- Save – Saves changes and returns to the change-list page for this type of object.
- Save and continue editing – Saves changes and reloads the admin page for this object.
- Save and add another – Saves changes and loads a new, blank form for this type of object.
- Delete – Displays a delete confirmation page.

If the value of "Date published" doesn't match the time when you created the question in Tutorial 1, it probably means you forgot to set the correct value for the *`TIME_ZONE`* setting. Change it, reload the page and check that the correct value appears.

Change the "Date published" by clicking the "Today" and "Now" shortcuts. Then click "Save and continue editing." Then click "History" in the upper right. You'll see a page listing all changes made to this object via the Django admin, with the timestamp and username of the person who made the change:



When you're comfortable with the models API and have familiarized yourself with the admin site, read part 3 of this tutorial to learn about how to add more views to our polls app.

2.5 Writing your first Django app, part 3

This tutorial begins where Tutorial 2 left off. We're continuing the Web-poll application and will focus on creating the public interface – "views."

2.5.1 Overview

A view is a "type" of Web page in your Django application that generally serves a specific function and has a specific template. For example, in a blog application, you might have the following views:

- Blog homepage – displays the latest few entries.
- Entry "detail" page – permalink page for a single entry.
- Year-based archive page – displays all months with entries in the given year.
- Month-based archive page – displays all days with entries in the given month.
- Day-based archive page – displays all entries in the given day.
- Comment action – handles posting comments to a given entry.

In our poll application, we'll have the following four views:

- Question "index" page – displays the latest few questions.
- Question "detail" page – displays a question text, with no results but with a form to vote.
- Question "results" page – displays results for a particular question.
- Vote action – handles voting for a particular choice in a particular question.

In Django, web pages and other content are delivered by views. Each view is represented by a simple Python function (or method, in the case of class-based views). Django will choose a view by examining the URL that's requested (to be precise, the part of the URL after the domain name).

Now in your time on the web you may have come across such beauties as "ME2/Sites/dirmod.asp?sid=&type=gen&mod=Core+Pages&gid=A6CD4967199A42D9B65B1B". You will be pleased to know that Django allows us much more elegant *URL patterns* than that.

A URL pattern is simply the general form of a URL - for example: `/newsarchive/<year>/<month>/`.

To get from a URL to a view, Django uses what are known as 'URLconfs'. A URLconf maps URL patterns (described as regular expressions) to views.

This tutorial provides basic instruction in the use of URLconfs, and you can refer to *`django.urls`* for more information.

2.5.2 Writing more views

Now let's add a few more views to `polls/views.py`. These views are slightly different, because they take an argument:

`polls/views.py`

```
def detail(request, question_id):
    return HttpResponse("You're looking at question %s." % question_id)

def results(request, question_id):
    response = "You're looking at the results of question %s."
    return HttpResponse(response % question_id)
```

```
def vote(request, question_id):
    return HttpResponse("You're voting on question %s." % question_id)
```

Wire these new views into the `polls.urls` module by adding the following *`url()`* calls:

polls/urls.py

```
from django.conf.urls import url

from . import views

urlpatterns = [
    # ex: /polls/
    url(r'^$', views.index, name='index'),
    # ex: /polls/5/
    url(r'^(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
    # ex: /polls/5/results/
    url(r'^(?P<question_id>[0-9]+)/results/$', views.results, name='results'),
    # ex: /polls/5/vote/
    url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
]
```

Take a look in your browser, at "/polls/34/". It'll run the `detail()` method and display whatever ID you provide in the URL. Try "/polls/34/results/" and "/polls/34/vote/" too – these will display the placeholder results and voting pages.

When somebody requests a page from your website – say, "/polls/34/", Django will load the `mysite.urls` Python module because it's pointed to by the *`ROOT_URLCONF`* setting. It finds the variable named `urlpatterns` and traverses the regular expressions in order. After finding the match at `'^polls/'`, it strips off the matching text (`"polls/"`) and sends the remaining text – `"34/"` – to the 'polls.urls' URLconf for further processing. There it matches `r'^(?P<question_id>[0-9]+)/$'`, resulting in a call to the `detail()` view like so:

```
detail(request=<HttpRequest object>, question_id='34')
```

The `question_id='34'` part comes from `(?P<question_id>[0-9]+)`. Using parentheses around a pattern "captures" the text matched by that pattern and sends it as an argument to the view function; `?P<question_id>` defines the name that will be used to identify the matched pattern; and `[0-9]+` is a regular expression to match a sequence of digits (i.e., a number).

Because the URL patterns are regular expressions, there really is no limit on what you can do with them. And there's no need to add URL cruft such as `.html` – unless you want to, in which case you can do something like this:

```
url(r'^polls/latest\.html$', views.index),
```

But, don't do that. It's silly.

2.5.3 Write views that actually do something

Each view is responsible for doing one of two things: returning an *`HttpResponse`* object containing the content for the requested page, or raising an exception such as *`Http404`*. The rest is up to you.

Your view can read records from a database, or not. It can use a template system such as Django's – or a third-party Python template system – or not. It can generate a PDF file, output XML, create a ZIP file on the fly, anything you want, using whatever Python libraries you want.

All Django wants is that *`HttpResponse`*. Or an exception.

Because it's convenient, let's use Django's own database API, which we covered in Tutorial 2. Here's one stab at a new `index()` view, which displays the latest 5 poll questions in the system, separated by commas, according to publication date:

polls/views.py

```
from django.http import HttpResponse

from .models import Question


def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    output = ', '.join([q.question_text for q in latest_question_list])
    return HttpResponse(output)

# Leave the rest of the views (detail, results, vote) unchanged
```

There's a problem here, though: the page's design is hard-coded in the view. If you want to change the way the page looks, you'll have to edit this Python code. So let's use Django's template system to separate the design from Python by creating a template that the view can use.

First, create a directory called `templates` in your `polls` directory. Django will look for templates in there.

Your project's *TEMPLATES* setting describes how Django will load and render templates. The default settings file configures a `DjangoTemplates` backend whose *APP_DIRS* option is set to `True`. By convention `DjangoTemplates` looks for a "templates" subdirectory in each of the *INSTALLED_APPS*.

Within the `templates` directory you have just created, create another directory called `polls`, and within that create a file called `index.html`. In other words, your template should be at `polls/templates/polls/index.html`. Because of how the `app_directories` template loader works as described above, you can refer to this template within Django simply as `polls/index.html`.

Template namespacing

Now we *might* be able to get away with putting our templates directly in `polls/templates` (rather than creating another `polls` subdirectory), but it would actually be a bad idea. Django will choose the first template it finds whose name matches, and if you had a template with the same name in a *different* application, Django would be unable to distinguish between them. We need to be able to point Django at the right one, and the easiest way to ensure this is by *namespacing* them. That is, by putting those templates inside *another* directory named for the application itself.

Put the following code in that template:

polls/templates/polls/index.html

```
{% if latest_question_list %}
    <ul>
    {% for question in latest_question_list %}
        <li><a href="/polls/{{ question.id }}/">{{ question.question_text }}</a></li>
    {% endfor %}
    </ul>
{% else %}
    <p>No polls are available.</p>
{% endif %}
```

Now let's update our `index` view in `polls/views.py` to use the template:

polls/views.py

```
from django.http import HttpResponse
from django.template import loader

from .models import Question


def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    template = loader.get_template('polls/index.html')
    context = {
        'latest_question_list': latest_question_list,
    }
    return HttpResponse(template.render(context, request))
```

That code loads the template called `polls/index.html` and passes it a context. The context is a dictionary mapping template variable names to Python objects.

Load the page by pointing your browser at "/polls/", and you should see a bulleted-list containing the "What's up" question from Tutorial 2. The link points to the question's detail page.

A shortcut: `render()`

It's a very common idiom to load a template, fill a context and return an *`HttpResponse`* object with the result of the rendered template. Django provides a shortcut. Here's the full `index()` view, rewritten:

`polls/views.py`

```
from django.shortcuts import render

from .models import Question


def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    context = {'latest_question_list': latest_question_list}
    return render(request, 'polls/index.html', context)
```

Note that once we've done this in all these views, we no longer need to import *`loader`* and *`HttpResponse`* (you'll want to keep `HttpResponse` if you still have the stub methods for `detail`, `results`, and `vote`).

The *`render()`* function takes the request object as its first argument, a template name as its second argument and a dictionary as its optional third argument. It returns an *`HttpResponse`* object of the given template rendered with the given context.

2.5.4 Raising a 404 error

Now, let's tackle the question detail view – the page that displays the question text for a given poll. Here's the view:

`polls/views.py`

```
from django.http import Http404
from django.shortcuts import render

from .models import Question
# ...
def detail(request, question_id):
    try:
```

```
        question = Question.objects.get(pk=question_id)
    except Question.DoesNotExist:
        raise Http404("Question does not exist")
    return render(request, 'polls/detail.html', {'question': question})
```

The new concept here: The view raises the *Http404* exception if a question with the requested ID doesn't exist.

We'll discuss what you could put in that `polls/detail.html` template a bit later, but if you'd like to quickly get the above example working, a file containing just:

`polls/templates/polls/detail.html`

```
{{ question }}
```

will get you started for now.

A shortcut: `get_object_or_404()`

It's a very common idiom to use *get()* and raise *Http404* if the object doesn't exist. Django provides a shortcut. Here's the `detail()` view, rewritten:

`polls/views.py`

```
from django.shortcuts import get_object_or_404, render

from .models import Question
# ...
def detail(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    return render(request, 'polls/detail.html', {'question': question})
```

The *get_object_or_404()* function takes a Django model as its first argument and an arbitrary number of keyword arguments, which it passes to the *get()* function of the model's manager. It raises *Http404* if the object doesn't exist.

Philosophy

Why do we use a helper function *get_object_or_404()* instead of automatically catching the *ObjectDoesNotExist* exceptions at a higher level, or having the model API raise *Http404* instead of *ObjectDoesNotExist*?

Because that would couple the model layer to the view layer. One of the foremost design goals of Django is to maintain loose coupling. Some controlled coupling is introduced in the *django.shortcuts* module.

There's also a *get_list_or_404()* function, which works just as *get_object_or_404()* – except using *filter()* instead of *get()*. It raises *Http404* if the list is empty.

2.5.5 Use the template system

Back to the `detail()` view for our poll application. Given the context variable `question`, here's what the `polls/detail.html` template might look like:

`polls/templates/polls/detail.html`

```
<h1>{{ question.question_text }}</h1>
<ul>
{% for choice in question.choice_set.all %}
    <li>{{ choice.choice_text }}</li>
{% endfor %}
</ul>
```

The template system uses dot-lookup syntax to access variable attributes. In the example of `{{ question.question_text }}`, first Django does a dictionary lookup on the object `question`. Failing that, it tries an attribute lookup – which works, in this case. If attribute lookup had failed, it would've tried a list-index lookup.

Method-calling happens in the *{% for %}* loop: `question.choice_set.all` is interpreted as the Python code `question.choice_set.all()`, which returns an iterable of `Choice` objects and is suitable for use in the *{% for %}* tag.

See the template guide for more about templates.

2.5.6 Removing hardcoded URLs in templates

Remember, when we wrote the link to a question in the `polls/index.html` template, the link was partially hardcoded like this:

```
<li><a href="/polls/{{ question.id }}/">{{ question.question_text }}</a></li>
```

The problem with this hardcoded, tightly-coupled approach is that it becomes challenging to change URLs on projects with a lot of templates. However, since you defined the name argument in the *url()* functions in the `polls.urls` module, you can remove a reliance on specific URL paths defined in your url configurations by using the `{% url %}` template tag:

```
<li><a href="{% url 'detail' question.id %}">{{ question.question_text }}</a></li>
```

The way this works is by looking up the URL definition as specified in the `polls.urls` module. You can see exactly where the URL name of 'detail' is defined below:

```
...
# the 'name' value as called by the {% url %} template tag
url(r'^(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
...
```

If you want to change the URL of the polls detail view to something else, perhaps to something like `polls/specifics/12/` instead of doing it in the template (or templates) you would change it in `polls/urls.py`:

```
...
# added the word 'specifics'
url(r'^specifics/(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
...
```

2.5.7 Namespacing URL names

The tutorial project has just one app, `polls`. In real Django projects, there might be five, ten, twenty apps or more. How does Django differentiate the URL names between them? For example, the `polls` app has a `detail` view, and so might an app on the same project that is for a blog. How does one make it so that Django knows which app view to create for a url when using the `{% url %}` template tag?

The answer is to add namespaces to your URLconf. In the `polls/urls.py` file, go ahead and add an `app_name` to set the application namespace:

polls/urls.py

```
from django.conf.urls import url

from . import views

app_name = 'polls'
urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^(?P<question_id>[0-9]+)/$', views.detail, name='detail'),
    url(r'^(?P<question_id>[0-9]+)/results/$', views.results, name='results'),
    url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
]
```

Now change your `polls/index.html` template from:

polls/templates/polls/index.html

```
<li><a href="{% url 'detail' question.id %}">{{ question.question_text }}</a></li>
```

to point at the namespaced detail view:

polls/templates/polls/index.html

```
<li><a href="{% url 'polls:detail' question.id %}">{{ question.question_text }}</a></li>
```

When you're comfortable with writing views, read part 4 of this tutorial to learn about simple form processing and generic views.

2.6 Writing your first Django app, part 4

This tutorial begins where Tutorial 3 left off. We're continuing the Web-poll application and will focus on simple form processing and cutting down our code.

2.6.1 Write a simple form

Let's update our poll detail template ("polls/detail.html") from the last tutorial, so that the template contains an HTML `<form>` element:

polls/templates/polls/detail.html

```
<h1>{{ question.question_text }}</h1>

{% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}

<form action="{% url 'polls:vote' question.id %}" method="post">
{% csrf_token %}
{% for choice in question.choice_set.all %}
    <input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice.id }}" />
    <label for="choice{{ forloop.counter }}">{{ choice.choice_text }}</label><br />
{% endfor %}
<input type="submit" value="Vote" />
</form>
```

A quick rundown:

- The above template displays a radio button for each question choice. The `value` of each radio button is the associated question choice's ID. The `name` of each radio button is `"choice"`. That means, when somebody

selects one of the radio buttons and submits the form, it'll send the POST data `choice=#` where # is the ID of the selected choice. This is the basic concept of HTML forms.

- We set the form's `action` to `{% url 'polls:vote' question.id %}`, and we set `method="post"`. Using `method="post"` (as opposed to `method="get"`) is very important, because the act of submitting this form will alter data server-side. Whenever you create a form that alters data server-side, use `method="post"`. This tip isn't specific to Django; it's just good Web development practice.
- `forloop.counter` indicates how many times the *for* tag has gone through its loop
- Since we're creating a POST form (which can have the effect of modifying data), we need to worry about Cross Site Request Forgeries. Thankfully, you don't have to worry too hard, because Django comes with a very easy-to-use system for protecting against it. In short, all POST forms that are targeted at internal URLs should use the *{% csrf_token %}* template tag.

Now, let's create a Django view that handles the submitted data and does something with it. Remember, in Tutorial 3, we created a URLconf for the polls application that includes this line:

polls/urls.py

```
url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
```

We also created a dummy implementation of the `vote()` function. Let's create a real version. Add the following to `polls/views.py`:

polls/views.py

```
from django.shortcuts import get_object_or_404, render
from django.http import HttpResponseRedirect, HttpResponse
from django.urls import reverse

from .models import Choice, Question
# ...
def vote(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    try:
        selected_choice = question.choice_set.get(pk=request.POST['choice'])
    except (KeyError, Choice.DoesNotExist):
        # Redisplay the question voting form.
        return render(request, 'polls/detail.html', {
            'question': question,
            'error_message': "You didn't select a choice.",
        })
    else:
        selected_choice.votes += 1
        selected_choice.save()
        # Always return an HttpResponseRedirect after successfully dealing
        # with POST data. This prevents data from being posted twice if a
        # user hits the Back button.
        return HttpResponseRedirect(reverse('polls:results', args=(question.id,)))
```

This code includes a few things we haven't covered yet in this tutorial:

- *request.POST* is a dictionary-like object that lets you access submitted data by key name. In this case, `request.POST['choice']` returns the ID of the selected choice, as a string. *request.POST* values are always strings.

  Note that Django also provides *request.GET* for accessing GET data in the same way – but we're explicitly using *request.POST* in our code, to ensure that data is only altered via a POST call.

- `request.POST['choice']` will raise `KeyError` if `choice` wasn't provided in POST data. The above code checks for `KeyError` and redisplays the question form with an error message if `choice` isn't given.
- After incrementing the choice count, the code returns an *`HttpResponseRedirect`* rather than a normal *`HttpResponse`*. *`HttpResponseRedirect`* takes a single argument: the URL to which the user will be redirected (see the following point for how we construct the URL in this case).

  As the Python comment above points out, you should always return an *`HttpResponseRedirect`* after successfully dealing with POST data. This tip isn't specific to Django; it's just good Web development practice.
- We are using the *`reverse()`* function in the *`HttpResponseRedirect`* constructor in this example. This function helps avoid having to hardcode a URL in the view function. It is given the name of the view that we want to pass control to and the variable portion of the URL pattern that points to that view. In this case, using the URLconf we set up in Tutorial 3, this *`reverse()`* call will return a string like

```
'/polls/3/results/'
```

  where the 3 is the value of `question.id`. This redirected URL will then call the `'results'` view to display the final page.

As mentioned in Tutorial 3, `request` is an *`HttpRequest`* object. For more on *`HttpRequest`* objects, see the request and response documentation.

After somebody votes in a question, the `vote()` view redirects to the results page for the question. Let's write that view:

polls/views.py

```
from django.shortcuts import get_object_or_404, render


def results(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    return render(request, 'polls/results.html', {'question': question})
```

This is almost exactly the same as the `detail()` view from Tutorial 3. The only difference is the template name. We'll fix this redundancy later.

Now, create a `polls/results.html` template:

polls/templates/polls/results.html

```
<h1>{{ question.question_text }}</h1>

<ul>
{% for choice in question.choice_set.all %}
    <li>{{ choice.choice_text }} -- {{ choice.votes }} vote{{ choice.votes|pluralize }}</li>
{% endfor %}
</ul>

<a href="{% url 'polls:detail' question.id %}">Vote again?</a>
```

Now, go to `/polls/1/` in your browser and vote in the question. You should see a results page that gets updated each time you vote. If you submit the form without having chosen a choice, you should see the error message.

: The code for our `vote()` view does have a small problem. It first gets the `selected_choice` object from the database, then computes the new value of `votes`, and then saves it back to the database. If two users of your website try to vote at *exactly the same time*, this might go wrong: The same value, let's say 42, will be retrieved for `votes`. Then, for both users the new value of 43 is computed and saved, but 44 would be the expected value.

This is called a *race condition*. If you are interested, you can read *Avoiding race conditions using F()* to learn how you can solve this issue.

2.6.2 Use generic views: Less code is better

The `detail()` (from Tutorial 3) and `results()` views are very simple – and, as mentioned above, redundant. The `index()` view, which displays a list of polls, is similar.

These views represent a common case of basic Web development: getting data from the database according to a parameter passed in the URL, loading a template and returning the rendered template. Because this is so common, Django provides a shortcut, called the "generic views" system.

Generic views abstract common patterns to the point where you don't even need to write Python code to write an app.

Let's convert our poll app to use the generic views system, so we can delete a bunch of our own code. We'll just have to take a few steps to make the conversion. We will:

1. Convert the URLconf.
2. Delete some of the old, unneeded views.
3. Introduce new views based on Django's generic views.

Read on for details.

Why the code-shuffle?

Generally, when writing a Django app, you'll evaluate whether generic views are a good fit for your problem, and you'll use them from the beginning, rather than refactoring your code halfway through. But this tutorial intentionally has focused on writing the views "the hard way" until now, to focus on core concepts.

You should know basic math before you start using a calculator.

Amend URLconf

First, open the `polls/urls.py` URLconf and change it like so:

polls/urls.py

```
from django.conf.urls import url

from . import views

app_name = 'polls'
urlpatterns = [
    url(r'^$', views.IndexView.as_view(), name='index'),
    url(r'^(?P<pk>[0-9]+)/$', views.DetailView.as_view(), name='detail'),
    url(r'^(?P<pk>[0-9]+)/results/$', views.ResultsView.as_view(), name='results'),
    url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
]
```

Note that the name of the matched pattern in the regexes of the second and third patterns has changed from `<question_id>` to `<pk>`.

Amend views

Next, we're going to remove our old `index`, `detail`, and `results` views and use Django's generic views instead. To do so, open the `polls/views.py` file and change it like so:

polls/views.py

```
from django.shortcuts import get_object_or_404, render
from django.http import HttpResponseRedirect
from django.urls import reverse
from django.views import generic

from .models import Choice, Question


class IndexView(generic.ListView):
    template_name = 'polls/index.html'
    context_object_name = 'latest_question_list'

    def get_queryset(self):
        """Return the last five published questions."""
        return Question.objects.order_by('-pub_date')[:5]


class DetailView(generic.DetailView):
    model = Question
    template_name = 'polls/detail.html'


class ResultsView(generic.DetailView):
    model = Question
    template_name = 'polls/results.html'


def vote(request, question_id):
    ... # same as above, no changes needed.
```

We're using two generic views here: *ListView* and *DetailView*. Respectively, those two views abstract the concepts of "display a list of objects" and "display a detail page for a particular type of object."

- Each generic view needs to know what model it will be acting upon. This is provided using the `model` attribute.
- The *DetailView* generic view expects the primary key value captured from the URL to be called `"pk"`, so we've changed `question_id` to `pk` for the generic views.

By default, the *DetailView* generic view uses a template called `<app name>/<model name>_detail.html`. In our case, it would use the template `"polls/question_detail.html"`. The `template_name` attribute is used to tell Django to use a specific template name instead of the autogenerated default template name. We also specify the `template_name` for the `results` list view – this ensures that the results view and the detail view have a different appearance when rendered, even though they're both a *DetailView* behind the scenes.

Similarly, the *ListView* generic view uses a default template called `<app name>/<model name>_list.html`; we use `template_name` to tell *ListView* to use our existing `"polls/index.html"` template.

In previous parts of the tutorial, the templates have been provided with a context that contains the `question` and `latest_question_list` context variables. For `DetailView` the `question` variable is provided automatically – since we're using a Django model (`Question`), Django is able to determine an appropriate name for the context variable. However, for ListView, the automatically generated context variable is

`question_list`. To override this we provide the `context_object_name` attribute, specifying that we want to use `latest_question_list` instead. As an alternative approach, you could change your templates to match the new default context variables – but it's a lot easier to just tell Django to use the variable you want.

Run the server, and use your new polling app based on generic views.

For full details on generic views, see the generic views documentation.

When you're comfortable with forms and generic views, read part 5 of this tutorial to learn about testing our polls app.

2.7 Writing your first Django app, part 5

This tutorial begins where Tutorial 4 left off. We've built a Web-poll application, and we'll now create some automated tests for it.

2.7.1 Introducing automated testing

What are automated tests?

Tests are simple routines that check the operation of your code.

Testing operates at different levels. Some tests might apply to a tiny detail (*does a particular model method return values as expected?*) while others examine the overall operation of the software (*does a sequence of user inputs on the site produce the desired result?*). That's no different from the kind of testing you did earlier in Tutorial 2, using the `shell` to examine the behavior of a method, or running the application and entering data to check how it behaves.

What's different in *automated* tests is that the testing work is done for you by the system. You create a set of tests once, and then as you make changes to your app, you can check that your code still works as you originally intended, without having to perform time consuming manual testing.

Why you need to create tests

So why create tests, and why now?

You may feel that you have quite enough on your plate just learning Python/Django, and having yet another thing to learn and do may seem overwhelming and perhaps unnecessary. After all, our polls application is working quite happily now; going through the trouble of creating automated tests is not going to make it work any better. If creating the polls application is the last bit of Django programming you will ever do, then true, you don't need to know how to create automated tests. But, if that's not the case, now is an excellent time to learn.

Tests will save you time

Up to a certain point, 'checking that it seems to work' will be a satisfactory test. In a more sophisticated application, you might have dozens of complex interactions between components.

A change in any of those components could have unexpected consequences on the application's behavior. Checking that it still 'seems to work' could mean running through your code's functionality with twenty different variations of your test data just to make sure you haven't broken something - not a good use of your time.

That's especially true when automated tests could do this for you in seconds. If something's gone wrong, tests will also assist in identifying the code that's causing the unexpected behavior.

Sometimes it may seem a chore to tear yourself away from your productive, creative programming work to face the unglamorous and unexciting business of writing tests, particularly when you know your code is working properly.

However, the task of writing tests is a lot more fulfilling than spending hours testing your application manually or trying to identify the cause of a newly-introduced problem.

Tests don't just identify problems, they prevent them

It's a mistake to think of tests merely as a negative aspect of development.

Without tests, the purpose or intended behavior of an application might be rather opaque. Even when it's your own code, you will sometimes find yourself poking around in it trying to find out what exactly it's doing.

Tests change that; they light up your code from the inside, and when something goes wrong, they focus light on the part that has gone wrong - *even if you hadn't even realized it had gone wrong*.

Tests make your code more attractive

You might have created a brilliant piece of software, but you will find that many other developers will simply refuse to look at it because it lacks tests; without tests, they won't trust it. Jacob Kaplan-Moss, one of Django's original developers, says "Code without tests is broken by design."

That other developers want to see tests in your software before they take it seriously is yet another reason for you to start writing tests.

Tests help teams work together

The previous points are written from the point of view of a single developer maintaining an application. Complex applications will be maintained by teams. Tests guarantee that colleagues don't inadvertently break your code (and that you don't break theirs without knowing). If you want to make a living as a Django programmer, you must be good at writing tests!

2.7.2 Basic testing strategies

There are many ways to approach writing tests.

Some programmers follow a discipline called "test-driven development"; they actually write their tests before they write their code. This might seem counter-intuitive, but in fact it's similar to what most people will often do anyway: they describe a problem, then create some code to solve it. Test-driven development simply formalizes the problem in a Python test case.

More often, a newcomer to testing will create some code and later decide that it should have some tests. Perhaps it would have been better to write some tests earlier, but it's never too late to get started.

Sometimes it's difficult to figure out where to get started with writing tests. If you have written several thousand lines of Python, choosing something to test might not be easy. In such a case, it's fruitful to write your first test the next time you make a change, either when you add a new feature or fix a bug.

So let's do that right away.

2.7.3 Writing our first test

We identify a bug

Fortunately, there's a little bug in the `polls` application for us to fix right away: the `Question.was_published_recently()` method returns `True` if the `Question` was published within the

last day (which is correct) but also if the `Question`'s `pub_date` field is in the future (which certainly isn't).

To check if the bug really exists, using the Admin create a question whose date lies in the future and check the method using the *`shell`*:

```
>>> import datetime
>>> from django.utils import timezone
>>> from polls.models import Question
>>> # create a Question instance with pub_date 30 days in the future
>>> future_question = Question(pub_date=timezone.now() + datetime.timedelta(days=30))
>>> # was it published recently?
>>> future_question.was_published_recently()
True
```

Since things in the future are not 'recent', this is clearly wrong.

Create a test to expose the bug

What we've just done in the *`shell`* to test for the problem is exactly what we can do in an automated test, so let's turn that into an automated test.

A conventional place for an application's tests is in the application's `tests.py` file; the testing system will automatically find tests in any file whose name begins with `test`.

Put the following in the `tests.py` file in the `polls` application:

polls/tests.py

```
import datetime

from django.utils import timezone
from django.test import TestCase

from .models import Question


class QuestionMethodTests(TestCase):

    def test_was_published_recently_with_future_question(self):
        """
        was_published_recently() should return False for questions whose
        pub_date is in the future.
        """
        time = timezone.now() + datetime.timedelta(days=30)
        future_question = Question(pub_date=time)
        self.assertIs(future_question.was_published_recently(), False)
```

What we have done here is created a *`django.test.TestCase`* subclass with a method that creates a `Question` instance with a `pub_date` in the future. We then check the output of `was_published_recently()` - which *ought* to be False.

Running tests

In the terminal, we can run our test:

```
$ python manage.py test polls
```

and you'll see something like:

```
Creating test database for alias 'default'...
F
======================================================================
FAIL: test_was_published_recently_with_future_question (polls.tests.QuestionMethodTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/path/to/mysite/polls/tests.py", line 16, in test_was_published_recently_with_future_question
    self.assertIs(future_question.was_published_recently(), False)
AssertionError: True is not False

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

What happened is this:

- `python manage.py test polls` looked for tests in the `polls` application
- it found a subclass of the *`django.test.TestCase`* class
- it created a special database for the purpose of testing
- it looked for test methods - ones whose names begin with `test`
- in `test_was_published_recently_with_future_question` it created a `Question` instance whose `pub_date` field is 30 days in the future
- ... and using the `assertIs()` method, it discovered that its `was_published_recently()` returns `True`, though we wanted it to return `False`

The test informs us which test failed and even the line on which the failure occurred.

Fixing the bug

We already know what the problem is: `Question.was_published_recently()` should return `False` if its `pub_date` is in the future. Amend the method in `models.py`, so that it will only return `True` if the date is also in the past:

polls/models.py

```
def was_published_recently(self):
    now = timezone.now()
    return now - datetime.timedelta(days=1) <= self.pub_date <= now
```

and run the test again:

```
Creating test database for alias 'default'...
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
Destroying test database for alias 'default'...
```

After identifying a bug, we wrote a test that exposes it and corrected the bug in the code so our test passes.

Many other things might go wrong with our application in the future, but we can be sure that we won't inadvertently reintroduce this bug, because simply running the test will warn us immediately. We can consider this little portion of the application pinned down safely forever.

More comprehensive tests

While we're here, we can further pin down the `was_published_recently()` method; in fact, it would be positively embarrassing if in fixing one bug we had introduced another.

Add two more test methods to the same class, to test the behavior of the method more comprehensively:

polls/tests.py

```
def test_was_published_recently_with_old_question(self):
    """
    was_published_recently() should return False for questions whose
    pub_date is older than 1 day.
    """
    time = timezone.now() - datetime.timedelta(days=30)
    old_question = Question(pub_date=time)
    self.assertIs(old_question.was_published_recently(), False)

def test_was_published_recently_with_recent_question(self):
    """
    was_published_recently() should return True for questions whose
    pub_date is within the last day.
    """
    time = timezone.now() - datetime.timedelta(hours=1)
    recent_question = Question(pub_date=time)
    self.assertIs(recent_question.was_published_recently(), True)
```

And now we have three tests that confirm that `Question.was_published_recently()` returns sensible values for past, recent, and future questions.

Again, `polls` is a simple application, but however complex it grows in the future and whatever other code it interacts with, we now have some guarantee that the method we have written tests for will behave in expected ways.

2.7.4 Test a view

The polls application is fairly undiscriminating: it will publish any question, including ones whose `pub_date` field lies in the future. We should improve this. Setting a `pub_date` in the future should mean that the Question is published at that moment, but invisible until then.

A test for a view

When we fixed the bug above, we wrote the test first and then the code to fix it. In fact that was a simple example of test-driven development, but it doesn't really matter in which order we do the work.

In our first test, we focused closely on the internal behavior of the code. For this test, we want to check its behavior as it would be experienced by a user through a web browser.

Before we try to fix anything, let's have a look at the tools at our disposal.

The Django test client

Django provides a test *Client* to simulate a user interacting with the code at the view level. We can use it in `tests.py` or even in the *shell*.

We will start again with the *shell*, where we need to do a couple of things that won't be necessary in `tests.py`. The first is to set up the test environment in the *shell*:

```
>>> from django.test.utils import setup_test_environment
>>> setup_test_environment()
```

setup_test_environment() installs a template renderer which will allow us to examine some additional attributes on responses such as `response.context` that otherwise wouldn't be available. Note that this method *does not* setup a test database, so the following will be run against the existing database and the output may differ slightly depending on what questions you already created. You might get unexpected results if your `TIME_ZONE` in `settings.py` isn't correct. If you don't remember setting it earlier, check it before continuing.

Next we need to import the test client class (later in `tests.py` we will use the *django.test.TestCase* class, which comes with its own client, so this won't be required):

```
>>> from django.test import Client
>>> # create an instance of the client for our use
>>> client = Client()
```

With that ready, we can ask the client to do some work for us:

```
>>> # get a response from '/'
>>> response = client.get('/')
>>> # we should expect a 404 from that address
>>> response.status_code
404
>>> # on the other hand we should expect to find something at '/polls/'
>>> # we'll use 'reverse()' rather than a hardcoded URL
>>> from django.urls import reverse
>>> response = client.get(reverse('polls:index'))
>>> response.status_code
200
>>> response.content
b'\n    <ul>\n    \n        <li><a href="/polls/1/">What&#39;s up?</a></li>\n    \n    </ul>\n\n'
>>> # If the following doesn't work, you probably omitted the call to
>>> # setup_test_environment() described above
>>> response.context['latest_question_list']
<QuerySet [<Question: What's up?>]>
```

Improving our view

The list of polls shows polls that aren't published yet (i.e. those that have a `pub_date` in the future). Let's fix that.

In Tutorial 4 we introduced a class-based view, based on *ListView*:

polls/views.py

```
class IndexView(generic.ListView):
    template_name = 'polls/index.html'
    context_object_name = 'latest_question_list'

    def get_queryset(self):
        """Return the last five published questions."""
        return Question.objects.order_by('-pub_date')[:5]
```

We need to amend the `get_queryset()` method and change it so that it also checks the date by comparing it with `timezone.now()`. First we need to add an import:

polls/views.py

```
from django.utils import timezone
```

and then we must amend the `get_queryset` method like so:

polls/views.py

```
def get_queryset(self):
    """
    Return the last five published questions (not including those set to be
    published in the future).
    """
    return Question.objects.filter(
        pub_date__lte=timezone.now()
    ).order_by('-pub_date')[:5]
```

`Question.objects.filter(pub_date__lte=timezone.now())` returns a queryset containing `Questions` whose `pub_date` is less than or equal to - that is, earlier than or equal to - `timezone.now`.

Testing our new view

Now you can satisfy yourself that this behaves as expected by firing up the runserver, loading the site in your browser, creating `Questions` with dates in the past and future, and checking that only those that have been published are listed. You don't want to have to do that *every single time you make any change that might affect this* - so let's also create a test, based on our *`shell`* session above.

Add the following to `polls/tests.py`:

polls/tests.py

```
from django.urls import reverse
```

and we'll create a shortcut function to create questions as well as a new test class:

polls/tests.py

```
def create_question(question_text, days):
    """
    Creates a question with the given `question_text` and published the
    given number of `days` offset to now (negative for questions published
    in the past, positive for questions that have yet to be published).
    """
    time = timezone.now() + datetime.timedelta(days=days)
    return Question.objects.create(question_text=question_text, pub_date=time)


class QuestionViewTests(TestCase):
    def test_index_view_with_no_questions(self):
        """
        If no questions exist, an appropriate message should be displayed.
        """
        response = self.client.get(reverse('polls:index'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "No polls are available.")
        self.assertQuerysetEqual(response.context['latest_question_list'], [])

    def test_index_view_with_a_past_question(self):
        """
        Questions with a pub_date in the past should be displayed on the
        index page.
        """
        create_question(question_text="Past question.", days=-30)
```

```
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question.>']
        )

    def test_index_view_with_a_future_question(self):
        """
        Questions with a pub_date in the future should not be displayed on
        the index page.
        """
        create_question(question_text="Future question.", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertContains(response, "No polls are available.")
        self.assertQuerysetEqual(response.context['latest_question_list'], [])

    def test_index_view_with_future_question_and_past_question(self):
        """
        Even if both past and future questions exist, only past questions
        should be displayed.
        """
        create_question(question_text="Past question.", days=-30)
        create_question(question_text="Future question.", days=30)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question.>']
        )

    def test_index_view_with_two_past_questions(self):
        """
        The questions index page may display multiple questions.
        """
        create_question(question_text="Past question 1.", days=-30)
        create_question(question_text="Past question 2.", days=-5)
        response = self.client.get(reverse('polls:index'))
        self.assertQuerysetEqual(
            response.context['latest_question_list'],
            ['<Question: Past question 2.>', '<Question: Past question 1.>']
        )
```

Let's look at some of these more closely.

First is a question shortcut function, `create_question`, to take some repetition out of the process of creating questions.

`test_index_view_with_no_questions` doesn't create any questions, but checks the message: "No polls are available." and verifies the `latest_question_list` is empty. Note that the *`django.test.TestCase`* class provides some additional assertion methods. In these examples, we use *`assertContains()`* and *`assertQuerysetEqual()`*.

In `test_index_view_with_a_past_question`, we create a question and verify that it appears in the list.

In `test_index_view_with_a_future_question`, we create a question with a `pub_date` in the future. The database is reset for each test method, so the first question is no longer there, and so again the index shouldn't have any questions in it.

And so on. In effect, we are using the tests to tell a story of admin input and user experience on the site, and checking that at every state and for every new change in the state of the system, the expected results are published.

Testing the DetailView

What we have works well; however, even though future questions don't appear in the *index*, users can still reach them if they know or guess the right URL. So we need to add a similar constraint to DetailView:

polls/views.py

```
class DetailView(generic.DetailView):
    ...
    def get_queryset(self):
        """
        Excludes any questions that aren't published yet.
        """
        return Question.objects.filter(pub_date__lte=timezone.now())
```

And of course, we will add some tests, to check that a Question whose pub_date is in the past can be displayed, and that one with a pub_date in the future is not:

polls/tests.py

```
class QuestionIndexDetailTests(TestCase):
    def test_detail_view_with_a_future_question(self):
        """
        The detail view of a question with a pub_date in the future should
        return a 404 not found.
        """
        future_question = create_question(question_text='Future question.', days=5)
        url = reverse('polls:detail', args=(future_question.id,))
        response = self.client.get(url)
        self.assertEqual(response.status_code, 404)

    def test_detail_view_with_a_past_question(self):
        """
        The detail view of a question with a pub_date in the past should
        display the question's text.
        """
        past_question = create_question(question_text='Past Question.', days=-5)
        url = reverse('polls:detail', args=(past_question.id,))
        response = self.client.get(url)
        self.assertContains(response, past_question.question_text)
```

Ideas for more tests

We ought to add a similar get_queryset method to ResultsView and create a new test class for that view. It'll be very similar to what we have just created; in fact there will be a lot of repetition.

We could also improve our application in other ways, adding tests along the way. For example, it's silly that Questions can be published on the site that have no Choices. So, our views could check for this, and exclude such Questions. Our tests would create a Question without Choices and then test that it's not published, as well as create a similar Question *with* Choices, and test that it *is* published.

Perhaps logged-in admin users should be allowed to see unpublished Questions, but not ordinary visitors. Again: whatever needs to be added to the software to accomplish this should be accompanied by a test, whether you write the test first and then make the code pass the test, or work out the logic in your code first and then write a test to prove it.

At a certain point you are bound to look at your tests and wonder whether your code is suffering from test bloat, which brings us to:

2.7.5 When testing, more is better

It might seem that our tests are growing out of control. At this rate there will soon be more code in our tests than in our application, and the repetition is unaesthetic, compared to the elegant conciseness of the rest of our code.

It doesn't matter. Let them grow. For the most part, you can write a test once and then forget about it. It will continue performing its useful function as you continue to develop your program.

Sometimes tests will need to be updated. Suppose that we amend our views so that only `Questions` with `Choices` are published. In that case, many of our existing tests will fail - *telling us exactly which tests need to be amended to bring them up to date*, so to that extent tests help look after themselves.

At worst, as you continue developing, you might find that you have some tests that are now redundant. Even that's not a problem; in testing redundancy is a *good* thing.

As long as your tests are sensibly arranged, they won't become unmanageable. Good rules-of-thumb include having:

- a separate `TestClass` for each model or view
- a separate test method for each set of conditions you want to test
- test method names that describe their function

2.7.6 Further testing

This tutorial only introduces some of the basics of testing. There's a great deal more you can do, and a number of very useful tools at your disposal to achieve some very clever things.

For example, while our tests here have covered some of the internal logic of a model and the way our views publish information, you can use an "in-browser" framework such as Selenium to test the way your HTML actually renders in a browser. These tools allow you to check not just the behavior of your Django code, but also, for example, of your JavaScript. It's quite something to see the tests launch a browser, and start interacting with your site, as if a human being were driving it! Django includes `LiveServerTestCase` to facilitate integration with tools like Selenium.

If you have a complex application, you may want to run tests automatically with every commit for the purposes of continuous integration, so that quality control is itself - at least partially - automated.

A good way to spot untested parts of your application is to check code coverage. This also helps identify fragile or even dead code. If you can't test a piece of code, it usually means that code should be refactored or removed. Coverage will help to identify dead code. See *Integration with coverage.py* for details.

Testing in Django has comprehensive information about testing.

2.7.7 What's next?

For full details on testing, see Testing in Django.

When you're comfortable with testing Django views, read part 6 of this tutorial to learn about static files management.

2.8 Writing your first Django app, part 6

This tutorial begins where Tutorial 5 left off. We've built a tested Web-poll application, and we'll now add a stylesheet and an image.

Aside from the HTML generated by the server, web applications generally need to serve additional files — such as images, JavaScript, or CSS — necessary to render the complete web page. In Django, we refer to these files as "static files".

For small projects, this isn't a big deal, because you can just keep the static files somewhere your web server can find it. However, in bigger projects – especially those comprised of multiple apps – dealing with the multiple sets of static files provided by each application starts to get tricky.

That's what `django.contrib.staticfiles` is for: it collects static files from each of your applications (and any other places you specify) into a single location that can easily be served in production.

2.8.1 Customize your *app's* look and feel

First, create a directory called `static` in your `polls` directory. Django will look for static files there, similarly to how Django finds templates inside `polls/templates/`.

Django's *STATICFILES_FINDERS* setting contains a list of finders that know how to discover static files from various sources. One of the defaults is `AppDirectoriesFinder` which looks for a "static" subdirectory in each of the *INSTALLED_APPS*, like the one in `polls` we just created. The admin site uses the same directory structure for its static files.

Within the `static` directory you have just created, create another directory called `polls` and within that create a file called `style.css`. In other words, your stylesheet should be at `polls/static/polls/style.css`. Because of how the `AppDirectoriesFinder` staticfile finder works, you can refer to this static file in Django simply as `polls/style.css`, similar to how you reference the path for templates.

Static file namespacing

Just like templates, we *might* be able to get away with putting our static files directly in `polls/static` (rather than creating another `polls` subdirectory), but it would actually be a bad idea. Django will choose the first static file it finds whose name matches, and if you had a static file with the same name in a *different* application, Django would be unable to distinguish between them. We need to be able to point Django at the right one, and the easiest way to ensure this is by *namespacing* them. That is, by putting those static files inside *another* directory named for the application itself.

Put the following code in that stylesheet (`polls/static/polls/style.css`):

`polls/static/polls/style.css`

```
li a {
    color: green;
}
```

Next, add the following at the top of `polls/templates/polls/index.html`:

`polls/templates/polls/index.html`

```
{% load static %}

<link rel="stylesheet" type="text/css" href="{% static 'polls/style.css' %}" />
```

The `{% static %}` template tag generates the absolute URL of static files.

That's all you need to do for development. Reload `http://localhost:8000/polls/` and you should see that the question links are green (Django style!) which means that your stylesheet was properly loaded.

2.8.2 Adding a background-image

Next, we'll create a subdirectory for images. Create an `images` subdirectory in the `polls/static/polls/` directory. Inside this directory, put an image called `background.gif`. In other words, put your image in

`polls/static/polls/images/background.gif`.

Then, add to your stylesheet (`polls/static/polls/style.css`):

`polls/static/polls/style.css`

```
body {
    background: white url("images/background.gif") no-repeat right bottom;
}
```

Reload `http://localhost:8000/polls/` and you should see the background loaded in the bottom right of the screen.

: Of course the `{% static %}` template tag is not available for use in static files like your stylesheet which aren't generated by Django. You should always use **relative paths** to link your static files between each other, because then you can change *`STATIC_URL`* (used by the *`static`* template tag to generate its URLs) without having to modify a bunch of paths in your static files as well.

These are the **basics**. For more details on settings and other bits included with the framework see the static files howto and the staticfiles reference. Deploying static files discusses how to use static files on a real server.

When you're comfortable with the static files, read part 7 of this tutorial to learn how to customize Django's automatically-generated admin site.

2.9 Writing your first Django app, part 7

This tutorial begins where Tutorial 6 left off. We're continuing the Web-poll application and will focus on customizing Django's automatically-generated admin site that we first explored in Tutorial 2.

2.9.1 Customize the admin form

By registering the `Question` model with `admin.site.register(Question)`, Django was able to construct a default form representation. Often, you'll want to customize how the admin form looks and works. You'll do this by telling Django the options you want when you register the object.

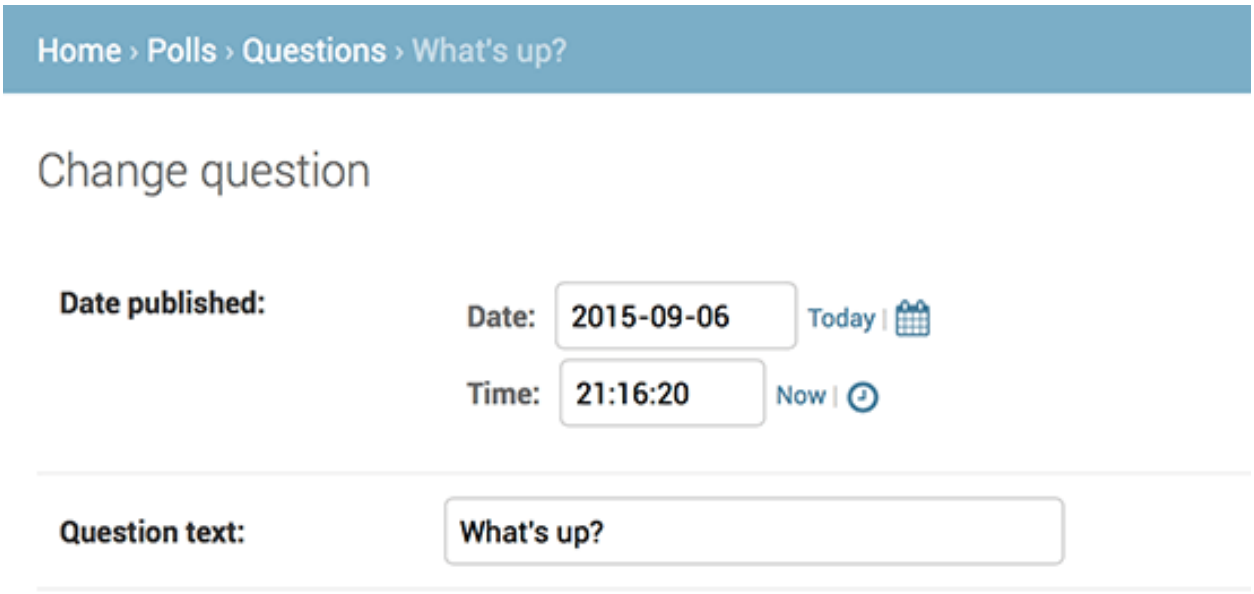
Let's see how this works by reordering the fields on the edit form. Replace the `admin.site.register(Question)` line with:

`polls/admin.py`

```
from django.contrib import admin

from .models import Question


class QuestionAdmin(admin.ModelAdmin):
    fields = ['pub_date', 'question_text']

admin.site.register(Question, QuestionAdmin)
```

You'll follow this pattern – create a model admin class, then pass it as the second argument to `admin.site.register()` – any time you need to change the admin options for a model.

This particular change above makes the "Publication date" come before the "Question" field:



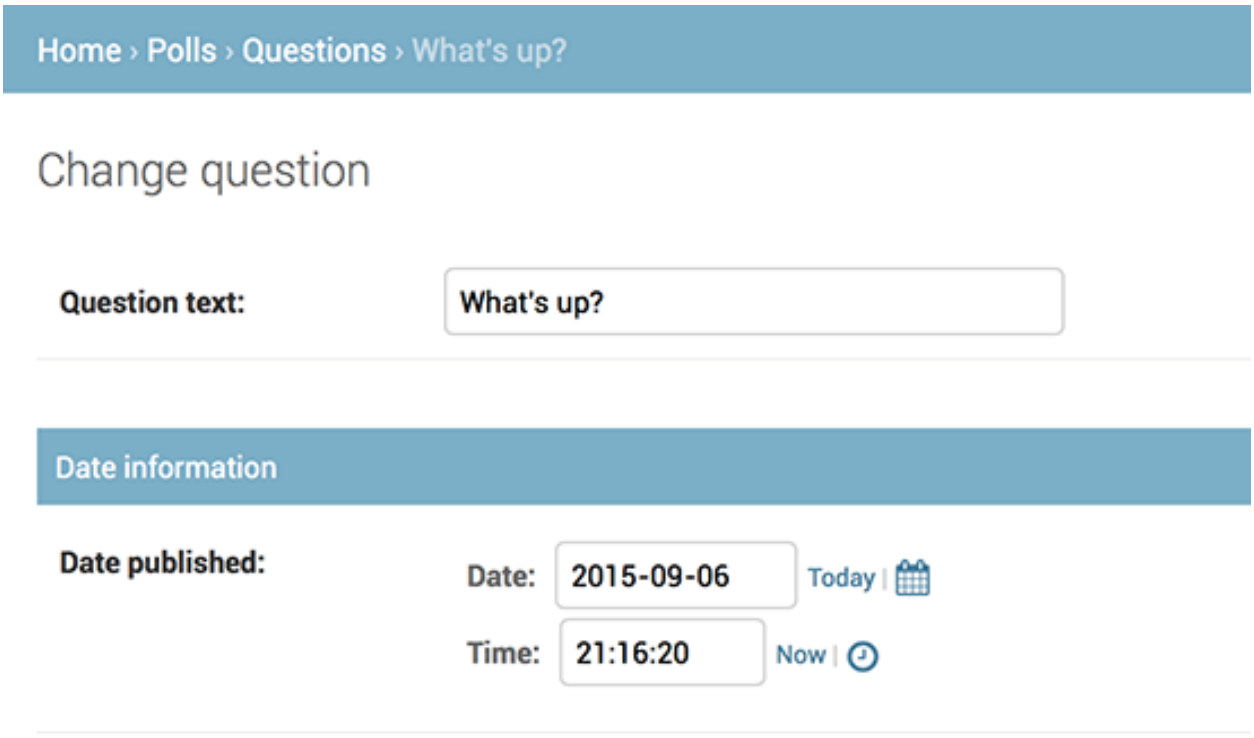
This isn't impressive with only two fields, but for admin forms with dozens of fields, choosing an intuitive order is an important usability detail.

And speaking of forms with dozens of fields, you might want to split the form up into fieldsets:

polls/admin.py

```
from django.contrib import admin

from .models import Question


class QuestionAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['question_text']}),
        ('Date information', {'fields': ['pub_date']}),
    ]

admin.site.register(Question, QuestionAdmin)
```

The first element of each tuple in *fieldsets* is the title of the fieldset. Here's what our form looks like now:



2.9.2 Adding related objects

OK, we have our Question admin page, but a `Question` has multiple `Choices`, and the admin page doesn't display choices.

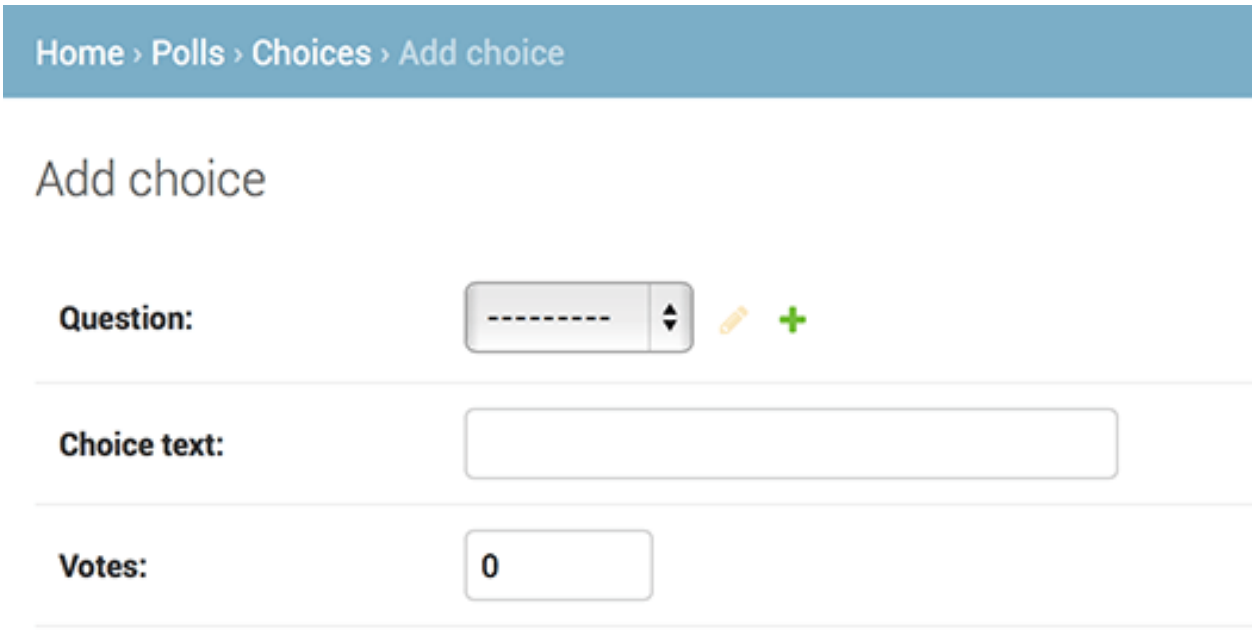
Yet.

There are two ways to solve this problem. The first is to register `Choice` with the admin just as we did with `Question`. That's easy:

polls/admin.py

```
from django.contrib import admin

from .models import Choice, Question
# ...
admin.site.register(Choice)
```

Now "Choices" is an available option in the Django admin. The "Add choice" form looks like this:



In that form, the "Question" field is a select box containing every question in the database. Django knows that a *ForeignKey* should be represented in the admin as a `<select>` box. In our case, only one question exists at this point.

Also note the "Add Another" link next to "Question." Every object with a `ForeignKey` relationship to another gets this for free. When you click "Add Another", you'll get a popup window with the "Add question" form. If you add a question in that window and click "Save", Django will save the question to the database and dynamically add it as the selected choice on the "Add choice" form you're looking at.

But, really, this is an inefficient way of adding `Choice` objects to the system. It'd be better if you could add a bunch of Choices directly when you create the `Question` object. Let's make that happen.

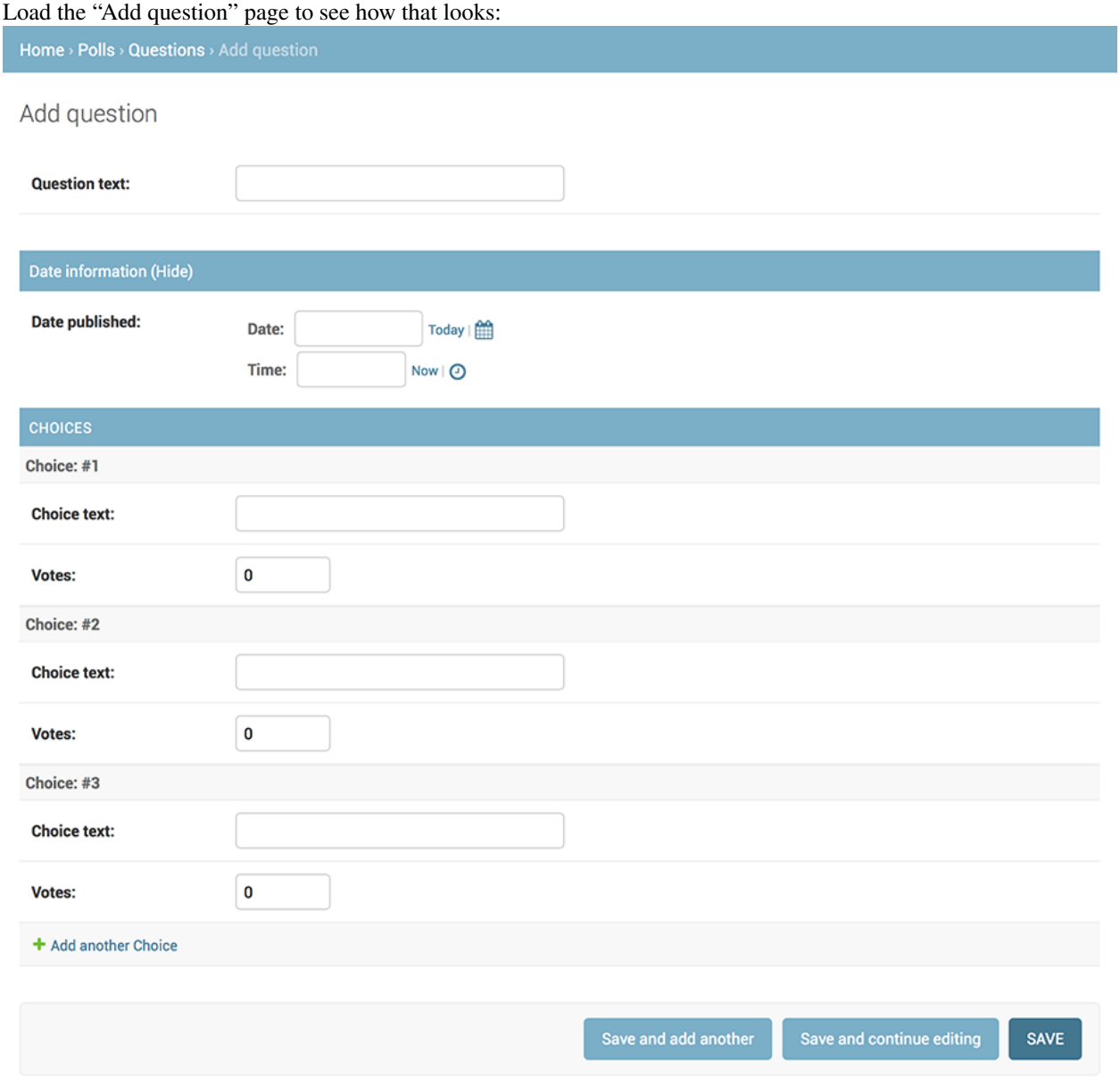
Remove the `register()` call for the `Choice` model. Then, edit the `Question` registration code to read:

polls/admin.py

```
from django.contrib import admin

from .models import Choice, Question


class ChoiceInline(admin.StackedInline):
    model = Choice
    extra = 3


class QuestionAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['question_text']}),
        ('Date information', {'fields': ['pub_date'], 'classes': ['collapse']}),
    ]
    inlines = [ChoiceInline]

admin.site.register(Question, QuestionAdmin)
```

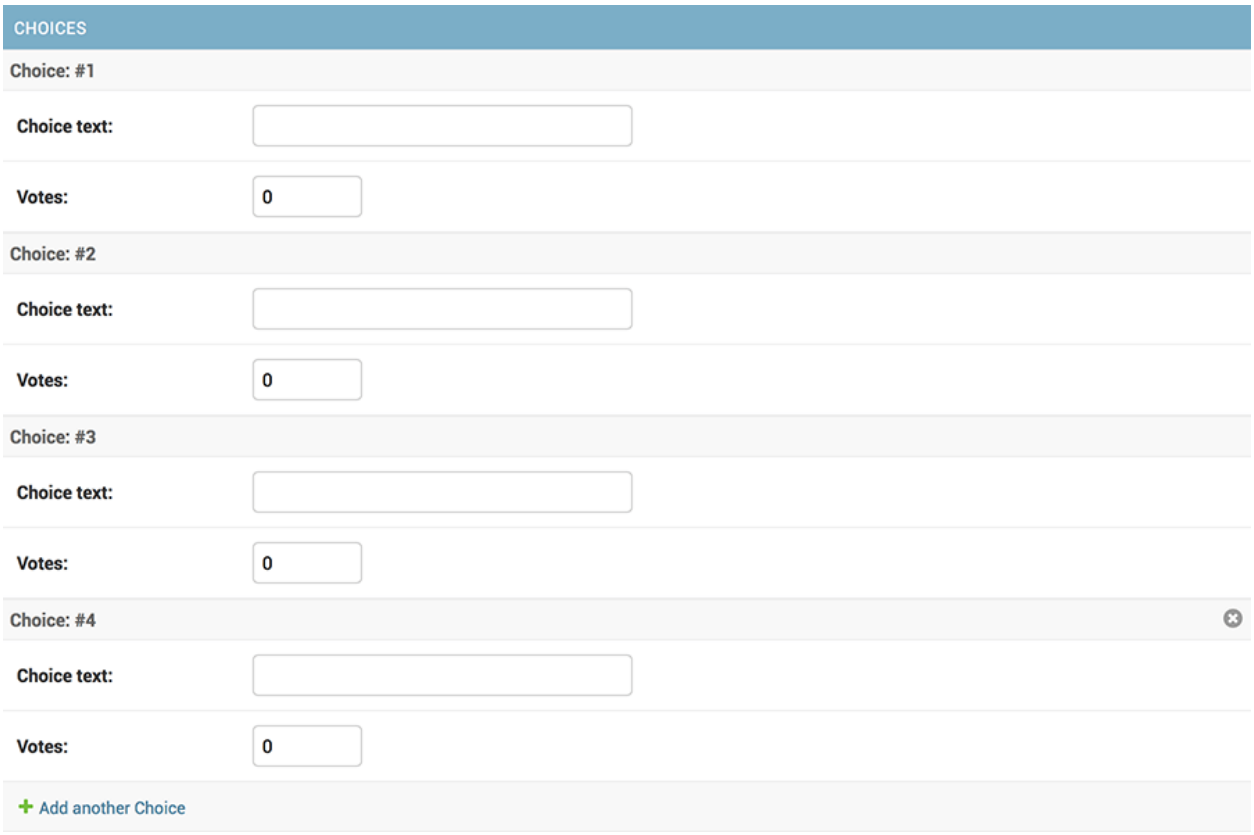
This tells Django: "`Choice` objects are edited on the `Question` admin page. By default, provide enough fields for 3 choices."

Load the "Add question" page to see how that looks:



It works like this: There are three slots for related Choices – as specified by `extra` – and each time you come back to the "Change" page for an already-created object, you get another three extra slots.

At the end of the three current slots you will find an "Add another Choice" link. If you click on it, a new slot will be added. If you want to remove the added slot, you can click on the X to the top right of the added slot. Note that you can't remove the original three slots. This image shows an added slot:

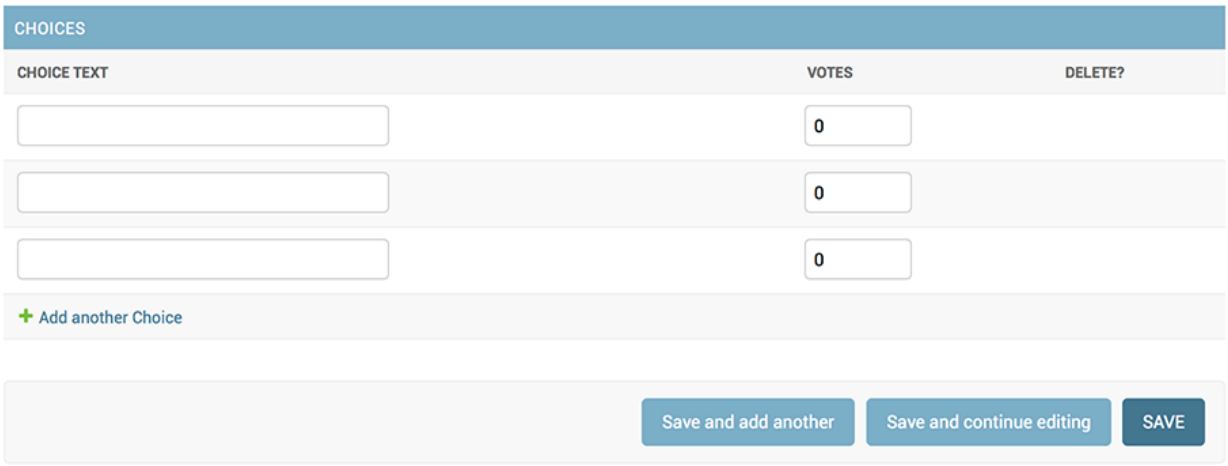


One small problem, though. It takes a lot of screen space to display all the fields for entering related `Choice` objects. For that reason, Django offers a tabular way of displaying inline related objects; you just need to change the `ChoiceInline` declaration to read:

polls/admin.py

```
class ChoiceInline(admin.TabularInline):
    #...
```

With that `TabularInline` (instead of `StackedInline`), the related objects are displayed in a more compact, table-based format:

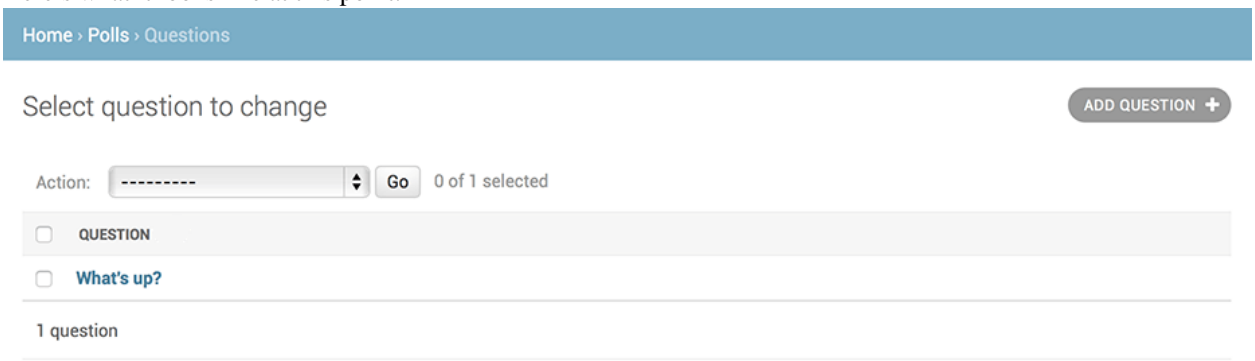


Note that there is an extra "Delete?" column that allows removing rows added using the "Add Another Choice" button and rows that have already been saved.

2.9.3 Customize the admin change list

Now that the Question admin page is looking good, let's make some tweaks to the "change list" page – the one that displays all the questions in the system.

Here's what it looks like at this point:



By default, Django displays the `str()` of each object. But sometimes it'd be more helpful if we could display individual fields. To do that, use the *`list_display`* admin option, which is a tuple of field names to display, as columns, on the change list page for the object:

polls/admin.py

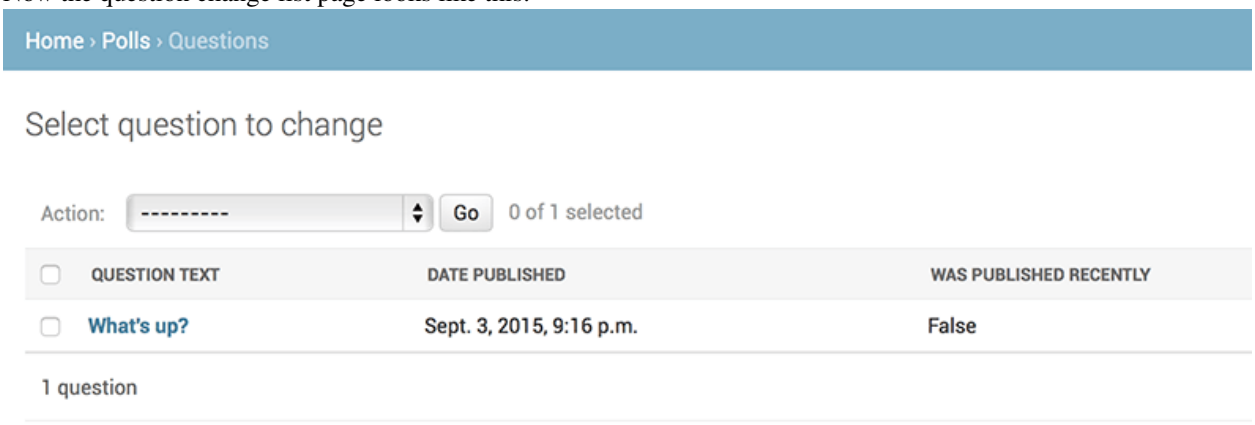
```
class QuestionAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question_text', 'pub_date')
```

Just for good measure, let's also include the `was_published_recently()` method from Tutorial 2:

polls/admin.py

```
class QuestionAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question_text', 'pub_date', 'was_published_recently')
```

Now the question change list page looks like this:



You can click on the column headers to sort by those values – except in the case of the `was_published_recently` header, because sorting by the output of an arbitrary method is not supported. Also note that the column header for `was_published_recently` is, by default, the name of the method (with underscores replaced with spaces), and that each line contains the string representation of the output.

You can improve that by giving that method (in `polls/models.py`) a few attributes, as follows:

polls/models.py

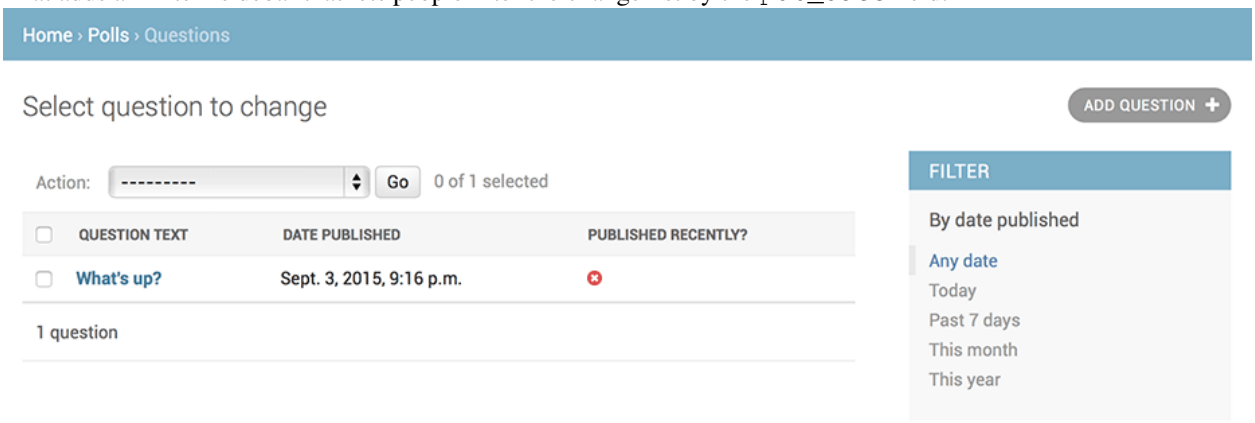
```
class Question(models.Model):
    # ...
    def was_published_recently(self):
        now = timezone.now()
        return now - datetime.timedelta(days=1) <= self.pub_date <= now
    was_published_recently.admin_order_field = 'pub_date'
    was_published_recently.boolean = True
    was_published_recently.short_description = 'Published recently?'
```

For more information on these method properties, see *list_display*.

Edit your `polls/admin.py` file again and add an improvement to the `Question` change list page: filters using the *list_filter*. Add the following line to `QuestionAdmin`:

```
list_filter = ['pub_date']
```

That adds a "Filter" sidebar that lets people filter the change list by the `pub_date` field:



The type of filter displayed depends on the type of field you're filtering on. Because `pub_date` is a *DateTimeField*, Django knows to give appropriate filter options: "Any date", "Today", "Past 7 days", "This month", "This year".

This is shaping up well. Let's add some search capability:

```
search_fields = ['question_text']
```

That adds a search box at the top of the change list. When somebody enters search terms, Django will search the `question_text` field. You can use as many fields as you'd like – although because it uses a `LIKE` query behind the scenes, limiting the number of search fields to a reasonable number will make it easier for your database to do the search.

Now's also a good time to note that change lists give you free pagination. The default is to display 100 items per page. *Change list pagination*, *search boxes*, *filters*, *date-hierarchies*, and *column-header-ordering* all work together like you think they should.

2.9.4 Customize the admin look and feel

Clearly, having "Django administration" at the top of each admin page is ridiculous. It's just placeholder text.

That's easy to change, though, using Django's template system. The Django admin is powered by Django itself, and its interfaces use Django's own template system.

Customizing your *project's* templates

Create a `templates` directory in your project directory (the one that contains `manage.py`). Templates can live anywhere on your filesystem that Django can access. (Django runs as whatever user your server runs.) However, keeping your templates within the project is a good convention to follow.

Open your settings file (`mysite/settings.py`, remember) and add a *DIRS* option in the *TEMPLATES* setting:

mysite/settings.py

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

DIRS is a list of filesystem directories to check when loading Django templates; it's a search path.

Organizing templates

Just like the static files, we *could* have all our templates together, in one big templates directory, and it would work perfectly well. However, templates that belong to a particular application should be placed in that application's template directory (e.g. `polls/templates`) rather than the project's (`templates`). We'll discuss in more detail in the reusable apps tutorial *why* we do this.

Now create a directory called `admin` inside `templates`, and copy the template `admin/base_site.html` from within the default Django admin template directory in the source code of Django itself (`django/contrib/admin/templates`) into that directory.

Where are the Django source files?

If you have difficulty finding where the Django source files are located on your system, run the following command:

```
$ python -c "import django; print(django.__path__)"
```

Then, just edit the file and replace `{{ site_header|default:_('Django administration') }}` (including the curly braces) with your own site's name as you see fit. You should end up with a section of code like:

```
{% block branding %}
<h1 id="site-name"><a href="{% url 'admin:index' %}">Polls Administration</a></h1>
{% endblock %}
```

We use this approach to teach you how to override templates. In an actual project, you would probably use the *django.contrib.admin.AdminSite.site_header* attribute to more easily make this particular customization.

This template file contains lots of text like `{% block branding %}` and `{{ title }}`. The `{%` and `{{` tags are part of Django's template language. When Django renders `admin/base_site.html`, this template language will be evaluated to produce the final HTML page, just like we saw in Tutorial 3.

Note that any of Django's default admin templates can be overridden. To override a template, just do the same thing you did with `base_site.html` – copy it from the default directory into your custom directory, and make changes.

Customizing your *application's* templates

Astute readers will ask: But if *DIRS* was empty by default, how was Django finding the default admin templates? The answer is that, since *APP_DIRS* is set to `True`, Django automatically looks for a `templates/` subdirectory within each application package, for use as a fallback (don't forget that `django.contrib.admin` is an application).

Our poll application is not very complex and doesn't need custom admin templates. But if it grew more sophisticated and required modification of Django's standard admin templates for some of its functionality, it would be more sensible to modify the *application's* templates, rather than those in the *project*. That way, you could include the polls application in any new project and be assured that it would find the custom templates it needed.

See the *template loading documentation* for more information about how Django finds its templates.

2.9.5 Customize the admin index page

On a similar note, you might want to customize the look and feel of the Django admin index page.

By default, it displays all the apps in *INSTALLED_APPS* that have been registered with the admin application, in alphabetical order. You may want to make significant changes to the layout. After all, the index is probably the most important page of the admin, and it should be easy to use.

The template to customize is `admin/index.html`. (Do the same as with `admin/base_site.html` in the previous section – copy it from the default directory to your custom template directory). Edit the file, and you'll see it uses a template variable called `app_list`. That variable contains every installed Django app. Instead of using that, you can hard-code links to object-specific admin pages in whatever way you think is best.

2.9.6 What's next?

The beginner tutorial ends here. In the meantime, you might want to check out some pointers on where to go from here.

If you are familiar with Python packaging and interested in learning how to turn polls into a "reusable app", check out Advanced tutorial: How to write reusable apps.

2.10 Advanced tutorial: How to write reusable apps

This advanced tutorial begins where Tutorial 7 left off. We'll be turning our Web-poll into a standalone Python package you can reuse in new projects and share with other people.

If you haven't recently completed Tutorials 1–7, we encourage you to review these so that your example project matches the one described below.

2.10.1 Reusability matters

It's a lot of work to design, build, test and maintain a web application. Many Python and Django projects share common problems. Wouldn't it be great if we could save some of this repeated work?

Reusability is the way of life in Python. The Python Package Index (PyPI) has a vast range of packages you can use in your own Python programs. Check out Django Packages for existing reusable apps you could incorporate in your project. Django itself is also just a Python package. This means that you can take existing Python packages or Django apps and compose them into your own web project. You only need to write the parts that make your project unique.

Let's say you were starting a new project that needed a polls app like the one we've been working on. How do you make this app reusable? Luckily, you're well on the way already. In Tutorial 3, we saw how we could decouple polls from the project-level URLconf using an `include`. In this tutorial, we'll take further steps to make the app easy to use in new projects and ready to publish for others to install and use.

Package? App?

A Python package provides a way of grouping related Python code for easy reuse. A package contains one or more files of Python code (also known as "modules").

A package can be imported with `import foo.bar` or `from foo import bar`. For a directory (like `polls`) to form a package, it must contain a special file `__init__.py`, even if this file is empty.

A Django *application* is just a Python package that is specifically intended for use in a Django project. An application may use common Django conventions, such as having `models`, `tests`, `urls`, and `views` submodules.

Later on we use the term *packaging* to describe the process of making a Python package easy for others to install. It can be a little confusing, we know.

2.10.2 Your project and your reusable app

After the previous tutorials, our project should look like this:

```
mysite/
    manage.py
    mysite/
        __init__.py
        settings.py
        urls.py
        wsgi.py
    polls/
        __init__.py
        admin.py
        migrations/
            __init__.py
            0001_initial.py
        models.py
        static/
            polls/
                images/
                    background.gif
                style.css
        templates/
            polls/
                detail.html
                index.html
                results.html
```

```
        tests.py
        urls.py
        views.py
    templates/
        admin/
            base_site.html
```

You created `mysite/templates` in Tutorial 7, and `polls/templates` in Tutorial 3. Now perhaps it is clearer why we chose to have separate template directories for the project and application: everything that is part of the polls application is in `polls`. It makes the application self-contained and easier to drop into a new project.

The `polls` directory could now be copied into a new Django project and immediately reused. It's not quite ready to be published though. For that, we need to package the app to make it easy for others to install.

2.10.3 Installing some prerequisites

The current state of Python packaging is a bit muddled with various tools. For this tutorial, we're going to use setuptools to build our package. It's the recommended packaging tool (merged with the `distribute` fork). We'll also be using pip to install and uninstall it. You should install these two packages now. If you need help, you can refer to *how to install Django with pip*. You can install `setuptools` the same way.

2.10.4 Packaging your app

Python *packaging* refers to preparing your app in a specific format that can be easily installed and used. Django itself is packaged very much like this. For a small app like polls, this process isn't too difficult.

1. First, create a parent directory for `polls`, outside of your Django project. Call this directory `django-polls`.

   Choosing a name for your app

   When choosing a name for your package, check resources like PyPI to avoid naming conflicts with existing packages. It's often useful to prepend `django-` to your module name when creating a package to distribute. This helps others looking for Django apps identify your app as Django specific.

   Application labels (that is, the final part of the dotted path to application packages) *must* be unique in *INSTALLED_APPS*. Avoid using the same label as any of the Django contrib packages, for example `auth`, `admin`, or `messages`.

2. Move the `polls` directory into the `django-polls` directory.

3. Create a file `django-polls/README.rst` with the following contents:

   `django-polls/README.rst`

```
=====
Polls
=====

Polls is a simple Django app to conduct Web-based polls. For each
question, visitors can choose between a fixed number of answers.

Detailed documentation is in the "docs" directory.

Quick start
-----------
```

```
1. Add "polls" to your INSTALLED_APPS setting like this::

    INSTALLED_APPS = [
        ...
        'polls',
    ]

2. Include the polls URLconf in your project urls.py like this::

    url(r'^polls/', include('polls.urls')),

3. Run `python manage.py migrate` to create the polls models.

4. Start the development server and visit http://127.0.0.1:8000/admin/
   to create a poll (you'll need the Admin app enabled).

5. Visit http://127.0.0.1:8000/polls/ to participate in the poll.
```

4. Create a `django-polls/LICENSE` file. Choosing a license is beyond the scope of this tutorial, but suffice it to say that code released publicly without a license is *useless*. Django and many Django-compatible apps are distributed under the BSD license; however, you're free to pick your own license. Just be aware that your licensing choice will affect who is able to use your code.

5. Next we'll create a `setup.py` file which provides details about how to build and install the app. A full explanation of this file is beyond the scope of this tutorial, but the setuptools docs have a good explanation. Create a file `django-polls/setup.py` with the following contents:

   `django-polls/setup.py`

```
import os
from setuptools import find_packages, setup

with open(os.path.join(os.path.dirname(__file__), 'README.rst')) as readme:
    README = readme.read()

# allow setup.py to be run from any path
os.chdir(os.path.normpath(os.path.join(os.path.abspath(__file__), os.pardir)))

setup(
    name='django-polls',
    version='0.1',
    packages=find_packages(),
    include_package_data=True,
    license='BSD License',  # example license
    description='A simple Django app to conduct Web-based polls.',
    long_description=README,
    url='https://www.example.com/',
    author='Your Name',
    author_email='yourname@example.com',
    classifiers=[
        'Environment :: Web Environment',
        'Framework :: Django',
        'Framework :: Django :: X.Y',  # replace "X.Y" as appropriate
        'Intended Audience :: Developers',
        'License :: OSI Approved :: BSD License',  # example license
        'Operating System :: OS Independent',
        'Programming Language :: Python',
        # Replace these appropriately if you are stuck on Python 2.
        'Programming Language :: Python :: 3',
```

```
            'Programming Language :: Python :: 3.4',
            'Programming Language :: Python :: 3.5',
            'Topic :: Internet :: WWW/HTTP',
            'Topic :: Internet :: WWW/HTTP :: Dynamic Content',
        ],
    )
```

6. Only Python modules and packages are included in the package by default. To include additional files, we'll need to create a `MANIFEST.in` file. The setuptools docs referred to in the previous step discuss this file in more details. To include the templates, the `README.rst` and our `LICENSE` file, create a file `django-polls/MANIFEST.in` with the following contents:

   `django-polls/MANIFEST.in`

```
include LICENSE
include README.rst
recursive-include polls/static *
recursive-include polls/templates *
```

7. It's optional, but recommended, to include detailed documentation with your app. Create an empty directory `django-polls/docs` for future documentation. Add an additional line to `django-polls/MANIFEST.in`:

```
recursive-include docs *
```

   Note that the `docs` directory won't be included in your package unless you add some files to it. Many Django apps also provide their documentation online through sites like readthedocs.org.

8. Try building your package with `python setup.py sdist` (run from inside `django-polls`). This creates a directory called `dist` and builds your new package, `django-polls-0.1.tar.gz`.

For more information on packaging, see Python's Tutorial on Packaging and Distributing Projects.

2.10.5 Using your own package

Since we moved the `polls` directory out of the project, it's no longer working. We'll now fix this by installing our new `django-polls` package.

Installing as a user library

The following steps install `django-polls` as a user library. Per-user installs have a lot of advantages over installing the package system-wide, such as being usable on systems where you don't have administrator access as well as preventing the package from affecting system services and other users of the machine.

Note that per-user installations can still affect the behavior of system tools that run as that user, so `virtualenv` is a more robust solution (see below).

---

1. To install the package, use pip (you already *installed it*, right?):

```
pip install --user django-polls/dist/django-polls-0.1.tar.gz
```

2. With luck, your Django project should now work correctly again. Run the server again to confirm this.
3. To uninstall the package, use pip:

```
pip uninstall django-polls
```

2.10.6 Publishing your app

Now that we've packaged and tested `django-polls`, it's ready to share with the world! If this wasn't just an example, you could now:

- Email the package to a friend.
- Upload the package on your website.
- Post the package on a public repository, such as the Python Package Index (PyPI). packaging.python.org has a good tutorial for doing this.

2.10.7 Installing Python packages with virtualenv

Earlier, we installed the polls app as a user library. This has some disadvantages:

- Modifying the user libraries can affect other Python software on your system.
- You won't be able to run multiple versions of this package (or others with the same name).

Typically, these situations only arise once you're maintaining several Django projects. When they do, the best solution is to use virtualenv. This tool allows you to maintain multiple isolated Python environments, each with its own copy of the libraries and package namespace.

2.11 What to read next

So you've read all the introductory material and have decided you'd like to keep using Django. We've only just scratched the surface with this intro (in fact, if you've read every single word, you've read about 5% of the overall documentation).

So what's next?

Well, we've always been big fans of learning by doing. At this point you should know enough to start a project of your own and start fooling around. As you need to learn new tricks, come back to the documentation.

We've put a lot of effort into making Django's documentation useful, easy to read and as complete as possible. The rest of this document explains more about how the documentation works so that you can get the most out of it.

(Yes, this is documentation about documentation. Rest assured we have no plans to write a document about how to read the document about documentation.)

2.11.1 Finding documentation

Django's got a *lot* of documentation – almost 450,000 words and counting – so finding what you need can sometimes be tricky. A few good places to start are the search and the genindex.

Or you can just browse around!

2.11.2 How the documentation is organized

Django's main documentation is broken up into "chunks" designed to fill different needs:

- The introductory material is designed for people new to Django – or to Web development in general. It doesn't cover anything in depth, but instead gives a high-level overview of how developing in Django "feels".

- The topic guides, on the other hand, dive deep into individual parts of Django. There are complete guides to Django's model system, template engine, forms framework, and much more.

  This is probably where you'll want to spend most of your time; if you work your way through these guides you should come out knowing pretty much everything there is to know about Django.

- Web development is often broad, not deep – problems span many domains. We've written a set of how-to guides that answer common "How do I ...?" questions. Here you'll find information about generating PDFs with Django, writing custom template tags, and more.

  Answers to really common questions can also be found in the FAQ.

- The guides and how-to's don't cover every single class, function, and method available in Django – that would be overwhelming when you're trying to learn. Instead, details about individual classes, functions, methods, and modules are kept in the reference. This is where you'll turn to find the details of a particular function or whatever you need.

- If you are interested in deploying a project for public use, our docs have several guides for various deployment setups as well as a deployment checklist for some things you'll need to think about.

- Finally, there's some "specialized" documentation not usually relevant to most developers. This includes the release notes and internals documentation for those who want to add code to Django itself, and a few other things that simply don't fit elsewhere.

2.11.3 How documentation is updated

Just as the Django code base is developed and improved on a daily basis, our documentation is consistently improving. We improve documentation for several reasons:

- To make content fixes, such as grammar/typo corrections.
- To add information and/or examples to existing sections that need to be expanded.
- To document Django features that aren't yet documented. (The list of such features is shrinking but exists nonetheless.)
- To add documentation for new features as new features get added, or as Django APIs or behaviors change.

Django's documentation is kept in the same source control system as its code. It lives in the docs directory of our Git repository. Each document online is a separate text file in the repository.

2.11.4 Where to get it

You can read Django documentation in several ways. They are, in order of preference:

On the Web

The most recent version of the Django documentation lives at https://docs.djangoproject.com/en/dev/. These HTML pages are generated automatically from the text files in source control. That means they reflect the "latest and greatest" in Django – they include the very latest corrections and additions, and they discuss the latest Django features, which may only be available to users of the Django development version. (See "Differences between versions" below.)

We encourage you to help improve the docs by submitting changes, corrections and suggestions in the ticket system. The Django developers actively monitor the ticket system and use your feedback to improve the documentation for everybody.

Note, however, that tickets should explicitly relate to the documentation, rather than asking broad tech-support questions. If you need help with your particular Django setup, try the *django-users* mailing list or the #django IRC channel instead.

In plain text

For offline reading, or just for convenience, you can read the Django documentation in plain text.

If you're using an official release of Django, note that the zipped package (tarball) of the code includes a `docs/` directory, which contains all the documentation for that release.

If you're using the development version of Django (aka "trunk"), note that the `docs/` directory contains all of the documentation. You can update your Git checkout to get the latest changes.

One low-tech way of taking advantage of the text documentation is by using the Unix `grep` utility to search for a phrase in all of the documentation. For example, this will show you each mention of the phrase "max_length" in any Django document:

```
$ grep -r max_length /path/to/django/docs/
```

As HTML, locally

You can get a local copy of the HTML documentation following a few easy steps:

- Django's documentation uses a system called Sphinx to convert from plain text to HTML. You'll need to install Sphinx by either downloading and installing the package from the Sphinx website, or with `pip`:

```
$ pip install Sphinx
```

- Then, just use the included `Makefile` to turn the documentation into HTML:

```
$ cd path/to/django/docs
$ make html
```

  You'll need GNU Make installed for this.

  If you're on Windows you can alternatively use the included batch file:

```
cd path\to\django\docs
make.bat html
```

- The HTML documentation will be placed in `docs/_build/html`.

2.11.5 Differences between versions

As previously mentioned, the text documentation in our Git repository contains the "latest and greatest" changes and additions. These changes often include documentation of new features added in the Django development version – the Git ("trunk") version of Django. For that reason, it's worth pointing out our policy on keeping straight the documentation for various versions of the framework.

We follow this policy:

- The primary documentation on djangoproject.com is an HTML version of the latest docs in Git. These docs always correspond to the latest official Django release, plus whatever features we've added/changed in the framework *since* the latest release.
- As we add features to Django's development version, we try to update the documentation in the same Git commit transaction.

- To distinguish feature changes/additions in the docs, we use the phrase: "New in version X.Y", being X.Y the next release version (hence, the one being developed).
- Documentation fixes and improvements may be backported to the last release branch, at the discretion of the committer, however, once a version of Django is *no longer supported*, that version of the docs won't get any further updates.
- The main documentation Web page includes links to documentation for all previous versions. Be sure you are using the version of the docs corresponding to the version of Django you are using!

2.12 Writing your first patch for Django

2.12.1 Introduction

Interested in giving back to the community a little? Maybe you've found a bug in Django that you'd like to see fixed, or maybe there's a small feature you want added.

Contributing back to Django itself is the best way to see your own concerns addressed. This may seem daunting at first, but it's really pretty simple. We'll walk you through the entire process, so you can learn by example.

Who's this tutorial for?

:

If you are looking for a reference on how to submit patches, see the Submitting patches documentation.

For this tutorial, we expect that you have at least a basic understanding of how Django works. This means you should be comfortable going through the existing tutorials on writing your first Django app. In addition, you should have a good understanding of Python itself. But if you don't, Dive Into Python is a fantastic (and free) online book for beginning Python programmers.

Those of you who are unfamiliar with version control systems and Trac will find that this tutorial and its links include just enough information to get started. However, you'll probably want to read some more about these different tools if you plan on contributing to Django regularly.

For the most part though, this tutorial tries to explain as much as possible, so that it can be of use to the widest audience.

Where to get help:

If you're having trouble going through this tutorial, please post a message to *django-developers* or drop by #django-dev on irc.freenode.net to chat with other Django users who might be able to help.

What does this tutorial cover?

We'll be walking you through contributing a patch to Django for the first time. By the end of this tutorial, you should have a basic understanding of both the tools and the processes involved. Specifically, we'll be covering the following:

- Installing Git.
- How to download a development copy of Django.
- Running Django's test suite.
- Writing a test for your patch.

- Writing the code for your patch.
- Testing your patch.
- Generating a patch file for your changes.
- Where to look for more information.

Once you're done with the tutorial, you can look through the rest of Django's documentation on contributing. It contains lots of great information and is a must read for anyone who'd like to become a regular contributor to Django. If you've got questions, it's probably got the answers.

Python 3 required!

This tutorial assumes you are using Python 3. Get the latest version at Python's download page or with your operating system's package manager.

For Windows users

When installing Python on Windows, make sure you check the option "Add python.exe to Path", so that it is always available on the command line.

2.12.2 Code of Conduct

As a contributor, you can help us keep the Django community open and inclusive. Please read and follow our Code of Conduct.

2.12.3 Installing Git

For this tutorial, you'll need Git installed to download the current development version of Django and to generate patch files for the changes you make.

To check whether or not you have Git installed, enter `git` into the command line. If you get messages saying that this command could not be found, you'll have to download and install it, see Git's download page.

For Windows users

When installing Git on Windows, it is recommended that you pick the "Git Bash" option so that Git runs in its own shell. This tutorial assumes that's how you have installed it.

If you're not that familiar with Git, you can always find out more about its commands (once it's installed) by typing `git help` into the command line.

2.12.4 Getting a copy of Django's development version

The first step to contributing to Django is to get a copy of the source code. First, fork Django on GitHub. Then, from the command line, use the `cd` command to navigate to the directory where you'll want your local copy of Django to live.

Download the Django source code repository using the following command:

```
$ git clone git@github.com:YourGitHubName/django.git
```

Now that you have a local copy of Django, you can install it just like you would install any package using `pip`. The most convenient way to do so is by using a *virtual environment* (or virtualenv) which is a feature built into Python that allows you to keep a separate directory of installed packages for each of your projects so that they don't interfere with each other.

It's a good idea to keep all your virtualenvs in one place, for example in `.virtualenvs/` in your home directory. Create it if it doesn't exist yet:

```
$ mkdir ~/.virtualenvs
```

Now create a new virtualenv by running:

```
$ python3 -m venv ~/.virtualenvs/djangodev
```

The path is where the new environment will be saved on your computer.

For Windows users

Using the built-in `venv` module will not work if you are also using the Git Bash shell on Windows, since activation scripts are only created for the system shell (`.bat`) and PowerShell (`.ps1`). Use the `virtualenv` package instead:

```
$ pip install virtualenv
$ virtualenv ~/.virtualenvs/djangodev
```

For Ubuntu users

On some versions of Ubuntu the above command might fail. Use the `virtualenv` package instead, first making sure you have `pip3`:

```
$ sudo apt-get install python3-pip
$ # Prefix the next command with sudo if it gives a permission denied error
$ pip3 install virtualenv
$ virtualenv --python=`which python3` ~/.virtualenvs/djangodev
```

The final step in setting up your virtualenv is to activate it:

```
$ source ~/.virtualenvs/djangodev/bin/activate
```

If the `source` command is not available, you can try using a dot instead:

```
$ . ~/.virtualenvs/djangodev/bin/activate
```

For Windows users

To activate your virtualenv on Windows, run:

```
$ source ~/virtualenvs/djangodev/Scripts/activate
```

You have to activate the virtualenv whenever you open a new terminal window. virtualenvwrapper is a useful tool for making this more convenient.

Anything you install through `pip` from now on will be installed in your new virtualenv, isolated from other environments and system-wide packages. Also, the name of the currently activated virtualenv is displayed on the command line to help you keep track of which one you are using. Go ahead and install the previously cloned copy of Django:

```
$ pip install -e /path/to/your/local/clone/django/
```

The installed version of Django is now pointing at your local copy. You will immediately see any changes you make to it, which is of great help when writing your first patch.

2.12.5 Rolling back to a previous revision of Django

For this tutorial, we'll be using ticket #24788 as a case study, so we'll rewind Django's version history in git to before that ticket's patch was applied. This will allow us to go through all of the steps involved in writing that patch from scratch, including running Django's test suite.

Keep in mind that while we'll be using an older revision of Django's trunk for the purposes of the tutorial below, you should always use the current development revision of Django when working on your own patch for a ticket!

: The patch for this ticket was written by Paweł Marczewski, and it was applied to Django as commit 4df7e8483b2679fc1cba3410f08960bac6f51115. Consequently, we'll be using the revision of Django just prior to that, commit 4ccfc4439a7add24f8db4ef3960d02ef8ae09887.

Navigate into Django's root directory (that's the one that contains `django`, `docs`, `tests`, `AUTHORS`, etc.). You can then check out the older revision of Django that we'll be using in the tutorial below:

```
$ git checkout 4ccfc4439a7add24f8db4ef3960d02ef8ae09887
```

2.12.6 Running Django's test suite for the first time

When contributing to Django it's very important that your code changes don't introduce bugs into other areas of Django. One way to check that Django still works after you make your changes is by running Django's test suite. If all the tests still pass, then you can be reasonably sure that your changes haven't completely broken Django. If you've never run Django's test suite before, it's a good idea to run it once beforehand just to get familiar with what its output is supposed to look like.

Before running the test suite, install its dependencies by first `cd`-ing into the Django `tests/` directory and then running:

```
$ pip install -r requirements/py3.txt
```

If you encounter an error during the installation, your system might be missing a dependency for one or more of the Python packages. Consult the failing package's documentation or search the Web with the error message that you encounter.

Now we are ready to run the test suite. If you're using GNU/Linux, Mac OS X or some other flavor of Unix, run:

```
$ ./runtests.py
```

Now sit back and relax. Django's entire test suite has over 9,600 different tests, so it can take anywhere from 5 to 15 minutes to run, depending on the speed of your computer.

While Django's test suite is running, you'll see a stream of characters representing the status of each test as it's run. `E` indicates that an error was raised during a test, and `F` indicates that a test's assertions failed. Both of these are considered to be test failures. Meanwhile, `x` and `s` indicated expected failures and skipped tests, respectively. Dots indicate passing tests.

Skipped tests are typically due to missing external libraries required to run the test; see *Running all the tests* for a list of dependencies and be sure to install any for tests related to the changes you are making (we won't need any for this

tutorial). Some tests are specific to a particular database backend and will be skipped if not testing with that backend. SQLite is the database backend for the default settings. To run the tests using a different backend, see *Using another settings module*.

Once the tests complete, you should be greeted with a message informing you whether the test suite passed or failed. Since you haven't yet made any changes to Django's code, the entire test suite **should** pass. If you get failures or errors make sure you've followed all of the previous steps properly. See *Running the unit tests* for more information. If you're using Python 3.5+, there will be a couple failures related to deprecation warnings that you can ignore. These failures have since been fixed in Django.

Note that the latest Django trunk may not always be stable. When developing against trunk, you can check Django's continuous integration builds to determine if the failures are specific to your machine or if they are also present in Django's official builds. If you click to view a particular build, you can view the "Configuration Matrix" which shows failures broken down by Python version and database backend.

: For this tutorial and the ticket we're working on, testing against SQLite is sufficient, however, it's possible (and sometimes necessary) to *run the tests using a different database*.

2.12.7 Writing some tests for your ticket

In most cases, for a patch to be accepted into Django it has to include tests. For bug fix patches, this means writing a regression test to ensure that the bug is never reintroduced into Django later on. A regression test should be written in such a way that it will fail while the bug still exists and pass once the bug has been fixed. For patches containing new features, you'll need to include tests which ensure that the new features are working correctly. They too should fail when the new feature is not present, and then pass once it has been implemented.

A good way to do this is to write your new tests first, before making any changes to the code. This style of development is called test-driven development and can be applied to both entire projects and single patches. After writing your tests, you then run them to make sure that they do indeed fail (since you haven't fixed that bug or added that feature yet). If your new tests don't fail, you'll need to fix them so that they do. After all, a regression test that passes regardless of whether a bug is present is not very helpful at preventing that bug from reoccurring down the road.

Now for our hands-on example.

Writing some tests for ticket #24788

Ticket #24788 proposes a small feature addition: the ability to specify the class level attribute `prefix` on Form classes, so that:

```
[...] forms which ship with apps could effectively namespace themselves such
that N overlapping form fields could be POSTed at once and resolved to the
correct form.
```

In order to resolve this ticket, we'll add a `prefix` attribute to the `BaseForm` class. When creating instances of this class, passing a prefix to the `__init__()` method will still set that prefix on the created instance. But not passing a prefix (or passing `None`) will use the class-level prefix. Before we make those changes though, we're going to write a couple tests to verify that our modification functions correctly and continues to function correctly in the future.

Navigate to Django's `tests/forms_tests/tests/` folder and open the `test_forms.py` file. Add the following code on line 1674 right before the `test_forms_with_null_boolean` function:

```
def test_class_prefix(self):
    # Prefix can be also specified at the class level.
    class Person(Form):
        first_name = CharField()
```

```
        prefix = 'foo'

    p = Person()
    self.assertEqual(p.prefix, 'foo')

    p = Person(prefix='bar')
    self.assertEqual(p.prefix, 'bar')
```

This new test checks that setting a class level prefix works as expected, and that passing a `prefix` parameter when creating an instance still works too.

But this testing thing looks kinda hard...

If you've never had to deal with tests before, they can look a little hard to write at first glance. Fortunately, testing is a *very* big subject in computer programming, so there's lots of information out there:

- A good first look at writing tests for Django can be found in the documentation on Writing and running tests.
- Dive Into Python (a free online book for beginning Python developers) includes a great introduction to Unit Testing.
- After reading those, if you want something a little meatier to sink your teeth into, there's always the Python `unittest` documentation.

Running your new test

Remember that we haven't actually made any modifications to `BaseForm` yet, so our tests are going to fail. Let's run all the tests in the `forms_tests` folder to make sure that's really what happens. From the command line, `cd` into the Django `tests/` directory and run:

```
$ ./runtests.py forms_tests
```

If the tests ran correctly, you should see one failure corresponding to the test method we added. If all of the tests passed, then you'll want to make sure that you added the new test shown above to the appropriate folder and class.

2.12.8 Writing the code for your ticket

Next we'll be adding the functionality described in ticket #24788 to Django.

Writing the code for ticket #24788

Navigate to the `django/django/forms/` folder and open the `forms.py` file. Find the `BaseForm` class on line 72 and add the `prefix` class attribute right after the `field_order` attribute:

```
class BaseForm(object):
    # This is the main implementation of all the Form logic. Note that this
    # class is different than Form. See the comments by the Form class for
    # more information. Any improvements to the form API should be made to
    # *this* class, not to the Form class.
    field_order = None
    prefix = None
```

Verifying your test now passes

Once you're done modifying Django, we need to make sure that the tests we wrote earlier pass, so we can see whether the code we wrote above is working correctly. To run the tests in the `forms_tests` folder, `cd` into the Django `tests/` directory and run:

```
$ ./runtests.py forms_tests
```

Oops, good thing we wrote those tests! You should still see one failure with the following exception:

```
AssertionError: None != 'foo'
```

We forgot to add the conditional statement in the `__init__` method. Go ahead and change `self.prefix = prefix` that is now on line 87 of `django/forms/forms.py`, adding a conditional statement:

```
if prefix is not None:
    self.prefix = prefix
```

Re-run the tests and everything should pass. If it doesn't, make sure you correctly modified the `BaseForm` class as shown above and copied the new test correctly.

2.12.9 Running Django's test suite for the second time

Once you've verified that your patch and your test are working correctly, it's a good idea to run the entire Django test suite just to verify that your change hasn't introduced any bugs into other areas of Django. While successfully passing the entire test suite doesn't guarantee your code is bug free, it does help identify many bugs and regressions that might otherwise go unnoticed.

To run the entire Django test suite, `cd` into the Django `tests/` directory and run:

```
$ ./runtests.py
```

As long as you don't see any failures, you're good to go.

2.12.10 Writing Documentation

This is a new feature, so it should be documented. Add the following section on line 1068 (at the end of the file) of `django/docs/ref/forms/api.txt`:

```
The prefix can also be specified on the form class::

    >>> class PersonForm(forms.Form):
    ...     ...
    ...     prefix = 'person'

.. versionadded:: 1.9

    The ability to specify ``prefix`` on the form class was added.
```

Since this new feature will be in an upcoming release it is also added to the release notes for Django 1.9, on line 164 under the "Forms" section in the file `docs/releases/1.9.txt`:

```
* A form prefix can be specified inside a form class, not only when
  instantiating a form. See :ref:`form-prefix` for details.
```

For more information on writing documentation, including an explanation of what the `versionadded` bit is all about, see Writing documentation. That page also includes an explanation of how to build a copy of the documentation locally, so you can preview the HTML that will be generated.

2.12.11 Generating a patch for your changes

Now it's time to generate a patch file that can be uploaded to Trac or applied to another copy of Django. To get a look at the content of your patch, run the following command:

```
$ git diff
```

This will display the differences between your current copy of Django (with your changes) and the revision that you initially checked out earlier in the tutorial.

Once you're done looking at the patch, hit the `q` key to exit back to the command line. If the patch's content looked okay, you can run the following command to save the patch file to your current working directory:

```
$ git diff > 24788.diff
```

You should now have a file in the root Django directory called `24788.diff`. This patch file contains all your changes and should look this:

```
diff --git a/django/forms/forms.py b/django/forms/forms.py
index 509709f..d1370de 100644
--- a/django/forms/forms.py
+++ b/django/forms/forms.py
@@ -75,6 +75,7 @@ class BaseForm(object):
     # information. Any improvements to the form API should be made to *this*
     # class, not to the Form class.
     field_order = None
+    prefix = None

     def __init__(self, data=None, files=None, auto_id='id_%s', prefix=None,
                  initial=None, error_class=ErrorList, label_suffix=None,
@@ -83,7 +84,8 @@ class BaseForm(object):
         self.data = data or {}
         self.files = files or {}
         self.auto_id = auto_id
-        self.prefix = prefix
+        if prefix is not None:
+            self.prefix = prefix
         self.initial = initial or {}
         self.error_class = error_class
         # Translators: This is the default suffix added to form field labels
diff --git a/docs/ref/forms/api.txt b/docs/ref/forms/api.txt
index 3bc39cd..008170d 100644
--- a/docs/ref/forms/api.txt
+++ b/docs/ref/forms/api.txt
@@ -1065,3 +1065,13 @@ You can put several Django forms inside one ``<form>`` tag. To give each
     >>> print(father.as_ul())
     <li><label for="id_father-first_name">First name:</label> <input type="text" name="father-first_
     <li><label for="id_father-last_name">Last name:</label> <input type="text" name="father-last_nar
+
+The prefix can also be specified on the form class::
+
+    >>> class PersonForm(forms.Form):
+    ...     ...
+    ...     prefix = 'person'
+
+.. versionadded:: 1.9
+
+    The ability to specify ``prefix`` on the form class was added.
diff --git a/docs/releases/1.9.txt b/docs/releases/1.9.txt
```

```
index 5b58f79..f9bb9de 100644
--- a/docs/releases/1.9.txt
+++ b/docs/releases/1.9.txt
@@ -161,6 +161,9 @@ Forms
   :attr:`~django.forms.Form.field_order` attribute, the ``field_order``
   constructor argument , or the :meth:`~django.forms.Form.order_fields` method.

+* A form prefix can be specified inside a form class, not only when
+  instantiating a form. See :ref:`form-prefix` for details.
+
 Generic Views
 ^^^^^^^^^^^^^

diff --git a/tests/forms_tests/tests/test_forms.py b/tests/forms_tests/tests/test_forms.py
index 690f205..e07fae2 100644
--- a/tests/forms_tests/tests/test_forms.py
+++ b/tests/forms_tests/tests/test_forms.py
@@ -1671,6 +1671,18 @@ class FormsTestCase(SimpleTestCase):
         self.assertEqual(p.cleaned_data['last_name'], 'Lennon')
         self.assertEqual(p.cleaned_data['birthday'], datetime.date(1940, 10, 9))

+    def test_class_prefix(self):
+        # Prefix can be also specified at the class level.
+        class Person(Form):
+            first_name = CharField()
+            prefix = 'foo'
+
+        p = Person()
+        self.assertEqual(p.prefix, 'foo')
+
+        p = Person(prefix='bar')
+        self.assertEqual(p.prefix, 'bar')
+
     def test_forms_with_null_boolean(self):
         # NullBooleanField is a bit of a special case because its presentation (widget)
         # is different than its data. This is handled transparently, though.
```

2.12.12 So what do I do next?

Congratulations, you've generated your very first Django patch! Now that you've got that under your belt, you can put those skills to good use by helping to improve Django's codebase. Generating patches and attaching them to Trac tickets is useful, however, since we are using git - adopting a more git oriented workflow is recommended.

Since we never committed our changes locally, perform the following to get your git branch back to a good starting point:

```
$ git reset --hard HEAD
$ git checkout master
```

More information for new contributors

Before you get too into writing patches for Django, there's a little more information on contributing that you should probably take a look at:

- You should make sure to read Django's documentation on claiming tickets and submitting patches. It covers Trac etiquette, how to claim tickets for yourself, expected coding style for patches, and many other important

details.

- First time contributors should also read Django's documentation for first time contributors. It has lots of good advice for those of us who are new to helping out with Django.
- After those, if you're still hungry for more information about contributing, you can always browse through the rest of Django's documentation on contributing. It contains a ton of useful information and should be your first source for answering any questions you might have.

Finding your first real ticket

Once you've looked through some of that information, you'll be ready to go out and find a ticket of your own to write a patch for. Pay special attention to tickets with the "easy pickings" criterion. These tickets are often much simpler in nature and are great for first time contributors. Once you're familiar with contributing to Django, you can move on to writing patches for more difficult and complicated tickets.

If you just want to get started already (and nobody would blame you!), try taking a look at the list of easy tickets that need patches and the easy tickets that have patches which need improvement. If you're familiar with writing tests, you can also look at the list of easy tickets that need tests. Just remember to follow the guidelines about claiming tickets that were mentioned in the link to Django's documentation on claiming tickets and submitting patches.

What's next?

After a ticket has a patch, it needs to be reviewed by a second set of eyes. After uploading a patch or submitting a pull request, be sure to update the ticket metadata by setting the flags on the ticket to say "has patch", "doesn't need tests", etc, so others can find it for review. Contributing doesn't necessarily always mean writing a patch from scratch. Reviewing existing patches is also a very helpful contribution. See Triaging tickets for details.

:

If you're new to Python, you might want to start by getting an idea of what the language is like. Django is 100% Python, so if you've got minimal comfort with Python you'll probably get a lot more out of Django.

If you're new to programming entirely, you might want to start with this list of Python resources for non-programmers

If you already know a few other languages and want to get up to speed with Python quickly, we recommend Dive Into Python. If that's not quite your style, there are many other books about Python.

CHAPTER 3

Using Django

Introductions to all the key parts of Django you'll need to know:

3.1 How to install Django

This document will get you up and running with Django.

3.1.1 Install Python

Being a Python Web framework, Django requires Python. See *What Python version can I use with Django?* for details.

Get the latest version of Python at https://www.python.org/download/ or with your operating system's package manager.

Django on Jython

If you use Jython (a Python implementation for the Java platform), you'll need to follow a few additional steps. See Running Django on Jython for details.

Python on Windows

If you are just starting with Django and using Windows, you may find How to install Django on Windows useful.

3.1.2 Install Apache and `mod_wsgi`

If you just want to experiment with Django, skip ahead to the next section; Django includes a lightweight web server you can use for testing, so you won't need to set up Apache until you're ready to deploy Django in production.

If you want to use Django on a production site, use Apache with mod_wsgi. mod_wsgi can operate in one of two modes: an embedded mode and a daemon mode. In embedded mode, mod_wsgi is similar to mod_perl – it embeds Python within Apache and loads Python code into memory when the server starts. Code stays in memory throughout the life of an Apache process, which leads to significant performance gains over other server arrangements. In daemon mode, mod_wsgi spawns an independent daemon process that handles requests. The daemon process can run as a different user than the Web server, possibly leading to improved security, and the daemon process can be restarted without restarting the entire Apache Web server, possibly making refreshing your codebase more seamless. Consult

the mod_wsgi documentation to determine which mode is right for your setup. Make sure you have Apache installed, with the mod_wsgi module activated. Django will work with any version of Apache that supports mod_wsgi.

See How to use Django with mod_wsgi for information on how to configure mod_wsgi once you have it installed.

If you can't use mod_wsgi for some reason, fear not: Django supports many other deployment options. One is uWSGI; it works very well with nginx. Additionally, Django follows the WSGI spec (**PEP 3333**), which allows it to run on a variety of server platforms.

3.1.3 Get your database running

If you plan to use Django's database API functionality, you'll need to make sure a database server is running. Django supports many different database servers and is officially supported with PostgreSQL, MySQL, Oracle and SQLite.

If you are developing a simple project or something you don't plan to deploy in a production environment, SQLite is generally the simplest option as it doesn't require running a separate server. However, SQLite has many differences from other databases, so if you are working on something substantial, it's recommended to develop with the same database as you plan on using in production.

In addition to the officially supported databases, there are *backends provided by 3rd parties* that allow you to use other databases with Django.

In addition to a database backend, you'll need to make sure your Python database bindings are installed.

- If you're using PostgreSQL, you'll need the psycopg2 package. Refer to the *PostgreSQL notes* for further details.
- If you're using MySQL, you'll need a *DB API driver* like `mysqlclient`. See *notes for the MySQL backend* for details.
- If you're using SQLite you might want to read the *SQLite backend notes*.
- If you're using Oracle, you'll need a copy of cx_Oracle, but please read the *notes for the Oracle backend* for details regarding supported versions of both Oracle and `cx_Oracle`.
- If you're using an unofficial 3rd party backend, please consult the documentation provided for any additional requirements.

If you plan to use Django's `manage.py migrate` command to automatically create database tables for your models (after first installing Django and creating a project), you'll need to ensure that Django has permission to create and alter tables in the database you're using; if you plan to manually create the tables, you can simply grant Django `SELECT`, `INSERT`, `UPDATE` and `DELETE` permissions. After creating a database user with these permissions, you'll specify the details in your project's settings file, see *`DATABASES`* for details.

If you're using Django's testing framework to test database queries, Django will need permission to create a test database.

3.1.4 Remove any old versions of Django

If you are upgrading your installation of Django from a previous version, you will need to uninstall the old Django version before installing the new version.

If you installed Django using pip or `easy_install` previously, installing with pip or `easy_install` again will automatically take care of the old version, so you don't need to do it yourself.

If you previously installed Django using `python setup.py install`, uninstalling is as simple as deleting the `django` directory from your Python `site-packages`. To find the directory you need to remove, you can run the following at your shell prompt (not the interactive Python prompt):

```
$ python -c "import django; print(django.__path__)"
```

3.1.5 Install the Django code

Installation instructions are slightly different depending on whether you're installing a distribution-specific package, downloading the latest official release, or fetching the latest development version.

It's easy, no matter which way you choose.

Installing an official release with `pip`

This is the recommended way to install Django.

1. Install pip. The easiest is to use the standalone pip installer. If your distribution already has `pip` installed, you might need to update it if it's outdated. If it's outdated, you'll know because installation won't work.
2. Take a look at virtualenv and virtualenvwrapper. These tools provide isolated Python environments, which are more practical than installing packages systemwide. They also allow installing packages without administrator privileges. The contributing tutorial walks through how to create a virtualenv on Python 3.
3. After you've created and activated a virtual environment, enter the command `pip install Django` at the shell prompt.

Installing a distribution-specific package

Check the distribution specific notes to see if your platform/distribution provides official Django packages/installers. Distribution-provided packages will typically allow for automatic installation of dependencies and easy upgrade paths; however, these packages will rarely contain the latest release of Django.

Installing the development version

Tracking Django development

If you decide to use the latest development version of Django, you'll want to pay close attention to the development timeline, and you'll want to keep an eye on the *release notes for the upcoming release*. This will help you stay on top of any new features you might want to use, as well as any changes you'll need to make to your code when updating your copy of Django. (For stable releases, any necessary changes are documented in the release notes.)

If you'd like to be able to update your Django code occasionally with the latest bug fixes and improvements, follow these instructions:

1. Make sure that you have Git installed and that you can run its commands from a shell. (Enter `git help` at a shell prompt to test this.)
2. Check out Django's main development branch like so:

```
$ git clone git://github.com/django/django.git
```

   This will create a directory `django` in your current directory.

3. Make sure that the Python interpreter can load Django's code. The most convenient way to do this is to use virtualenv, virtualenvwrapper, and pip. The contributing tutorial walks through how to create a virtualenv on Python 3.

4. After setting up and activating the virtualenv, run the following command:

```
$ pip install -e django/
```

   This will make Django's code importable, and will also make the `django-admin` utility command available. In other words, you're all set!

When you want to update your copy of the Django source code, just run the command `git pull` from within the `django` directory. When you do this, Git will automatically download any changes.

3.2 Models and databases

A model is the single, definitive source of data about your data. It contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.

3.2.1 Models

A model is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.

The basics:

- Each model is a Python class that subclasses *`django.db.models.Model`*.
- Each attribute of the model represents a database field.
- With all of this, Django gives you an automatically-generated database-access API; see Making queries.

Quick example

This example model defines a `Person`, which has a `first_name` and `last_name`:

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

`first_name` and `last_name` are *fields* of the model. Each field is specified as a class attribute, and each attribute maps to a database column.

The above `Person` model would create a database table like this:

```
CREATE TABLE myapp_person (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

Some technical notes:

- The name of the table, `myapp_person`, is automatically derived from some model metadata but can be overridden. See *Table names* for more details.
- An `id` field is added automatically, but this behavior can be overridden. See *Automatic primary key fields*.
- The `CREATE TABLE` SQL in this example is formatted using PostgreSQL syntax, but it's worth noting Django uses SQL tailored to the database backend specified in your settings file.

Using models

Once you have defined your models, you need to tell Django you're going to *use* those models. Do this by editing your settings file and changing the *INSTALLED_APPS* setting to add the name of the module that contains your `models.py`.

For example, if the models for your application live in the module `myapp.models` (the package structure that is created for an application by the *manage.py startapp* script), *INSTALLED_APPS* should read, in part:

```
INSTALLED_APPS = [
    #...
    'myapp',
    #...
]
```

When you add new apps to *INSTALLED_APPS*, be sure to run *manage.py migrate*, optionally making migrations for them first with *manage.py makemigrations*.

Fields

The most important part of a model – and the only required part of a model – is the list of database fields it defines. Fields are specified by class attributes. Be careful not to choose field names that conflict with the models API like `clean`, `save`, or `delete`.

Example:

```
from django.db import models

class Musician(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    instrument = models.CharField(max_length=100)

class Album(models.Model):
    artist = models.ForeignKey(Musician, on_delete=models.CASCADE)
    name = models.CharField(max_length=100)
    release_date = models.DateField()
    num_stars = models.IntegerField()
```

Field types

Each field in your model should be an instance of the appropriate *Field* class. Django uses the field class types to determine a few things:

- The column type, which tells the database what kind of data to store (e.g. `INTEGER`, `VARCHAR`, `TEXT`).
- The default HTML widget to use when rendering a form field (e.g. `<input type="text">`, `<select>`).
- The minimal validation requirements, used in Django's admin and in automatically-generated forms.

Django ships with dozens of built-in field types; you can find the complete list in the *model field reference*. You can easily write your own fields if Django's built-in ones don't do the trick; see Writing custom model fields.

Field options

Each field takes a certain set of field-specific arguments (documented in the *model field reference*). For example, *CharField* (and its subclasses) require a *max_length* argument which specifies the size of the `VARCHAR` database

field used to store the data.

There's also a set of common arguments available to all field types. All are optional. They're fully explained in the *reference*, but here's a quick summary of the most often-used ones:

null If `True`, Django will store empty values as `NULL` in the database. Default is `False`.

blank If `True`, the field is allowed to be blank. Default is `False`.

Note that this is different than *null*. *null* is purely database-related, whereas *blank* is validation-related. If a field has *blank=True*, form validation will allow entry of an empty value. If a field has *blank=False*, the field will be required.

choices An iterable (e.g., a list or tuple) of 2-tuples to use as choices for this field. If this is given, the default form widget will be a select box instead of the standard text field and will limit choices to the choices given.

A choices list looks like this:

```
YEAR_IN_SCHOOL_CHOICES = (
    ('FR', 'Freshman'),
    ('SO', 'Sophomore'),
    ('JR', 'Junior'),
    ('SR', 'Senior'),
    ('GR', 'Graduate'),
)
```

The first element in each tuple is the value that will be stored in the database. The second element will be displayed by the default form widget or in a *ModelChoiceField*. Given a model instance, the display value for a choices field can be accessed using the `get_FOO_display()` method. For example:

```
from django.db import models

class Person(models.Model):
    SHIRT_SIZES = (
        ('S', 'Small'),
        ('M', 'Medium'),
        ('L', 'Large'),
    )
    name = models.CharField(max_length=60)
    shirt_size = models.CharField(max_length=1, choices=SHIRT_SIZES)
```

```
>>> p = Person(name="Fred Flintstone", shirt_size="L")
>>> p.save()
>>> p.shirt_size
'L'
>>> p.get_shirt_size_display()
'Large'
```

default The default value for the field. This can be a value or a callable object. If callable it will be called every time a new object is created.

help_text Extra "help" text to be displayed with the form widget. It's useful for documentation even if your field isn't used on a form.

primary_key If `True`, this field is the primary key for the model.

If you don't specify *primary_key=True* for any fields in your model, Django will automatically add an *IntegerField* to hold the primary key, so you don't need to set *primary_key=True* on any of your fields unless you want to override the default primary-key behavior. For more, see *Automatic primary key fields*.

The primary key field is read-only. If you change the value of the primary key on an existing object and then save it, a new object will be created alongside the old one. For example:

```
    from django.db import models

    class Fruit(models.Model):
        name = models.CharField(max_length=100, primary_key=True)
```

```
    >>> fruit = Fruit.objects.create(name='Apple')
    >>> fruit.name = 'Pear'
    >>> fruit.save()
    >>> Fruit.objects.values_list('name', flat=True)
    ['Apple', 'Pear']
```

unique If `True`, this field must be unique throughout the table.

Again, these are just short descriptions of the most common field options. Full details can be found in the *common model field option reference*.

Automatic primary key fields

By default, Django gives each model the following field:

```
id = models.AutoField(primary_key=True)
```

This is an auto-incrementing primary key.

If you'd like to specify a custom primary key, just specify *primary_key=True* on one of your fields. If Django sees you've explicitly set *Field.primary_key*, it won't add the automatic `id` column.

Each model requires exactly one field to have *primary_key=True* (either explicitly declared or automatically added).

Verbose field names

Each field type, except for *ForeignKey*, *ManyToManyField* and *OneToOneField*, takes an optional first positional argument – a verbose name. If the verbose name isn't given, Django will automatically create it using the field's attribute name, converting underscores to spaces.

In this example, the verbose name is `"person's first name"`:

```
first_name = models.CharField("person's first name", max_length=30)
```

In this example, the verbose name is `"first name"`:

```
first_name = models.CharField(max_length=30)
```

ForeignKey, *ManyToManyField* and *OneToOneField* require the first argument to be a model class, so use the *verbose_name* keyword argument:

```
poll = models.ForeignKey(
    Poll,
    on_delete=models.CASCADE,
    verbose_name="the related poll",
)
sites = models.ManyToManyField(Site, verbose_name="list of sites")
place = models.OneToOneField(
    Place,
    on_delete=models.CASCADE,
    verbose_name="related place",
)
```

The convention is not to capitalize the first letter of the *verbose_name*. Django will automatically capitalize the first letter where it needs to.

Relationships

Clearly, the power of relational databases lies in relating tables to each other. Django offers ways to define the three most common types of database relationships: many-to-one, many-to-many and one-to-one.

Many-to-one relationships To define a many-to-one relationship, use *django.db.models.ForeignKey*. You use it just like any other *Field* type: by including it as a class attribute of your model.

ForeignKey requires a positional argument: the class to which the model is related.

For example, if a `Car` model has a `Manufacturer` – that is, a `Manufacturer` makes multiple cars but each `Car` only has one `Manufacturer` – use the following definitions:

```
from django.db import models

class Manufacturer(models.Model):
    # ...
    pass

class Car(models.Model):
    manufacturer = models.ForeignKey(Manufacturer, on_delete=models.CASCADE)
    # ...
```

You can also create *recursive relationships* (an object with a many-to-one relationship to itself) and *relationships to models not yet defined*; see *the model field reference* for details.

It's suggested, but not required, that the name of a *ForeignKey* field (`manufacturer` in the example above) be the name of the model, lowercase. You can, of course, call the field whatever you want. For example:

```
class Car(models.Model):
    company_that_makes_it = models.ForeignKey(
        Manufacturer,
        on_delete=models.CASCADE,
    )
    # ...
```

:

ForeignKey fields accept a number of extra arguments which are explained in *the model field reference*. These options help define how the relationship should work; all are optional.

For details on accessing backwards-related objects, see the *Following relationships backward example*.

For sample code, see the Many-to-one relationship model example.

Many-to-many relationships To define a many-to-many relationship, use *ManyToManyField*. You use it just like any other *Field* type: by including it as a class attribute of your model.

ManyToManyField requires a positional argument: the class to which the model is related.

For example, if a `Pizza` has multiple `Topping` objects – that is, a `Topping` can be on multiple pizzas and each `Pizza` has multiple toppings – here's how you'd represent that:

```
from django.db import models

class Topping(models.Model):
    # ...
    pass

class Pizza(models.Model):
    # ...
    toppings = models.ManyToManyField(Topping)
```

As with *ForeignKey*, you can also create *recursive relationships* (an object with a many-to-many relationship to itself) and *relationships to models not yet defined*.

It's suggested, but not required, that the name of a *ManyToManyField* (`toppings` in the example above) be a plural describing the set of related model objects.

It doesn't matter which model has the *ManyToManyField*, but you should only put it in one of the models – not both.

Generally, *ManyToManyField* instances should go in the object that's going to be edited on a form. In the above example, `toppings` is in `Pizza` (rather than `Topping` having a `pizzas` *ManyToManyField*) because it's more natural to think about a pizza having toppings than a topping being on multiple pizzas. The way it's set up above, the `Pizza` form would let users select the toppings.

:

See the Many-to-many relationship model example for a full example.

ManyToManyField fields also accept a number of extra arguments which are explained in *the model field reference*. These options help define how the relationship should work; all are optional.

Extra fields on many-to-many relationships When you're only dealing with simple many-to-many relationships such as mixing and matching pizzas and toppings, a standard *ManyToManyField* is all you need. However, sometimes you may need to associate data with the relationship between two models.

For example, consider the case of an application tracking the musical groups which musicians belong to. There is a many-to-many relationship between a person and the groups of which they are a member, so you could use a *ManyToManyField* to represent this relationship. However, there is a lot of detail about the membership that you might want to collect, such as the date at which the person joined the group.

For these situations, Django allows you to specify the model that will be used to govern the many-to-many relationship. You can then put extra fields on the intermediate model. The intermediate model is associated with the *ManyToManyField* using the *through* argument to point to the model that will act as an intermediary. For our musician example, the code would look something like this:

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=128)

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, through='Membership')

    def __str__(self):              # __unicode__ on Python 2
        return self.name
```

```
class Membership(models.Model):
    person = models.ForeignKey(Person, on_delete=models.CASCADE)
    group = models.ForeignKey(Group, on_delete=models.CASCADE)
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)
```

When you set up the intermediary model, you explicitly specify foreign keys to the models that are involved in the many-to-many relationship. This explicit declaration defines how the two models are related.

There are a few restrictions on the intermediate model:

- Your intermediate model must contain one - and *only* one - foreign key to the source model (this would be `Group` in our example), or you must explicitly specify the foreign keys Django should use for the relationship using *`ManyToManyField.through_fields`*. If you have more than one foreign key and `through_fields` is not specified, a validation error will be raised. A similar restriction applies to the foreign key to the target model (this would be `Person` in our example).
- For a model which has a many-to-many relationship to itself through an intermediary model, two foreign keys to the same model are permitted, but they will be treated as the two (different) sides of the many-to-many relationship. If there are *more* than two foreign keys though, you must also specify `through_fields` as above, or a validation error will be raised.
- When defining a many-to-many relationship from a model to itself, using an intermediary model, you *must* use *`symmetrical=False`* (see *the model field reference*).

Now that you have set up your *`ManyToManyField`* to use your intermediary model (`Membership`, in this case), you're ready to start creating some many-to-many relationships. You do this by creating instances of the intermediate model:

```
>>> ringo = Person.objects.create(name="Ringo Starr")
>>> paul = Person.objects.create(name="Paul McCartney")
>>> beatles = Group.objects.create(name="The Beatles")
>>> m1 = Membership(person=ringo, group=beatles,
...     date_joined=date(1962, 8, 16),
...     invite_reason="Needed a new drummer.")
>>> m1.save()
>>> beatles.members.all()
<QuerySet [<Person: Ringo Starr>]>
>>> ringo.group_set.all()
<QuerySet [<Group: The Beatles>]>
>>> m2 = Membership.objects.create(person=paul, group=beatles,
...     date_joined=date(1960, 8, 1),
...     invite_reason="Wanted to form a band.")
>>> beatles.members.all()
<QuerySet [<Person: Ringo Starr>, <Person: Paul McCartney>]>
```

Unlike normal many-to-many fields, you *can't* use `add()`, `create()`, or `set()` to create relationships:

```
>>> # THIS WILL NOT WORK
>>> beatles.members.add(john)
>>> # NEITHER WILL THIS
>>> beatles.members.create(name="George Harrison")
>>> # AND NEITHER WILL THIS
>>> beatles.members.set([john, paul, ringo, george])
```

Why? You can't just create a relationship between a `Person` and a `Group` - you need to specify all the detail for the relationship required by the `Membership` model. The simple `add`, `create` and assignment calls don't provide a way to specify this extra detail. As a result, they are disabled for many-to-many relationships that use an intermediate model. The only way to create this type of relationship is to create instances of the intermediate model.

The *remove()* method is disabled for similar reasons. For example, if the custom through table defined by the intermediate model does not enforce uniqueness on the `(model1, model2)` pair, a `remove()` call would not provide enough information as to which intermediate model instance should be deleted:

```
>>> Membership.objects.create(person=ringo, group=beatles,
...     date_joined=date(1968, 9, 4),
...     invite_reason="You've been gone for a month and we miss you.")
>>> beatles.members.all()
<QuerySet [<Person: Ringo Starr>, <Person: Paul McCartney>, <Person: Ringo Starr>]>
>>> # THIS WILL NOT WORK BECAUSE IT CANNOT TELL WHICH MEMBERSHIP TO REMOVE
>>> beatles.members.remove(ringo)
```

However, the *clear()* method can be used to remove all many-to-many relationships for an instance:

```
>>> # Beatles have broken up
>>> beatles.members.clear()
>>> # Note that this deletes the intermediate model instances
>>> Membership.objects.all()
<QuerySet []>
```

Once you have established the many-to-many relationships by creating instances of your intermediate model, you can issue queries. Just as with normal many-to-many relationships, you can query using the attributes of the many-to-many-related model:

```
# Find all the groups with a member whose name starts with 'Paul'
>>> Group.objects.filter(members__name__startswith='Paul')
<QuerySet [<Group: The Beatles>]>
```

As you are using an intermediate model, you can also query on its attributes:

```
# Find all the members of the Beatles that joined after 1 Jan 1961
>>> Person.objects.filter(
...     group__name='The Beatles',
...     membership__date_joined__gt=date(1961,1,1))
<QuerySet [<Person: Ringo Starr]>
```

If you need to access a membership's information you may do so by directly querying the `Membership` model:

```
>>> ringos_membership = Membership.objects.get(group=beatles, person=ringo)
>>> ringos_membership.date_joined
datetime.date(1962, 8, 16)
>>> ringos_membership.invite_reason
'Needed a new drummer.'
```

Another way to access the same information is by querying the *many-to-many reverse relationship* from a `Person` object:

```
>>> ringos_membership = ringo.membership_set.get(group=beatles)
>>> ringos_membership.date_joined
datetime.date(1962, 8, 16)
>>> ringos_membership.invite_reason
'Needed a new drummer.'
```

One-to-one relationships To define a one-to-one relationship, use *OneToOneField*. You use it just like any other `Field` type: by including it as a class attribute of your model.

This is most useful on the primary key of an object when that object "extends" another object in some way.

OneToOneField requires a positional argument: the class to which the model is related.

For example, if you were building a database of "places", you would build pretty standard stuff such as address, phone number, etc. in the database. Then, if you wanted to build a database of restaurants on top of the places, instead of repeating yourself and replicating those fields in the `Restaurant` model, you could make `Restaurant` have a *`OneToOneField`* to `Place` (because a restaurant "is a" place; in fact, to handle this you'd typically use *inheritance*, which involves an implicit one-to-one relation).

As with *`ForeignKey`*, a *recursive relationship* can be defined and *references to as-yet undefined models* can be made.

:

See the One-to-one relationship model example for a full example.

`OneToOneField` fields also accept an optional *`parent_link`* argument.

`OneToOneField` classes used to automatically become the primary key on a model. This is no longer true (although you can manually pass in the *`primary_key`* argument if you like). Thus, it's now possible to have multiple fields of type *`OneToOneField`* on a single model.

Models across files

It's perfectly OK to relate a model to one from another app. To do this, import the related model at the top of the file where your model is defined. Then, just refer to the other model class wherever needed. For example:

```
from django.db import models
from geography.models import ZipCode

class Restaurant(models.Model):
    # ...
    zip_code = models.ForeignKey(
        ZipCode,
        on_delete=models.SET_NULL,
        blank=True,
        null=True,
    )
```

Field name restrictions

Django places only two restrictions on model field names:

1. A field name cannot be a Python reserved word, because that would result in a Python syntax error. For example:

```
    class Example(models.Model):
        pass = models.IntegerField() # 'pass' is a reserved word!
```

2. A field name cannot contain more than one underscore in a row, due to the way Django's query lookup syntax works. For example:

```
    class Example(models.Model):
        foo__bar = models.IntegerField() # 'foo__bar' has two underscores!
```

These limitations can be worked around, though, because your field name doesn't necessarily have to match your database column name. See the *`db_column`* option.

SQL reserved words, such as `join`, `where` or `select`, *are* allowed as model field names, because Django escapes all database table names and column names in every underlying SQL query. It uses the quoting syntax of your particular database engine.

Custom field types

If one of the existing model fields cannot be used to fit your purposes, or if you wish to take advantage of some less common database column types, you can create your own field class. Full coverage of creating your own fields is provided in Writing custom model fields.

`Meta` options

Give your model metadata by using an inner `class Meta`, like so:

```
from django.db import models

class Ox(models.Model):
    horn_length = models.IntegerField()

    class Meta:
        ordering = ["horn_length"]
        verbose_name_plural = "oxen"
```

Model metadata is "anything that's not a field", such as ordering options (*`ordering`*), database table name (*`db_table`*), or human-readable singular and plural names (*`verbose_name`* and *`verbose_name_plural`*). None are required, and adding `class Meta` to a model is completely optional.

A complete list of all possible `Meta` options can be found in the model option reference.

Model attributes

`objects` The most important attribute of a model is the *`Manager`*. It's the interface through which database query operations are provided to Django models and is used to *retrieve the instances* from the database. If no custom `Manager` is defined, the default name is *`objects`*. Managers are only accessible via model classes, not the model instances.

Model methods

Define custom methods on a model to add custom "row-level" functionality to your objects. Whereas *`Manager`* methods are intended to do "table-wide" things, model methods should act on a particular model instance.

This is a valuable technique for keeping business logic in one place – the model.

For example, this model has a few custom methods:

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    birth_date = models.DateField()

    def baby_boomer_status(self):
        "Returns the person's baby-boomer status."
        import datetime
        if self.birth_date < datetime.date(1945, 8, 1):
            return "Pre-boomer"
        elif self.birth_date < datetime.date(1965, 1, 1):
            return "Baby boomer"
        else:
```

```
            return "Post-boomer"

    def _get_full_name(self):
        "Returns the person's full name."
        return '%s %s' % (self.first_name, self.last_name)
    full_name = property(_get_full_name)
```

The last method in this example is a *property*.

The model instance reference has a complete list of *methods automatically given to each model*. You can override most of these – see *overriding predefined model methods*, below – but there are a couple that you'll almost always want to define:

`__str__()` (Python 3) A Python "magic method" that returns a unicode "representation" of any object. This is what Python and Django will use whenever a model instance needs to be coerced and displayed as a plain string. Most notably, this happens when you display an object in an interactive console or in the admin.

You'll always want to define this method; the default isn't very helpful at all.

`__unicode__()` (Python 2) Python 2 equivalent of `__str__()`.

`get_absolute_url()` This tells Django how to calculate the URL for an object. Django uses this in its admin interface, and any time it needs to figure out a URL for an object.

Any object that has a URL that uniquely identifies it should define this method.

Overriding predefined model methods

There's another set of *model methods* that encapsulate a bunch of database behavior that you'll want to customize. In particular you'll often want to change the way `save()` and `delete()` work.

You're free to override these methods (and any other model method) to alter behavior.

A classic use-case for overriding the built-in methods is if you want something to happen whenever you save an object. For example (see `save()` for documentation of the parameters it accepts):

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, *args, **kwargs):
        do_something()
        super(Blog, self).save(*args, **kwargs) # Call the "real" save() method.
        do_something_else()
```

You can also prevent saving:

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, *args, **kwargs):
        if self.name == "Yoko Ono's blog":
            return # Yoko shall never have her own blog!
        else:
            super(Blog, self).save(*args, **kwargs) # Call the "real" save() method.
```

It's important to remember to call the superclass method – that's that `super(Blog, self).save(*args, **kwargs)` business – to ensure that the object still gets saved into the database. If you forget to call the superclass method, the default behavior won't happen and the database won't get touched.

It's also important that you pass through the arguments that can be passed to the model method – that's what the `*args, **kwargs` bit does. Django will, from time to time, extend the capabilities of built-in model methods, adding new arguments. If you use `*args, **kwargs` in your method definitions, you are guaranteed that your code will automatically support those arguments when they are added.

Overridden model methods are not called on bulk operations

Note that the *`delete()`* method for an object is not necessarily called when *deleting objects in bulk using a QuerySet* or as a result of a *`cascading delete`*. To ensure customized delete logic gets executed, you can use *`pre_delete`* and/or *`post_delete`* signals.

Unfortunately, there isn't a workaround when *`creating`* or *`updating`* objects in bulk, since none of *`save()`*, *`pre_save`*, and *`post_save`* are called.

Executing custom SQL

Another common pattern is writing custom SQL statements in model methods and module-level methods. For more details on using raw SQL, see the documentation on using raw SQL.

Model inheritance

Model inheritance in Django works almost identically to the way normal class inheritance works in Python, but the basics at the beginning of the page should still be followed. That means the base class should subclass *`django.db.models.Model`*.

The only decision you have to make is whether you want the parent models to be models in their own right (with their own database tables), or if the parents are just holders of common information that will only be visible through the child models.

There are three styles of inheritance that are possible in Django.

1. Often, you will just want to use the parent class to hold information that you don't want to have to type out for each child model. This class isn't going to ever be used in isolation, so *Abstract base classes* are what you're after.
2. If you're subclassing an existing model (perhaps something from another application entirely) and want each model to have its own database table, *Multi-table inheritance* is the way to go.
3. Finally, if you only want to modify the Python-level behavior of a model, without changing the models fields in any way, you can use *Proxy models*.

Abstract base classes

Abstract base classes are useful when you want to put some common information into a number of other models. You write your base class and put `abstract=True` in the *Meta* class. This model will then not be used to create any database table. Instead, when it is used as a base class for other models, its fields will be added to those of the child class. It is an error to have fields in the abstract base class with the same name as those in the child (and Django will raise an exception).

An example:

```
from django.db import models

class CommonInfo(models.Model):
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

    class Meta:
        abstract = True

class Student(CommonInfo):
    home_group = models.CharField(max_length=5)
```

The `Student` model will have three fields: `name`, `age` and `home_group`. The `CommonInfo` model cannot be used as a normal Django model, since it is an abstract base class. It does not generate a database table or have a manager, and cannot be instantiated or saved directly.

For many uses, this type of model inheritance will be exactly what you want. It provides a way to factor out common information at the Python level, while still only creating one database table per child model at the database level.

`Meta` inheritance When an abstract base class is created, Django makes any *Meta* inner class you declared in the base class available as an attribute. If a child class does not declare its own *Meta* class, it will inherit the parent's *Meta*. If the child wants to extend the parent's *Meta* class, it can subclass it. For example:

```
from django.db import models

class CommonInfo(models.Model):
    # ...
    class Meta:
        abstract = True
        ordering = ['name']

class Student(CommonInfo):
    # ...
    class Meta(CommonInfo.Meta):
        db_table = 'student_info'
```

Django does make one adjustment to the *Meta* class of an abstract base class: before installing the *Meta* attribute, it sets `abstract=False`. This means that children of abstract base classes don't automatically become abstract classes themselves. Of course, you can make an abstract base class that inherits from another abstract base class. You just need to remember to explicitly set `abstract=True` each time.

Some attributes won't make sense to include in the *Meta* class of an abstract base class. For example, including `db_table` would mean that all the child classes (the ones that don't specify their own *Meta*) would use the same database table, which is almost certainly not what you want.

Be careful with `related_name` and `related_query_name` If you are using *related_name* or *related_query_name* on a `ForeignKey` or `ManyToManyField`, you must always specify a *unique* reverse name and query name for the field. This would normally cause a problem in abstract base classes, since the fields on this class are included into each of the child classes, with exactly the same values for the attributes (including *related_name* and *related_query_name*) each time.

To work around this problem, when you are using *related_name* or *related_query_name* in an abstract base class (only), part of the value should contain `'%(app_label)s'` and `'%(class)s'`.

- `'%(class)s'` is replaced by the lower-cased name of the child class that the field is used in.

- `'%(app_label)s'` is replaced by the lower-cased name of the app the child class is contained within. Each installed application name must be unique and the model class names within each app must also be unique, therefore the resulting name will end up being different.

For example, given an app `common/models.py`:

```
from django.db import models

class Base(models.Model):
    m2m = models.ManyToManyField(
        OtherModel,
        related_name="%(app_label)s_%(class)s_related",
        related_query_name="%(app_label)s_%(class)ss",
    )

    class Meta:
        abstract = True

class ChildA(Base):
    pass

class ChildB(Base):
    pass
```

Along with another app `rare/models.py`:

```
from common.models import Base

class ChildB(Base):
    pass
```

The reverse name of the `common.ChildA.m2m` field will be `common_childa_related` and the reverse query name will be `common_childas`. The reverse name of the `common.ChildB.m2m` field will be `common_childb_related` and the reverse query name will be `common_childbs`. Finally, the reverse name of the `rare.ChildB.m2m` field will be `rare_childb_related` and the reverse query name will be `rare_childbs`. It's up to you how you use the *'%(class)s'* and `'%(app_label)s` portion to construct your related name or related query name but if you forget to use it, Django will raise errors when you perform system checks (or run *migrate*).

If you don't specify a *related_name* attribute for a field in an abstract base class, the default reverse name will be the name of the child class followed by `'_set'`, just as it normally would be if you'd declared the field directly on the child class. For example, in the above code, if the *related_name* attribute was omitted, the reverse name for the `m2m` field would be `childa_set` in the `ChildA` case and `childb_set` for the `ChildB` field.

Interpolation of `'%(app_label)s'` and `'%(class)s'` for `related_query_name` was added.

Multi-table inheritance

The second type of model inheritance supported by Django is when each model in the hierarchy is a model all by itself. Each model corresponds to its own database table and can be queried and created individually. The inheritance relationship introduces links between the child model and each of its parents (via an automatically-created *OneToOneField*). For example:

```
from django.db import models

class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)
```

```
class Restaurant(Place):
    serves_hot_dogs = models.BooleanField(default=False)
    serves_pizza = models.BooleanField(default=False)
```

All of the fields of `Place` will also be available in `Restaurant`, although the data will reside in a different database table. So these are both possible:

```
>>> Place.objects.filter(name="Bob's Cafe")
>>> Restaurant.objects.filter(name="Bob's Cafe")
```

If you have a `Place` that is also a `Restaurant`, you can get from the `Place` object to the `Restaurant` object by using the lower-case version of the model name:

```
>>> p = Place.objects.get(id=12)
# If p is a Restaurant object, this will give the child class:
>>> p.restaurant
<Restaurant: ...>
```

However, if `p` in the above example was *not* a `Restaurant` (it had been created directly as a `Place` object or was the parent of some other class), referring to `p.restaurant` would raise a `Restaurant.DoesNotExist` exception.

`Meta` and multi-table inheritance In the multi-table inheritance situation, it doesn't make sense for a child class to inherit from its parent's *Meta* class. All the *Meta* options have already been applied to the parent class and applying them again would normally only lead to contradictory behavior (this is in contrast with the abstract base class case, where the base class doesn't exist in its own right).

So a child model does not have access to its parent's *Meta* class. However, there are a few limited cases where the child inherits behavior from the parent: if the child does not specify an *`ordering`* attribute or a *`get_latest_by`* attribute, it will inherit these from its parent.

If the parent has an ordering and you don't want the child to have any natural ordering, you can explicitly disable it:

```
class ChildModel(ParentModel):
    # ...
    class Meta:
        # Remove parent's ordering effect
        ordering = []
```

Inheritance and reverse relations Because multi-table inheritance uses an implicit *`OneToOneField`* to link the child and the parent, it's possible to move from the parent down to the child, as in the above example. However, this uses up the name that is the default *`related_name`* value for *`ForeignKey`* and *`ManyToManyField`* relations. If you are putting those types of relations on a subclass of the parent model, you **must** specify the *`related_name`* attribute on each such field. If you forget, Django will raise a validation error.

For example, using the above `Place` class again, let's create another subclass with a *`ManyToManyField`*:

```
class Supplier(Place):
    customers = models.ManyToManyField(Place)
```

This results in the error:

```
Reverse query name for 'Supplier.customers' clashes with reverse query
name for 'Supplier.place_ptr'.

HINT: Add or change a related_name argument to the definition for
'Supplier.customers' or 'Supplier.place_ptr'.
```

Adding `related_name` to the `customers` field as follows would resolve the error: `models.ManyToManyField(Place, related_name='provider')`.

Specifying the parent link field As mentioned, Django will automatically create a *`OneToOneField`* linking your child class back to any non-abstract parent models. If you want to control the name of the attribute linking back to the parent, you can create your own *`OneToOneField`* and set *`parent_link=True`* to indicate that your field is the link back to the parent class.

Proxy models

When using *multi-table inheritance*, a new database table is created for each subclass of a model. This is usually the desired behavior, since the subclass needs a place to store any additional data fields that are not present on the base class. Sometimes, however, you only want to change the Python behavior of a model – perhaps to change the default manager, or add a new method.

This is what proxy model inheritance is for: creating a *proxy* for the original model. You can create, delete and update instances of the proxy model and all the data will be saved as if you were using the original (non-proxied) model. The difference is that you can change things like the default model ordering or the default manager in the proxy, without having to alter the original.

Proxy models are declared like normal models. You tell Django that it's a proxy model by setting the *`proxy`* attribute of the `Meta` class to `True`.

For example, suppose you want to add a method to the `Person` model. You can do it like this:

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)

class MyPerson(Person):
    class Meta:
        proxy = True

    def do_something(self):
        # ...
        pass
```

The `MyPerson` class operates on the same database table as its parent `Person` class. In particular, any new instances of `Person` will also be accessible through `MyPerson`, and vice-versa:

```
>>> p = Person.objects.create(first_name="foobar")
>>> MyPerson.objects.get(first_name="foobar")
<MyPerson: foobar>
```

You could also use a proxy model to define a different default ordering on a model. You might not always want to order the `Person` model, but regularly order by the `last_name` attribute when you use the proxy. This is easy:

```
class OrderedPerson(Person):
    class Meta:
        ordering = ["last_name"]
        proxy = True
```

Now normal `Person` queries will be unordered and `OrderedPerson` queries will be ordered by `last_name`.

Proxy models inherit `Meta` attributes *in the same way as regular models*.

`QuerySet`s still return the model that was requested There is no way to have Django return, say, a `MyPerson` object whenever you query for `Person` objects. A queryset for `Person` objects will return those types of objects. The whole point of proxy objects is that code relying on the original `Person` will use those and your own code can use the extensions you included (that no other code is relying on anyway). It is not a way to replace the `Person` (or any other) model everywhere with something of your own creation.

Base class restrictions A proxy model must inherit from exactly one non-abstract model class. You can't inherit from multiple non-abstract models as the proxy model doesn't provide any connection between the rows in the different database tables. A proxy model can inherit from any number of abstract model classes, providing they do *not* define any model fields. A proxy model may also inherit from any number of proxy models that share a common non-abstract parent class.

In earlier versions, a proxy model couldn't inherit more than one proxy model that shared the same parent class.

Proxy model managers If you don't specify any model managers on a proxy model, it inherits the managers from its model parents. If you define a manager on the proxy model, it will become the default, although any managers defined on the parent classes will still be available.

Continuing our example from above, you could change the default manager used when you query the `Person` model like this:

```
from django.db import models

class NewManager(models.Manager):
    # ...
    pass

class MyPerson(Person):
    objects = NewManager()

    class Meta:
        proxy = True
```

If you wanted to add a new manager to the Proxy, without replacing the existing default, you can use the techniques described in the *custom manager* documentation: create a base class containing the new managers and inherit that after the primary base class:

```
# Create an abstract class for the new manager.
class ExtraManagers(models.Model):
    secondary = NewManager()

    class Meta:
        abstract = True

class MyPerson(Person, ExtraManagers):
    class Meta:
        proxy = True
```

You probably won't need to do this very often, but, when you do, it's possible.

Differences between proxy inheritance and unmanaged models Proxy model inheritance might look fairly similar to creating an unmanaged model, using the *`managed`* attribute on a model's `Meta` class.

With careful setting of *`Meta.db_table`* you could create an unmanaged model that shadows an existing model and adds Python methods to it. However, that would be very repetitive and fragile as you need to keep both copies synchronized if you make any changes.

On the other hand, proxy models are intended to behave exactly like the model they are proxying for. They are always in sync with the parent model since they directly inherit its fields and managers.

The general rules are:

1. If you are mirroring an existing model or database table and don't want all the original database table columns, use `Meta.managed=False`. That option is normally useful for modeling database views and tables not under the control of Django.
2. If you are wanting to change the Python-only behavior of a model, but keep all the same fields as in the original, use `Meta.proxy=True`. This sets things up so that the proxy model is an exact copy of the storage structure of the original model when data is saved.

Multiple inheritance

Just as with Python's subclassing, it's possible for a Django model to inherit from multiple parent models. Keep in mind that normal Python name resolution rules apply. The first base class that a particular name (e.g. *Meta*) appears in will be the one that is used; for example, this means that if multiple parents contain a *Meta* class, only the first one is going to be used, and all others will be ignored.

Generally, you won't need to inherit from multiple parents. The main use-case where this is useful is for "mix-in" classes: adding a particular extra field or method to every class that inherits the mix-in. Try to keep your inheritance hierarchies as simple and straightforward as possible so that you won't have to struggle to work out where a particular piece of information is coming from.

Note that inheriting from multiple models that have a common `id` primary key field will raise an error. To properly use multiple inheritance, you can use an explicit *AutoField* in the base models:

```
class Article(models.Model):
    article_id = models.AutoField(primary_key=True)
    ...

class Book(models.Model):
    book_id = models.AutoField(primary_key=True)
    ...

class BookReview(Book, Article):
    pass
```

Or use a common ancestor to hold the *AutoField*:

```
class Piece(models.Model):
    pass

class Article(Piece):
    ...

class Book(Piece):
    ...

class BookReview(Book, Article):
    pass
```

Field name "hiding" is not permitted

In normal Python class inheritance, it is permissible for a child class to override any attribute from the parent class. In Django, this isn't usually permitted for model fields. If a non-abstract model base class has a field called `author`, you

can't create another model field or define an attribute called `author` in any class that inherits from that base class.

This restriction doesn't apply to model fields inherited from an abstract model. Such fields may be overridden with another field or value, or be removed by setting `field_name = None`.

The ability to override abstract fields was added.

: Model managers are inherited from abstract base classes. Overriding an inherited field which is referenced by an inherited *`Manager`* may cause subtle bugs. See *custom managers and model inheritance*.

: Some fields define extra attributes on the model, e.g. a *`ForeignKey`* defines an extra attribute with `_id` appended to the field name, as well as `related_name` and `related_query_name` on the foreign model.

These extra attributes cannot be overridden unless the field that defines it is changed or removed so that it no longer defines the extra attribute.

Overriding fields in a parent model leads to difficulties in areas such as initializing new instances (specifying which field is being initialized in `Model.__init__`) and serialization. These are features which normal Python class inheritance doesn't have to deal with in quite the same way, so the difference between Django model inheritance and Python class inheritance isn't arbitrary.

This restriction only applies to attributes which are *`Field`* instances. Normal Python attributes can be overridden if you wish. It also only applies to the name of the attribute as Python sees it: if you are manually specifying the database column name, you can have the same column name appearing in both a child and an ancestor model for multi-table inheritance (they are columns in two different database tables).

Django will raise a *`FieldError`* if you override any model field in any ancestor model.

Organizing models in a package

The *`manage.py startapp`* command creates an application structure that includes a `models.py` file. If you have many models, organizing them in separate files may be useful.

To do so, create a `models` package. Remove `models.py` and create a `myapp/models/` directory with an `__init__.py` file and the files to store your models. You must import the models in the `__init__.py` file.

For example, if you had `organic.py` and `synthetic.py` in the `models` directory:

```
myapp/models/__init__.py
```

```
from .organic import Person
from .synthetic import Robot
```

Explicitly importing each model rather than using `from .models import *` has the advantages of not cluttering the namespace, making code more readable, and keeping code analysis tools useful.

:

The Models Reference Covers all the model related APIs including model fields, related objects, and `QuerySet`.

3.2.2 Making queries

Once you've created your data models, Django automatically gives you a database-abstraction API that lets you create, retrieve, update and delete objects. This document explains how to use this API. Refer to the data model reference for full details of all the various model lookup options.

Throughout this guide (and in the reference), we'll refer to the following models, which comprise a Weblog application:

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Author(models.Model):
    name = models.CharField(max_length=200)
    email = models.EmailField()

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Entry(models.Model):
    blog = models.ForeignKey(Blog)
    headline = models.CharField(max_length=255)
    body_text = models.TextField()
    pub_date = models.DateField()
    mod_date = models.DateField()
    authors = models.ManyToManyField(Author)
    n_comments = models.IntegerField()
    n_pingbacks = models.IntegerField()
    rating = models.IntegerField()

    def __str__(self):              # __unicode__ on Python 2
        return self.headline
```

Creating objects

To represent database-table data in Python objects, Django uses an intuitive system: A model class represents a database table, and an instance of that class represents a particular record in the database table.

To create an object, instantiate it using keyword arguments to the model class, then call *save()* to save it to the database.

Assuming models live in a file `mysite/blog/models.py`, here's an example:

```
>>> from blog.models import Blog
>>> b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')
>>> b.save()
```

This performs an `INSERT` SQL statement behind the scenes. Django doesn't hit the database until you explicitly call *save()*.

The *save()* method has no return value.

:

save() takes a number of advanced options not described here. See the documentation for *save()* for complete details.

To create and save an object in a single step, use the *create()* method.

Saving changes to objects

To save changes to an object that's already in the database, use *save()*.

Given a Blog instance b5 that has already been saved to the database, this example changes its name and updates its record in the database:

```
>>> b5.name = 'New name'
>>> b5.save()
```

This performs an UPDATE SQL statement behind the scenes. Django doesn't hit the database until you explicitly call *save()*.

Saving ForeignKey and ManyToManyField fields

Updating a *ForeignKey* field works exactly the same way as saving a normal field – simply assign an object of the right type to the field in question. This example updates the blog attribute of an Entry instance entry, assuming appropriate instances of Entry and Blog are already saved to the database (so we can retrieve them below):

```
>>> from blog.models import Entry
>>> entry = Entry.objects.get(pk=1)
>>> cheese_blog = Blog.objects.get(name="Cheddar Talk")
>>> entry.blog = cheese_blog
>>> entry.save()
```

Updating a *ManyToManyField* works a little differently – use the *add()* method on the field to add a record to the relation. This example adds the Author instance joe to the entry object:

```
>>> from blog.models import Author
>>> joe = Author.objects.create(name="Joe")
>>> entry.authors.add(joe)
```

To add multiple records to a *ManyToManyField* in one go, include multiple arguments in the call to *add()*, like this:

```
>>> john = Author.objects.create(name="John")
>>> paul = Author.objects.create(name="Paul")
>>> george = Author.objects.create(name="George")
>>> ringo = Author.objects.create(name="Ringo")
>>> entry.authors.add(john, paul, george, ringo)
```

Django will complain if you try to assign or add an object of the wrong type.

Retrieving objects

To retrieve objects from your database, construct a *QuerySet* via a *Manager* on your model class.

A *QuerySet* represents a collection of objects from your database. It can have zero, one or many *filters*. Filters narrow down the query results based on the given parameters. In SQL terms, a *QuerySet* equates to a SELECT statement, and a filter is a limiting clause such as WHERE or LIMIT.

You get a *QuerySet* by using your model's *Manager*. Each model has at least one *Manager*, and it's called *objects* by default. Access it directly via the model class, like so:

```
>>> Blog.objects
<django.db.models.manager.Manager object at ...>
>>> b = Blog(name='Foo', tagline='Bar')
>>> b.objects
```

```
Traceback:
    ...
AttributeError: "Manager isn't accessible via Blog instances."
```

: `Managers` are accessible only via model classes, rather than from model instances, to enforce a separation between "table-level" operations and "record-level" operations.

The *Manager* is the main source of `QuerySets` for a model. For example, `Blog.objects.all()` returns a *QuerySet* that contains all `Blog` objects in the database.

Retrieving all objects

The simplest way to retrieve objects from a table is to get all of them. To do this, use the *all()* method on a *Manager*:

```
>>> all_entries = Entry.objects.all()
```

The *all()* method returns a *QuerySet* of all the objects in the database.

Retrieving specific objects with filters

The *QuerySet* returned by *all()* describes all objects in the database table. Usually, though, you'll need to select only a subset of the complete set of objects.

To create such a subset, you refine the initial *QuerySet*, adding filter conditions. The two most common ways to refine a *QuerySet* are:

filter(kwargs)** Returns a new *QuerySet* containing objects that match the given lookup parameters.

exclude(kwargs)** Returns a new *QuerySet* containing objects that do *not* match the given lookup parameters.

The lookup parameters (`**kwargs` in the above function definitions) should be in the format described in *Field lookups* below.

For example, to get a *QuerySet* of blog entries from the year 2006, use *filter()* like so:

```
Entry.objects.filter(pub_date__year=2006)
```

With the default manager class, it is the same as:

```
Entry.objects.all().filter(pub_date__year=2006)
```

Chaining filters The result of refining a *QuerySet* is itself a *QuerySet*, so it's possible to chain refinements together. For example:

```
>>> Entry.objects.filter(
...     headline__startswith='What'
... ).exclude(
...     pub_date__gte=datetime.date.today()
... ).filter(
...     pub_date__gte=datetime(2005, 1, 30)
... )
```

This takes the initial *QuerySet* of all entries in the database, adds a filter, then an exclusion, then another filter. The final result is a *QuerySet* containing all entries with a headline that starts with "What", that were published between January 30, 2005, and the current day.

Filtered QuerySets are unique Each time you refine a *QuerySet*, you get a brand-new *QuerySet* that is in no way bound to the previous *QuerySet*. Each refinement creates a separate and distinct *QuerySet* that can be stored, used and reused.

Example:

```
>>> q1 = Entry.objects.filter(headline__startswith="What")
>>> q2 = q1.exclude(pub_date__gte=datetime.date.today())
>>> q3 = q1.filter(pub_date__gte=datetime.date.today())
```

These three QuerySets are separate. The first is a base *QuerySet* containing all entries that contain a headline starting with "What". The second is a subset of the first, with an additional criteria that excludes records whose pub_date is today or in the future. The third is a subset of the first, with an additional criteria that selects only the records whose pub_date is today or in the future. The initial *QuerySet* (q1) is unaffected by the refinement process.

QuerySets are lazy QuerySets are lazy – the act of creating a *QuerySet* doesn't involve any database activity. You can stack filters together all day long, and Django won't actually run the query until the *QuerySet* is *evaluated*. Take a look at this example:

```
>>> q = Entry.objects.filter(headline__startswith="What")
>>> q = q.filter(pub_date__lte=datetime.date.today())
>>> q = q.exclude(body_text__icontains="food")
>>> print(q)
```

Though this looks like three database hits, in fact it hits the database only once, at the last line (print(q)). In general, the results of a *QuerySet* aren't fetched from the database until you "ask" for them. When you do, the *QuerySet* is *evaluated* by accessing the database. For more details on exactly when evaluation takes place, see *When QuerySets are evaluated*.

Retrieving a single object with get()

filter() will always give you a *QuerySet*, even if only a single object matches the query - in this case, it will be a *QuerySet* containing a single element.

If you know there is only one object that matches your query, you can use the *get()* method on a *Manager* which returns the object directly:

```
>>> one_entry = Entry.objects.get(pk=1)
```

You can use any query expression with *get()*, just like with *filter()* - again, see *Field lookups* below.

Note that there is a difference between using *get()*, and using *filter()* with a slice of [0]. If there are no results that match the query, *get()* will raise a DoesNotExist exception. This exception is an attribute of the model class that the query is being performed on - so in the code above, if there is no Entry object with a primary key of 1, Django will raise Entry.DoesNotExist.

Similarly, Django will complain if more than one item matches the *get()* query. In this case, it will raise *MultipleObjectsReturned*, which again is an attribute of the model class itself.

Other `QuerySet` methods

Most of the time you'll use *`all()`*, *`get()`*, *`filter()`* and *`exclude()`* when you need to look up objects from the database. However, that's far from all there is; see the *QuerySet API Reference* for a complete list of all the various *`QuerySet`* methods.

Limiting `QuerySets`

Use a subset of Python's array-slicing syntax to limit your *`QuerySet`* to a certain number of results. This is the equivalent of SQL's `LIMIT` and `OFFSET` clauses.

For example, this returns the first 5 objects (`LIMIT 5`):

```
>>> Entry.objects.all()[:5]
```

This returns the sixth through tenth objects (`OFFSET 5 LIMIT 5`):

```
>>> Entry.objects.all()[5:10]
```

Negative indexing (i.e. `Entry.objects.all()[-1]`) is not supported.

Generally, slicing a *`QuerySet`* returns a new *`QuerySet`* – it doesn't evaluate the query. An exception is if you use the "step" parameter of Python slice syntax. For example, this would actually execute the query in order to return a list of every *second* object of the first 10:

```
>>> Entry.objects.all()[:10:2]
```

To retrieve a *single* object rather than a list (e.g. `SELECT foo FROM bar LIMIT 1`), use a simple index instead of a slice. For example, this returns the first `Entry` in the database, after ordering entries alphabetically by headline:

```
>>> Entry.objects.order_by('headline')[0]
```

This is roughly equivalent to:

```
>>> Entry.objects.order_by('headline')[0:1].get()
```

Note, however, that the first of these will raise `IndexError` while the second will raise `DoesNotExist` if no objects match the given criteria. See *`get()`* for more details.

Field lookups

Field lookups are how you specify the meat of an SQL `WHERE` clause. They're specified as keyword arguments to the *`QuerySet`* methods *`filter()`*, *`exclude()`* and *`get()`*.

Basic lookups keyword arguments take the form `field__lookuptype=value`. (That's a double-underscore). For example:

```
>>> Entry.objects.filter(pub_date__lte='2006-01-01')
```

translates (roughly) into the following SQL:

```
SELECT * FROM blog_entry WHERE pub_date <= '2006-01-01';
```

How this is possible

Python has the ability to define functions that accept arbitrary name-value arguments whose names and values are evaluated at runtime. For more information, see Keyword Arguments in the official Python tutorial.

The field specified in a lookup has to be the name of a model field. There's one exception though, in case of a *ForeignKey* you can specify the field name suffixed with `_id`. In this case, the value parameter is expected to contain the raw value of the foreign model's primary key. For example:

```
>>> Entry.objects.filter(blog_id=4)
```

If you pass an invalid keyword argument, a lookup function will raise `TypeError`.

The database API supports about two dozen lookup types; a complete reference can be found in the *field lookup reference*. To give you a taste of what's available, here's some of the more common lookups you'll probably use:

exact An "exact" match. For example:

```
>>> Entry.objects.get(headline__exact="Cat bites dog")
```

Would generate SQL along these lines:

```
SELECT ... WHERE headline = 'Cat bites dog';
```

If you don't provide a lookup type – that is, if your keyword argument doesn't contain a double underscore – the lookup type is assumed to be `exact`.

For example, the following two statements are equivalent:

```
>>> Blog.objects.get(id__exact=14)  # Explicit form
>>> Blog.objects.get(id=14)         # __exact is implied
```

This is for convenience, because `exact` lookups are the common case.

iexact A case-insensitive match. So, the query:

```
>>> Blog.objects.get(name__iexact="beatles blog")
```

Would match a `Blog` titled `"Beatles Blog"`, `"beatles blog"`, or even `"BeAtlES blOG"`.

contains Case-sensitive containment test. For example:

```
Entry.objects.get(headline__contains='Lennon')
```

Roughly translates to this SQL:

```
SELECT ... WHERE headline LIKE '%Lennon%';
```

Note this will match the headline `'Today Lennon honored'` but not `'today lennon honored'`.

There's also a case-insensitive version, *icontains*.

startswith*, *endswith Starts-with and ends-with search, respectively. There are also case-insensitive versions called *istartswith* and *iendswith*.

Again, this only scratches the surface. A complete reference can be found in the *field lookup reference*.

Lookups that span relationships

Django offers a powerful and intuitive way to "follow" relationships in lookups, taking care of the SQL `JOIN`s for you automatically, behind the scenes. To span a relationship, just use the field name of related fields across models, separated by double underscores, until you get to the field you want.

This example retrieves all `Entry` objects with a `Blog` whose `name` is `'Beatles Blog'`:

```
>>> Entry.objects.filter(blog__name='Beatles Blog')
```

This spanning can be as deep as you'd like.

It works backwards, too. To refer to a "reverse" relationship, just use the lowercase name of the model.

This example retrieves all `Blog` objects which have at least one `Entry` whose `headline` contains `'Lennon'`:

```
>>> Blog.objects.filter(entry__headline__contains='Lennon')
```

If you are filtering across multiple relationships and one of the intermediate models doesn't have a value that meets the filter condition, Django will treat it as if there is an empty (all values are `NULL`), but valid, object there. All this means is that no error will be raised. For example, in this filter:

```
Blog.objects.filter(entry__authors__name='Lennon')
```

(if there was a related `Author` model), if there was no `author` associated with an entry, it would be treated as if there was also no `name` attached, rather than raising an error because of the missing `author`. Usually this is exactly what you want to have happen. The only case where it might be confusing is if you are using *`isnull`*. Thus:

```
Blog.objects.filter(entry__authors__name__isnull=True)
```

will return `Blog` objects that have an empty `name` on the `author` and also those which have an empty `author` on the `entry`. If you don't want those latter objects, you could write:

```
Blog.objects.filter(entry__authors__isnull=False, entry__authors__name__isnull=True)
```

Spanning multi-valued relationships When you are filtering an object based on a *`ManyToManyField`* or a reverse *`ForeignKey`*, there are two different sorts of filter you may be interested in. Consider the `Blog`/`Entry` relationship (`Blog` to `Entry` is a one-to-many relation). We might be interested in finding blogs that have an entry which has both *"Lennon"* in the headline and was published in 2008. Or we might want to find blogs that have an entry with *"Lennon"* in the headline as well as an entry that was published in 2008. Since there are multiple entries associated with a single `Blog`, both of these queries are possible and make sense in some situations.

The same type of situation arises with a *`ManyToManyField`*. For example, if an `Entry` has a *`ManyToManyField`* called `tags`, we might want to find entries linked to tags called *"music"* and *"bands"* or we might want an entry that contains a tag with a name of *"music"* and a status of *"public"*.

To handle both of these situations, Django has a consistent way of processing *`filter()`* calls. Everything inside a single *`filter()`* call is applied simultaneously to filter out items matching all those requirements. Successive *`filter()`* calls further restrict the set of objects, but for multi-valued relations, they apply to any object linked to the primary model, not necessarily those objects that were selected by an earlier *`filter()`* call.

That may sound a bit confusing, so hopefully an example will clarify. To select all blogs that contain entries with both *"Lennon"* in the headline and that were published in 2008 (the same entry satisfying both conditions), we would write:

```
Blog.objects.filter(entry__headline__contains='Lennon', entry__pub_date__year=2008)
```

To select all blogs that contain an entry with *"Lennon"* in the headline **as well as** an entry that was published in 2008, we would write:

```
Blog.objects.filter(entry__headline__contains='Lennon').filter(entry__pub_date__year=2008)
```

Suppose there is only one blog that had both entries containing *"Lennon"* and entries from 2008, but that none of the entries from 2008 contained *"Lennon"*. The first query would not return any blogs, but the second query would return that one blog.

In the second example, the first filter restricts the queryset to all those blogs linked to entries with *"Lennon"* in the headline. The second filter restricts the set of blogs *further* to those that are also linked to entries that were published in 2008. The entries selected by the second filter may or may not be the same as the entries in the first filter. We are filtering the `Blog` items with each filter statement, not the `Entry` items.

: The behavior of *`filter()`* for queries that span multi-value relationships, as described above, is not implemented equivalently for *`exclude()`*. Instead, the conditions in a single *`exclude()`* call will not necessarily refer to the same item.

For example, the following query would exclude blogs that contain *both* entries with *"Lennon"* in the headline *and* entries published in 2008:

```
Blog.objects.exclude(
    entry__headline__contains='Lennon',
    entry__pub_date__year=2008,
)
```

However, unlike the behavior when using *`filter()`*, this will not limit blogs based on entries that satisfy both conditions. In order to do that, i.e. to select all blogs that do not contain entries published with *"Lennon"* that were published in 2008, you need to make two queries:

```
Blog.objects.exclude(
    entry=Entry.objects.filter(
        headline__contains='Lennon',
        pub_date__year=2008,
    ),
)
```

Filters can reference fields on the model

In the examples given so far, we have constructed filters that compare the value of a model field with a constant. But what if you want to compare the value of a model field with another field on the same model?

Django provides *`F expressions`* to allow such comparisons. Instances of `F()` act as a reference to a model field within a query. These references can then be used in query filters to compare the values of two different fields on the same model instance.

For example, to find a list of all blog entries that have had more comments than pingbacks, we construct an `F()` object to reference the pingback count, and use that `F()` object in the query:

```
>>> from django.db.models import F
>>> Entry.objects.filter(n_comments__gt=F('n_pingbacks'))
```

Django supports the use of addition, subtraction, multiplication, division, modulo, and power arithmetic with `F()` objects, both with constants and with other `F()` objects. To find all the blog entries with more than *twice* as many comments as pingbacks, we modify the query:

```
>>> Entry.objects.filter(n_comments__gt=F('n_pingbacks') * 2)
```

To find all the entries where the rating of the entry is less than the sum of the pingback count and comment count, we would issue the query:

```
>>> Entry.objects.filter(rating__lt=F('n_comments') + F('n_pingbacks'))
```

You can also use the double underscore notation to span relationships in an `F()` object. An `F()` object with a double underscore will introduce any joins needed to access the related object. For example, to retrieve all the entries where the author's name is the same as the blog name, we could issue the query:

```
>>> Entry.objects.filter(authors__name=F('blog__name'))
```

For date and date/time fields, you can add or subtract a `timedelta` object. The following would return all entries that were modified more than 3 days after they were published:

```
>>> from datetime import timedelta
>>> Entry.objects.filter(mod_date__gt=F('pub_date') + timedelta(days=3))
```

The `F()` objects support bitwise operations by `.bitand()` and `.bitor()`, for example:

```
>>> F('somefield').bitand(16)
```

The `pk` lookup shortcut

For convenience, Django provides a `pk` lookup shortcut, which stands for "primary key".

In the example `Blog` model, the primary key is the `id` field, so these three statements are equivalent:

```
>>> Blog.objects.get(id__exact=14) # Explicit form
>>> Blog.objects.get(id=14) # __exact is implied
>>> Blog.objects.get(pk=14) # pk implies id__exact
```

The use of `pk` isn't limited to `__exact` queries – any query term can be combined with `pk` to perform a query on the primary key of a model:

```
# Get blogs entries with id 1, 4 and 7
>>> Blog.objects.filter(pk__in=[1,4,7])

# Get all blog entries with id > 14
>>> Blog.objects.filter(pk__gt=14)
```

`pk` lookups also work across joins. For example, these three statements are equivalent:

```
>>> Entry.objects.filter(blog__id__exact=3) # Explicit form
>>> Entry.objects.filter(blog__id=3)        # __exact is implied
>>> Entry.objects.filter(blog__pk=3)        # __pk implies __id__exact
```

Escaping percent signs and underscores in `LIKE` statements

The field lookups that equate to `LIKE` SQL statements (`iexact`, `contains`, `icontains`, `startswith`, `istartswith`, `endswith` and `iendswith`) will automatically escape the two special characters used in `LIKE` statements – the percent sign and the underscore. (In a `LIKE` statement, the percent sign signifies a multiple-character wildcard and the underscore signifies a single-character wildcard.)

This means things should work intuitively, so the abstraction doesn't leak. For example, to retrieve all the entries that contain a percent sign, just use the percent sign as any other character:

```
>>> Entry.objects.filter(headline__contains='%')
```

Django takes care of the quoting for you; the resulting SQL will look something like this:

```
SELECT ... WHERE headline LIKE '%\%%';
```

Same goes for underscores. Both percentage signs and underscores are handled for you transparently.

Caching and QuerySets

Each *QuerySet* contains a cache to minimize database access. Understanding how it works will allow you to write the most efficient code.

In a newly created *QuerySet*, the cache is empty. The first time a *QuerySet* is evaluated – and, hence, a database query happens – Django saves the query results in the *QuerySet*'s cache and returns the results that have been explicitly requested (e.g., the next element, if the *QuerySet* is being iterated over). Subsequent evaluations of the *QuerySet* reuse the cached results.

Keep this caching behavior in mind, because it may bite you if you don't use your *QuerySet*s correctly. For example, the following will create two *QuerySet*s, evaluate them, and throw them away:

```
>>> print([e.headline for e in Entry.objects.all()])
>>> print([e.pub_date for e in Entry.objects.all()])
```

That means the same database query will be executed twice, effectively doubling your database load. Also, there's a possibility the two lists may not include the same database records, because an `Entry` may have been added or deleted in the split second between the two requests.

To avoid this problem, simply save the *QuerySet* and reuse it:

```
>>> queryset = Entry.objects.all()
>>> print([p.headline for p in queryset]) # Evaluate the query set.
>>> print([p.pub_date for p in queryset]) # Re-use the cache from the evaluation.
```

When QuerySets are not cached Querysets do not always cache their results. When evaluating only *part* of the queryset, the cache is checked, but if it is not populated then the items returned by the subsequent query are not cached. Specifically, this means that *limiting the queryset* using an array slice or an index will not populate the cache.

For example, repeatedly getting a certain index in a queryset object will query the database each time:

```
>>> queryset = Entry.objects.all()
>>> print(queryset[5]) # Queries the database
>>> print(queryset[5]) # Queries the database again
```

However, if the entire queryset has already been evaluated, the cache will be checked instead:

```
>>> queryset = Entry.objects.all()
>>> [entry for entry in queryset] # Queries the database
>>> print(queryset[5]) # Uses cache
>>> print(queryset[5]) # Uses cache
```

Here are some examples of other actions that will result in the entire queryset being evaluated and therefore populate the cache:

```
>>> [entry for entry in queryset]
>>> bool(queryset)
>>> entry in queryset
>>> list(queryset)
```

: Simply printing the queryset will not populate the cache. This is because the call to `__repr__()` only returns a slice of the entire queryset.

Complex lookups with Q objects

Keyword argument queries – in *filter()*, etc. – are "AND"ed together. If you need to execute more complex queries (for example, queries with OR statements), you can use *Q objects*.

A *Q object* (`django.db.models.Q`) is an object used to encapsulate a collection of keyword arguments. These keyword arguments are specified as in "Field lookups" above.

For example, this Q object encapsulates a single LIKE query:

```
from django.db.models import Q
Q(question__startswith='What')
```

Q objects can be combined using the & and | operators. When an operator is used on two Q objects, it yields a new Q object.

For example, this statement yields a single Q object that represents the "OR" of two "question__startswith" queries:

```
Q(question__startswith='Who') | Q(question__startswith='What')
```

This is equivalent to the following SQL WHERE clause:

```
WHERE question LIKE 'Who%' OR question LIKE 'What%'
```

You can compose statements of arbitrary complexity by combining Q objects with the & and | operators and use parenthetical grouping. Also, Q objects can be negated using the ~ operator, allowing for combined lookups that combine both a normal query and a negated (NOT) query:

```
Q(question__startswith='Who') | ~Q(pub_date__year=2005)
```

Each lookup function that takes keyword-arguments (e.g. *filter()*, *exclude()*, *get()*) can also be passed one or more Q objects as positional (not-named) arguments. If you provide multiple Q object arguments to a lookup function, the arguments will be "AND"ed together. For example:

```
Poll.objects.get(
    Q(question__startswith='Who'),
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6))
)
```

... roughly translates into the SQL:

```
SELECT * from polls WHERE question LIKE 'Who%'
    AND (pub_date = '2005-05-02' OR pub_date = '2005-05-06')
```

Lookup functions can mix the use of Q objects and keyword arguments. All arguments provided to a lookup function (be they keyword arguments or Q objects) are "AND"ed together. However, if a Q object is provided, it must precede the definition of any keyword arguments. For example:

```
Poll.objects.get(
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)),
    question__startswith='Who',
)
```

... would be a valid query, equivalent to the previous example; but:

```
# INVALID QUERY
Poll.objects.get(
    question__startswith='Who',
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6))
)
```

... would not be valid.

:

The OR lookups examples in the Django unit tests show some possible uses of `Q`.

Comparing objects

To compare two model instances, just use the standard Python comparison operator, the double equals sign: `==`. Behind the scenes, that compares the primary key values of two models.

Using the `Entry` example above, the following two statements are equivalent:

```
>>> some_entry == other_entry
>>> some_entry.id == other_entry.id
```

If a model's primary key isn't called `id`, no problem. Comparisons will always use the primary key, whatever it's called. For example, if a model's primary key field is called `name`, these two statements are equivalent:

```
>>> some_obj == other_obj
>>> some_obj.name == other_obj.name
```

Deleting objects

The delete method, conveniently, is named *`delete()`*. This method immediately deletes the object and returns the number of objects deleted and a dictionary with the number of deletions per object type. Example:

```
>>> e.delete()
(1, {'weblog.Entry': 1})
```

The return value describing the number of objects deleted was added.

You can also delete objects in bulk. Every *`QuerySet`* has a *`delete()`* method, which deletes all members of that *`QuerySet`*.

For example, this deletes all `Entry` objects with a `pub_date` year of 2005:

```
>>> Entry.objects.filter(pub_date__year=2005).delete()
(5, {'webapp.Entry': 5})
```

Keep in mind that this will, whenever possible, be executed purely in SQL, and so the `delete()` methods of individual object instances will not necessarily be called during the process. If you've provided a custom `delete()` method on a model class and want to ensure that it is called, you will need to "manually" delete instances of that model (e.g., by iterating over a *`QuerySet`* and calling `delete()` on each object individually) rather than using the bulk *`delete()`* method of a *`QuerySet`*.

The return value describing the number of objects deleted was added.

When Django deletes an object, by default it emulates the behavior of the SQL constraint `ON DELETE CASCADE` – in other words, any objects which had foreign keys pointing at the object to be deleted will be deleted along with it. For example:

```
b = Blog.objects.get(pk=1)
# This will delete the Blog and all of its Entry objects.
b.delete()
```

This cascade behavior is customizable via the *`on_delete`* argument to the *`ForeignKey`*.

Note that *delete()* is the only *QuerySet* method that is not exposed on a *Manager* itself. This is a safety mechanism to prevent you from accidentally requesting `Entry.objects.delete()`, and deleting *all* the entries. If you *do* want to delete all the objects, then you have to explicitly request a complete query set:

```
Entry.objects.all().delete()
```

Copying model instances

Although there is no built-in method for copying model instances, it is possible to easily create new instance with all fields' values copied. In the simplest case, you can just set `pk` to `None`. Using our blog example:

```
blog = Blog(name='My blog', tagline='Blogging is easy')
blog.save() # blog.pk == 1

blog.pk = None
blog.save() # blog.pk == 2
```

Things get more complicated if you use inheritance. Consider a subclass of `Blog`:

```
class ThemeBlog(Blog):
    theme = models.CharField(max_length=200)

django_blog = ThemeBlog(name='Django', tagline='Django is easy', theme='python')
django_blog.save() # django_blog.pk == 3
```

Due to how inheritance works, you have to set both `pk` and `id` to None:

```
django_blog.pk = None
django_blog.id = None
django_blog.save() # django_blog.pk == 4
```

This process does not copy related objects. If you want to copy relations, you have to write a little bit more code. In our example, `Entry` has a many to many field to `Author`:

```
entry = Entry.objects.all()[0] # some previous entry
old_authors = entry.authors.all()
entry.pk = None
entry.save()
entry.authors = old_authors # saves new many2many relations
```

Updating multiple objects at once

Sometimes you want to set a field to a particular value for all the objects in a *QuerySet*. You can do this with the *update()* method. For example:

```
# Update all the headlines with pub_date in 2007.
Entry.objects.filter(pub_date__year=2007).update(headline='Everything is the same')
```

You can only set non-relation fields and *ForeignKey* fields using this method. To update a non-relation field, provide the new value as a constant. To update *ForeignKey* fields, set the new value to be the new model instance you want to point to. For example:

```
>>> b = Blog.objects.get(pk=1)

# Change every Entry so that it belongs to this Blog.
>>> Entry.objects.all().update(blog=b)
```

The `update()` method is applied instantly and returns the number of rows matched by the query (which may not be equal to the number of rows updated if some rows already have the new value). The only restriction on the *QuerySet* being updated is that it can only access one database table: the model's main table. You can filter based on related fields, but you can only update columns in the model's main table. Example:

```
>>> b = Blog.objects.get(pk=1)

# Update all the headlines belonging to this Blog.
>>> Entry.objects.select_related().filter(blog=b).update(headline='Everything is the same')
```

Be aware that the `update()` method is converted directly to an SQL statement. It is a bulk operation for direct updates. It doesn't run any *save()* methods on your models, or emit the `pre_save` or `post_save` signals (which are a consequence of calling *save()*), or honor the *auto_now* field option. If you want to save every item in a *QuerySet* and make sure that the *save()* method is called on each instance, you don't need any special function to handle that. Just loop over them and call *save()*:

```
for item in my_queryset:
    item.save()
```

Calls to update can also use *F expressions* to update one field based on the value of another field in the model. This is especially useful for incrementing counters based upon their current value. For example, to increment the pingback count for every entry in the blog:

```
>>> Entry.objects.all().update(n_pingbacks=F('n_pingbacks') + 1)
```

However, unlike `F()` objects in filter and exclude clauses, you can't introduce joins when you use `F()` objects in an update – you can only reference fields local to the model being updated. If you attempt to introduce a join with an `F()` object, a `FieldError` will be raised:

```
# THIS WILL RAISE A FieldError
>>> Entry.objects.update(headline=F('blog__name'))
```

Related objects

When you define a relationship in a model (i.e., a *ForeignKey*, *OneToOneField*, or *ManyToManyField*), instances of that model will have a convenient API to access the related object(s).

Using the models at the top of this page, for example, an `Entry` object `e` can get its associated `Blog` object by accessing the `blog` attribute: `e.blog`.

(Behind the scenes, this functionality is implemented by Python descriptors. This shouldn't really matter to you, but we point it out here for the curious.)

Django also creates API accessors for the "other" side of the relationship – the link from the related model to the model that defines the relationship. For example, a `Blog` object `b` has access to a list of all related `Entry` objects via the `entry_set` attribute: `b.entry_set.all()`.

All examples in this section use the sample `Blog`, `Author` and `Entry` models defined at the top of this page.

One-to-many relationships

Forward If a model has a *ForeignKey*, instances of that model will have access to the related (foreign) object via a simple attribute of the model.

Example:

```
>>> e = Entry.objects.get(id=2)
>>> e.blog # Returns the related Blog object.
```

You can get and set via a foreign-key attribute. As you may expect, changes to the foreign key aren't saved to the database until you call *save()*. Example:

```
>>> e = Entry.objects.get(id=2)
>>> e.blog = some_blog
>>> e.save()
```

If a *ForeignKey* field has `null=True` set (i.e., it allows `NULL` values), you can assign `None` to remove the relation. Example:

```
>>> e = Entry.objects.get(id=2)
>>> e.blog = None
>>> e.save() # "UPDATE blog_entry SET blog_id = NULL ...;"
```

Forward access to one-to-many relationships is cached the first time the related object is accessed. Subsequent accesses to the foreign key on the same object instance are cached. Example:

```
>>> e = Entry.objects.get(id=2)
>>> print(e.blog)  # Hits the database to retrieve the associated Blog.
>>> print(e.blog)  # Doesn't hit the database; uses cached version.
```

Note that the *select_related() QuerySet* method recursively prepopulates the cache of all one-to-many relationships ahead of time. Example:

```
>>> e = Entry.objects.select_related().get(id=2)
>>> print(e.blog)  # Doesn't hit the database; uses cached version.
>>> print(e.blog)  # Doesn't hit the database; uses cached version.
```

Following relationships "backward" If a model has a *ForeignKey*, instances of the foreign-key model will have access to a *Manager* that returns all instances of the first model. By default, this *Manager* is named `FOO_set`, where `FOO` is the source model name, lowercased. This *Manager* returns `QuerySets`, which can be filtered and manipulated as described in the "Retrieving objects" section above.

Example:

```
>>> b = Blog.objects.get(id=1)
>>> b.entry_set.all() # Returns all Entry objects related to Blog.

# b.entry_set is a Manager that returns QuerySets.
>>> b.entry_set.filter(headline__contains='Lennon')
>>> b.entry_set.count()
```

You can override the `FOO_set` name by setting the *related_name* parameter in the *ForeignKey* definition. For example, if the `Entry` model was altered to `blog = ForeignKey(Blog, on_delete=models.CASCADE, related_name='entries')`, the above example code would look like this:

```
>>> b = Blog.objects.get(id=1)
>>> b.entries.all() # Returns all Entry objects related to Blog.

# b.entries is a Manager that returns QuerySets.
>>> b.entries.filter(headline__contains='Lennon')
>>> b.entries.count()
```

Using a custom reverse manager By default the *RelatedManager* used for reverse relations is a subclass of the *default manager* for that model. If you would like to specify a different manager for a given query you can use the following syntax:

```
from django.db import models

class Entry(models.Model):
    #...
    objects = models.Manager()  # Default Manager
    entries = EntryManager()    # Custom Manager

b = Blog.objects.get(id=1)
b.entry_set(manager='entries').all()
```

If `EntryManager` performed default filtering in its `get_queryset()` method, that filtering would apply to the `all()` call.

Of course, specifying a custom reverse manager also enables you to call its custom methods:

```
b.entry_set(manager='entries').is_published()
```

Additional methods to handle related objects In addition to the *QuerySet* methods defined in "Retrieving objects" above, the *ForeignKey Manager* has additional methods used to handle the set of related objects. A synopsis of each is below, and complete details can be found in the related objects reference.

`add(obj1, obj2, ...)` Adds the specified model objects to the related object set.

`create(**kwargs)` Creates a new object, saves it and puts it in the related object set. Returns the newly created object.

`remove(obj1, obj2, ...)` Removes the specified model objects from the related object set.

`clear()` Removes all objects from the related object set.

`set(objs)` Replace the set of related objects.

To assign the members of a related set, use the `set()` method with an iterable of object instances or a list of primary key values. For example:

```
b = Blog.objects.get(id=1)
b.entry_set.set([e1, e2])
```

In this example, `e1` and `e2` can be full Entry instances, or integer primary key values.

If the `clear()` method is available, any pre-existing objects will be removed from the `entry_set` before all objects in the iterable (in this case, a list) are added to the set. If the `clear()` method is *not* available, all objects in the iterable will be added without removing any existing elements.

Each "reverse" operation described in this section has an immediate effect on the database. Every addition, creation and deletion is immediately and automatically saved to the database.

Many-to-many relationships

Both ends of a many-to-many relationship get automatic API access to the other end. The API works just as a "backward" one-to-many relationship, above.

The only difference is in the attribute naming: The model that defines the *ManyToManyField* uses the attribute name of that field itself, whereas the "reverse" model uses the lowercased model name of the original model, plus `'_set'` (just like reverse one-to-many relationships).

An example makes this easier to understand:

```
e = Entry.objects.get(id=3)
e.authors.all() # Returns all Author objects for this Entry.
e.authors.count()
e.authors.filter(name__contains='John')

a = Author.objects.get(id=5)
a.entry_set.all() # Returns all Entry objects for this Author.
```

Like *ForeignKey*, *ManyToManyField* can specify *related_name*. In the above example, if the *ManyToManyField* in `Entry` had specified `related_name='entries'`, then each `Author` instance would have an `entries` attribute instead of `entry_set`.

One-to-one relationships

One-to-one relationships are very similar to many-to-one relationships. If you define a *OneToOneField* on your model, instances of that model will have access to the related object via a simple attribute of the model.

For example:

```
class EntryDetail(models.Model):
    entry = models.OneToOneField(Entry, on_delete=models.CASCADE)
    details = models.TextField()

ed = EntryDetail.objects.get(id=2)
ed.entry # Returns the related Entry object.
```

The difference comes in "reverse" queries. The related model in a one-to-one relationship also has access to a *Manager* object, but that *Manager* represents a single object, rather than a collection of objects:

```
e = Entry.objects.get(id=2)
e.entrydetail # returns the related EntryDetail object
```

If no object has been assigned to this relationship, Django will raise a `DoesNotExist` exception.

Instances can be assigned to the reverse relationship in the same way as you would assign the forward relationship:

```
e.entrydetail = ed
```

How are the backward relationships possible?

Other object-relational mappers require you to define relationships on both sides. The Django developers believe this is a violation of the DRY (Don't Repeat Yourself) principle, so Django only requires you to define the relationship on one end.

But how is this possible, given that a model class doesn't know which other model classes are related to it until those other model classes are loaded?

The answer lies in the *app registry*. When Django starts, it imports each application listed in *INSTALLED_APPS*, and then the `models` module inside each application. Whenever a new model class is created, Django adds backward-relationships to any related models. If the related models haven't been imported yet, Django keeps tracks of the relationships and adds them when the related models eventually are imported.

For this reason, it's particularly important that all the models you're using be defined in applications listed in *INSTALLED_APPS*. Otherwise, backwards relations may not work properly.

Queries over related objects

Queries involving related objects follow the same rules as queries involving normal value fields. When specifying the value for a query to match, you may use either an object instance itself, or the primary key value for the object.

For example, if you have a Blog object `b` with `id=5`, the following three queries would be identical:

```
Entry.objects.filter(blog=b) # Query using object instance
Entry.objects.filter(blog=b.id) # Query using id from instance
Entry.objects.filter(blog=5) # Query using id directly
```

Falling back to raw SQL

If you find yourself needing to write an SQL query that is too complex for Django's database-mapper to handle, you can fall back on writing SQL by hand. Django has a couple of options for writing raw SQL queries; see Performing raw SQL queries.

Finally, it's important to note that the Django database layer is merely an interface to your database. You can access your database via other tools, programming languages or database frameworks; there's nothing Django-specific about your database.

3.2.3 Aggregation

The topic guide on Django's database-abstraction API described the way that you can use Django queries that create, retrieve, update and delete individual objects. However, sometimes you will need to retrieve values that are derived by summarizing or *aggregating* a collection of objects. This topic guide describes the ways that aggregate values can be generated and returned using Django queries.

Throughout this guide, we'll refer to the following models. These models are used to track the inventory for a series of online bookstores:

```
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()

class Publisher(models.Model):
    name = models.CharField(max_length=300)
    num_awards = models.IntegerField()

class Book(models.Model):
    name = models.CharField(max_length=300)
    pages = models.IntegerField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
    rating = models.FloatField()
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    pubdate = models.DateField()

class Store(models.Model):
    name = models.CharField(max_length=300)
    books = models.ManyToManyField(Book)
    registered_users = models.PositiveIntegerField()
```

Cheat sheet

In a hurry? Here's how to do common aggregate queries, assuming the models above:

```
# Total number of books.
>>> Book.objects.count()
2452

# Total number of books with publisher=BaloneyPress
>>> Book.objects.filter(publisher__name='BaloneyPress').count()
73

# Average price across all books.
>>> from django.db.models import Avg
>>> Book.objects.all().aggregate(Avg('price'))
{'price__avg': 34.35}

# Max price across all books.
>>> from django.db.models import Max
>>> Book.objects.all().aggregate(Max('price'))
{'price__max': Decimal('81.20')}

# Cost per page
>>> from django.db.models import F, FloatField, Sum
>>> Book.objects.all().aggregate(
...     price_per_page=Sum(F('price')/F('pages'), output_field=FloatField()))
{'price_per_page': 0.4470664529184653}

# All the following queries involve traversing the Book<->Publisher
# foreign key relationship backwards.

# Each publisher, each with a count of books as a "num_books" attribute.
>>> from django.db.models import Count
>>> pubs = Publisher.objects.annotate(num_books=Count('book'))
>>> pubs
<QuerySet [<Publisher: BaloneyPress>, <Publisher: SalamiPress>, ...]>
>>> pubs[0].num_books
73

# The top 5 publishers, in order by number of books.
>>> pubs = Publisher.objects.annotate(num_books=Count('book')).order_by('-num_books')[:5]
>>> pubs[0].num_books
1323
```

Generating aggregates over a `QuerySet`

Django provides two ways to generate aggregates. The first way is to generate summary values over an entire `QuerySet`. For example, say you wanted to calculate the average price of all books available for sale. Django's query syntax provides a means for describing the set of all books:

```
>>> Book.objects.all()
```

What we need is a way to calculate summary values over the objects that belong to this `QuerySet`. This is done by appending an `aggregate()` clause onto the `QuerySet`:

```
>>> from django.db.models import Avg
>>> Book.objects.all().aggregate(Avg('price'))
{'price__avg': 34.35}
```

The `all()` is redundant in this example, so this could be simplified to:

```
>>> Book.objects.aggregate(Avg('price'))
{'price__avg': 34.35}
```

The argument to the `aggregate()` clause describes the aggregate value that we want to compute - in this case, the average of the `price` field on the `Book` model. A list of the aggregate functions that are available can be found in the *QuerySet reference*.

`aggregate()` is a terminal clause for a `QuerySet` that, when invoked, returns a dictionary of name-value pairs. The name is an identifier for the aggregate value; the value is the computed aggregate. The name is automatically generated from the name of the field and the aggregate function. If you want to manually specify a name for the aggregate value, you can do so by providing that name when you specify the aggregate clause:

```
>>> Book.objects.aggregate(average_price=Avg('price'))
{'average_price': 34.35}
```

If you want to generate more than one aggregate, you just add another argument to the `aggregate()` clause. So, if we also wanted to know the maximum and minimum price of all books, we would issue the query:

```
>>> from django.db.models import Avg, Max, Min
>>> Book.objects.aggregate(Avg('price'), Max('price'), Min('price'))
{'price__avg': 34.35, 'price__max': Decimal('81.20'), 'price__min': Decimal('12.99')}
```

Generating aggregates for each item in a `QuerySet`

The second way to generate summary values is to generate an independent summary for each object in a `QuerySet`. For example, if you are retrieving a list of books, you may want to know how many authors contributed to each book. Each Book has a many-to-many relationship with the Author; we want to summarize this relationship for each book in the `QuerySet`.

Per-object summaries can be generated using the `annotate()` clause. When an `annotate()` clause is specified, each object in the `QuerySet` will be annotated with the specified values.

The syntax for these annotations is identical to that used for the `aggregate()` clause. Each argument to `annotate()` describes an aggregate that is to be calculated. For example, to annotate books with the number of authors:

```
# Build an annotated queryset
>>> from django.db.models import Count
>>> q = Book.objects.annotate(Count('authors'))
# Interrogate the first object in the queryset
>>> q[0]
<Book: The Definitive Guide to Django>
>>> q[0].authors__count
2
# Interrogate the second object in the queryset
>>> q[1]
<Book: Practical Django Projects>
>>> q[1].authors__count
1
```

As with `aggregate()`, the name for the annotation is automatically derived from the name of the aggregate function and the name of the field being aggregated. You can override this default name by providing an alias when you specify the annotation:

```
>>> q = Book.objects.annotate(num_authors=Count('authors'))
>>> q[0].num_authors
```

```
2
>>> q[1].num_authors
1
```

Unlike `aggregate()`, `annotate()` is *not* a terminal clause. The output of the `annotate()` clause is a `QuerySet`; this `QuerySet` can be modified using any other `QuerySet` operation, including `filter()`, `order_by()`, or even additional calls to `annotate()`.

Combining multiple aggregations

Combining multiple aggregations with `annotate()` will yield the wrong results because joins are used instead of subqueries:

```
>>> book = Book.objects.first()
>>> book.authors.count()
2
>>> book.store_set.count()
3
>>> q = Book.objects.annotate(Count('authors'), Count('store'))
>>> q[0].authors__count
6
>>> q[0].store__count
6
```

For most aggregates, there is no way to avoid this problem, however, the *`Count`* aggregate has a `distinct` parameter that may help:

```
>>> q = Book.objects.annotate(Count('authors', distinct=True), Count('store', distinct=True))
>>> q[0].authors__count
2
>>> q[0].store__count
3
```

If in doubt, inspect the SQL query!

In order to understand what happens in your query, consider inspecting the `query` property of your `QuerySet`.

Joins and aggregates

So far, we have dealt with aggregates over fields that belong to the model being queried. However, sometimes the value you want to aggregate will belong to a model that is related to the model you are querying.

When specifying the field to be aggregated in an aggregate function, Django will allow you to use the same *double underscore notation* that is used when referring to related fields in filters. Django will then handle any table joins that are required to retrieve and aggregate the related value.

For example, to find the price range of books offered in each store, you could use the annotation:

```
>>> from django.db.models import Max, Min
>>> Store.objects.annotate(min_price=Min('books__price'), max_price=Max('books__price'))
```

This tells Django to retrieve the `Store` model, join (through the many-to-many relationship) with the `Book` model, and aggregate on the price field of the book model to produce a minimum and maximum value.

The same rules apply to the `aggregate()` clause. If you wanted to know the lowest and highest price of any book that is available for sale in any of the stores, you could use the aggregate:

```
>>> Store.objects.aggregate(min_price=Min('books__price'), max_price=Max('books__price'))
```

Join chains can be as deep as you require. For example, to extract the age of the youngest author of any book available for sale, you could issue the query:

```
>>> Store.objects.aggregate(youngest_age=Min('books__authors__age'))
```

Following relationships backwards

In a way similar to *Lookups that span relationships*, aggregations and annotations on fields of models or models that are related to the one you are querying can include traversing "reverse" relationships. The lowercase name of related models and double-underscores are used here too.

For example, we can ask for all publishers, annotated with their respective total book stock counters (note how we use `'book'` to specify the `Publisher` -> `Book` reverse foreign key hop):

```
>>> from django.db.models import Count, Min, Sum, Avg
>>> Publisher.objects.annotate(Count('book'))
```

(Every `Publisher` in the resulting `QuerySet` will have an extra attribute called `book__count`.)

We can also ask for the oldest book of any of those managed by every publisher:

```
>>> Publisher.objects.aggregate(oldest_pubdate=Min('book__pubdate'))
```

(The resulting dictionary will have a key called `'oldest_pubdate'`. If no such alias were specified, it would be the rather long `'book__pubdate__min'`.)

This doesn't apply just to foreign keys. It also works with many-to-many relations. For example, we can ask for every author, annotated with the total number of pages considering all the books the author has (co-)authored (note how we use `'book'` to specify the `Author` -> `Book` reverse many-to-many hop):

```
>>> Author.objects.annotate(total_pages=Sum('book__pages'))
```

(Every `Author` in the resulting `QuerySet` will have an extra attribute called `total_pages`. If no such alias were specified, it would be the rather long `book__pages__sum`.)

Or ask for the average rating of all the books written by author(s) we have on file:

```
>>> Author.objects.aggregate(average_rating=Avg('book__rating'))
```

(The resulting dictionary will have a key called `'average_rating'`. If no such alias were specified, it would be the rather long `'book__rating__avg'`.)

Aggregations and other `QuerySet` clauses

`filter()` and `exclude()`

Aggregates can also participate in filters. Any `filter()` (or `exclude()`) applied to normal model fields will have the effect of constraining the objects that are considered for aggregation.

When used with an `annotate()` clause, a filter has the effect of constraining the objects for which an annotation is calculated. For example, you can generate an annotated list of all books that have a title starting with "Django" using the query:

```
>>> from django.db.models import Count, Avg
>>> Book.objects.filter(name__startswith="Django").annotate(num_authors=Count('authors'))
```

When used with an `aggregate()` clause, a filter has the effect of constraining the objects over which the aggregate is calculated. For example, you can generate the average price of all books with a title that starts with "Django" using the query:

```
>>> Book.objects.filter(name__startswith="Django").aggregate(Avg('price'))
```

Filtering on annotations Annotated values can also be filtered. The alias for the annotation can be used in `filter()` and `exclude()` clauses in the same way as any other model field.

For example, to generate a list of books that have more than one author, you can issue the query:

```
>>> Book.objects.annotate(num_authors=Count('authors')).filter(num_authors__gt=1)
```

This query generates an annotated result set, and then generates a filter based upon that annotation.

Order of `annotate()` and `filter()` clauses When developing a complex query that involves both `annotate()` and `filter()` clauses, pay particular attention to the order in which the clauses are applied to the `QuerySet`.

When an `annotate()` clause is applied to a query, the annotation is computed over the state of the query up to the point where the annotation is requested. The practical implication of this is that `filter()` and `annotate()` are not commutative operations.

Given:

- Publisher A has two books with ratings 4 and 5.
- Publisher B has two books with ratings 1 and 4.
- Publisher C has one book with rating 1.

Here's an example with the `Count` aggregate:

```
>>> a, b = Publisher.objects.annotate(num_books=Count('book', distinct=True)).filter(book__rating__gt
>>> a, a.num_books
(<Publisher: A>, 2)
>>> b, b.num_books
(<Publisher: B>, 2)

>>> a, b = Publisher.objects.filter(book__rating__gt=3.0).annotate(num_books=Count('book'))
>>> a, a.num_books
(<Publisher: A>, 2)
>>> b, b.num_books
(<Publisher: B>, 1)
```

Both queries return a list of publishers that have at least one book with a rating exceeding 3.0, hence publisher C is excluded.

In the first query, the annotation precedes the filter, so the filter has no effect on the annotation. `distinct=True` is required to avoid a *query bug*.

The second query counts the number of books that have a rating exceeding 3.0 for each publisher. The filter precedes the annotation, so the filter constrains the objects considered when calculating the annotation.

Here's another example with the `Avg` aggregate:

```
>>> a, b = Publisher.objects.annotate(avg_rating=Avg('book__rating')).filter(book__rating__gt=3.0)
>>> a, a.avg_rating
(<Publisher: A>, 4.5)  # (5+4)/2
>>> b, b.avg_rating
```

```
(<Publisher: B>, 2.5)  # (1+4)/2

>>> a, b = Publisher.objects.filter(book__rating__gt=3.0).annotate(avg_rating=Avg('book__rating'))
>>> a, a.avg_rating
(<Publisher: A>, 4.5)  # (5+4)/2
>>> b, b.avg_rating
(<Publisher: B>, 4.0)  # 4/1 (book with rating 1 excluded)
```

The first query asks for the average rating of all a publisher's books for publisher's that have at least one book with a rating exceeding 3.0. The second query asks for the average of a publisher's book's ratings for only those ratings exceeding 3.0.

It's difficult to intuit how the ORM will translate complex querysets into SQL queries so when in doubt, inspect the SQL with `str(queryset.query)` and write plenty of tests.

order_by()

Annotations can be used as a basis for ordering. When you define an `order_by()` clause, the aggregates you provide can reference any alias defined as part of an `annotate()` clause in the query.

For example, to order a `QuerySet` of books by the number of authors that have contributed to the book, you could use the following query:

```
>>> Book.objects.annotate(num_authors=Count('authors')).order_by('num_authors')
```

values()

Ordinarily, annotations are generated on a per-object basis - an annotated `QuerySet` will return one result for each object in the original `QuerySet`. However, when a `values()` clause is used to constrain the columns that are returned in the result set, the method for evaluating annotations is slightly different. Instead of returning an annotated result for each result in the original `QuerySet`, the original results are grouped according to the unique combinations of the fields specified in the `values()` clause. An annotation is then provided for each unique group; the annotation is computed over all members of the group.

For example, consider an author query that attempts to find out the average rating of books written by each author:

```
>>> Author.objects.annotate(average_rating=Avg('book__rating'))
```

This will return one result for each author in the database, annotated with their average book rating.

However, the result will be slightly different if you use a `values()` clause:

```
>>> Author.objects.values('name').annotate(average_rating=Avg('book__rating'))
```

In this example, the authors will be grouped by name, so you will only get an annotated result for each *unique* author name. This means if you have two authors with the same name, their results will be merged into a single result in the output of the query; the average will be computed as the average over the books written by both authors.

Order of `annotate()` and `values()` clauses As with the `filter()` clause, the order in which `annotate()` and `values()` clauses are applied to a query is significant. If the `values()` clause precedes the `annotate()`, the annotation will be computed using the grouping described by the `values()` clause.

However, if the `annotate()` clause precedes the `values()` clause, the annotations will be generated over the entire query set. In this case, the `values()` clause only constrains the fields that are generated on output.

For example, if we reverse the order of the `values()` and `annotate()` clause from our previous example:

```
>>> Author.objects.annotate(average_rating=Avg('book__rating')).values('name', 'average_rating')
```

This will now yield one unique result for each author; however, only the author's name and the `average_rating` annotation will be returned in the output data.

You should also note that `average_rating` has been explicitly included in the list of values to be returned. This is required because of the ordering of the `values()` and `annotate()` clause.

If the `values()` clause precedes the `annotate()` clause, any annotations will be automatically added to the result set. However, if the `values()` clause is applied after the `annotate()` clause, you need to explicitly include the aggregate column.

Interaction with default ordering or `order_by()` Fields that are mentioned in the `order_by()` part of a queryset (or which are used in the default ordering on a model) are used when selecting the output data, even if they are not otherwise specified in the `values()` call. These extra fields are used to group "like" results together and they can make otherwise identical result rows appear to be separate. This shows up, particularly, when counting things.

By way of example, suppose you have a model like this:

```
from django.db import models

class Item(models.Model):
    name = models.CharField(max_length=10)
    data = models.IntegerField()

    class Meta:
        ordering = ["name"]
```

The important part here is the default ordering on the `name` field. If you want to count how many times each distinct `data` value appears, you might try this:

```
# Warning: not quite correct!
Item.objects.values("data").annotate(Count("id"))
```

...which will group the `Item` objects by their common `data` values and then count the number of `id` values in each group. Except that it won't quite work. The default ordering by `name` will also play a part in the grouping, so this query will group by distinct `(data, name)` pairs, which isn't what you want. Instead, you should construct this queryset:

```
Item.objects.values("data").annotate(Count("id")).order_by()
```

...clearing any ordering in the query. You could also order by, say, `data` without any harmful effects, since that is already playing a role in the query.

This behavior is the same as that noted in the queryset documentation for *`distinct()`* and the general rule is the same: normally you won't want extra columns playing a part in the result, so clear out the ordering, or at least make sure it's restricted only to those fields you also select in a `values()` call.

: You might reasonably ask why Django doesn't remove the extraneous columns for you. The main reason is consistency with `distinct()` and other places: Django **never** removes ordering constraints that you have specified (and we can't change those other methods' behavior, as that would violate our API stability policy).

Aggregating annotations

You can also generate an aggregate on the result of an annotation. When you define an `aggregate()` clause, the aggregates you provide can reference any alias defined as part of an `annotate()` clause in the query.

For example, if you wanted to calculate the average number of authors per book you first annotate the set of books with the author count, then aggregate that author count, referencing the annotation field:

```
>>> from django.db.models import Count, Avg
>>> Book.objects.annotate(num_authors=Count('authors')).aggregate(Avg('num_authors'))
{'num_authors__avg': 1.66}
```

3.2.4 Search

A common task for web applications is to search some data in the database with user input. In a simple case, this could be filtering a list of objects by a category. A more complex use case might require searching with weighting, categorization, highlighting, multiple languages, and so on. This document explains some of the possible use cases and the tools you can use.

We'll refer to the same models used in Making queries.

Use Cases

Standard textual queries

Text-based fields have a selection of simple matching operations. For example, you may wish to allow lookup up an author like so:

```
>>> Author.objects.filter(name__contains='Terry')
[<Author: Terry Gilliam>, <Author: Terry Jones>]
```

This is a very fragile solution as it requires the user to know an exact substring of the author's name. A better approach could be a case-insensitive match (*`icontains`*), but this is only marginally better.

A database's more advanced comparison functions

If you're using PostgreSQL, Django provides a selection of database specific tools to allow you to leverage more complex querying options. Other databases have different selections of tools, possibly via plugins or user-defined functions. Django doesn't include any support for them at this time. We'll use some examples from PostgreSQL to demonstrate the kind of functionality databases may have.

Searching in other databases

All of the searching tools provided by *`django.contrib.postgres`* are constructed entirely on public APIs such as custom lookups and database functions. Depending on your database, you should be able to construct queries to allow similar APIs. If there are specific things which cannot be achieved this way, please open a ticket.

In the above example, we determined that a case insensitive lookup would be more useful. When dealing with non-English names, a further improvement is to use *`unaccented comparison`*:

```
>>> Author.objects.filter(name__unaccent__icontains='Helen')
[<Author: Helen Mirren>, <Author: Helena Bonham Carter>, <Actor: Hélène Joy>]
```

This shows another issue, where we are matching against a different spelling of the name. In this case we have an asymmetry though - a search for `Helen` will pick up `Helena` or `Hélène`, but not the reverse. Another option would be to use a *`trigram_similar`* comparison, which compares sequences of letters.

For example:

```
>>> Author.objects.filter(name__unaccent__lower__trigram_similar='Hélène')
[<Author: Helen Mirren>, <Actor: Hélène Joy>]
```

Now we have a different problem - the longer name of "Helena Bonham Carter" doesn't show up as it is much longer. Trigram searches consider all combinations of three letters, and compares how many appear in both search and source strings. For the longer name, there are more combinations which appear in the source string so it is no longer considered a close match.

The correct choice of comparison functions here depends on your particular data set, for example the language(s) used and the type of text being searched. All of the examples we've seen are on short strings where the user is likely to enter something close (by varying definitions) to the source data.

Document-based search

Simple database operations are too simple an approach when you start considering large blocks of text. Whereas the examples above can be thought of as operations on a string of characters, full text search looks at the actual words. Depending on the system used, it's likely to use some of the following ideas:

- Ignoring "stop words" such as "a", "the", "and".
- Stemming words, so that "pony" and "ponies" are considered similar.
- Weighting words based on different criteria such as how frequently they appear in the text, or the importance of the fields, such as the title or keywords, that they appear in.

There are many alternatives for using searching software, some of the most prominent are Elastic and Solr. These are full document-based search solutions. To use them with data from Django models, you'll need a layer which translates your data into a textual document, including back-references to the database ids. When a search using the engine returns a certain document, you can then look it up in the database. There are a variety of third-party libraries which are designed to help with this process.

PostgreSQL support PostgreSQL has its own full text search implementation built-in. While not as powerful as some other search engines, it has the advantage of being inside your database and so can easily be combined with other relational queries such as categorization.

The *`django.contrib.postgres`* module provides some helpers to make these queries. For example, a simple query might be to select all the blog entries which mention "cheese":

```
>>> Entry.objects.filter(body_text__search='cheese')
[<Entry: Cheese on Toast recipes>, <Entry: Pizza recipes>]
```

You can also filter on a combination of fields and on related models:

```
>>> Entry.objects.annotate(
...     search=SearchVector('blog__tagline', 'body_text'),
... ).filter(search='cheese')
[
    <Entry: Cheese on Toast recipes>,
    <Entry: Pizza Recipes>,
    <Entry: Dairy farming in Argentina>,
]
```

See the `contrib.postgres` Full text search document for complete details.

3.2.5 Managers

class Manager

A `Manager` is the interface through which database query operations are provided to Django models. At least one `Manager` exists for every model in a Django application.

The way `Manager` classes work is documented in Making queries; this document specifically touches on model options that customize `Manager` behavior.

Manager names

By default, Django adds a `Manager` with the name `objects` to every Django model class. However, if you want to use `objects` as a field name, or if you want to use a name other than `objects` for the `Manager`, you can rename it on a per-model basis. To rename the `Manager` for a given class, define a class attribute of type `models.Manager()` on that model. For example:

```
from django.db import models

class Person(models.Model):
    #...
    people = models.Manager()
```

Using this example model, `Person.objects` will generate an `AttributeError` exception, but `Person.people.all()` will provide a list of all `Person` objects.

Custom managers

You can use a custom `Manager` in a particular model by extending the base `Manager` class and instantiating your custom `Manager` in your model.

There are two reasons you might want to customize a `Manager`: to add extra `Manager` methods, and/or to modify the initial `QuerySet` the `Manager` returns.

Adding extra manager methods

Adding extra `Manager` methods is the preferred way to add "table-level" functionality to your models. (For "row-level" functionality – i.e., functions that act on a single instance of a model object – use *Model methods*, not custom `Manager` methods.)

A custom `Manager` method can return anything you want. It doesn't have to return a `QuerySet`.

For example, this custom `Manager` offers a method `with_counts()`, which returns a list of all `OpinionPoll` objects, each with an extra `num_responses` attribute that is the result of an aggregate query:

```
from django.db import models

class PollManager(models.Manager):
    def with_counts(self):
        from django.db import connection
        with connection.cursor() as cursor:
            cursor.execute("""
                SELECT p.id, p.question, p.poll_date, COUNT(*)
                FROM polls_opinionpoll p, polls_response r
                WHERE p.id = r.poll_id
                GROUP BY p.id, p.question, p.poll_date
```

```
                ORDER BY p.poll_date DESC""")
            result_list = []
            for row in cursor.fetchall():
                p = self.model(id=row[0], question=row[1], poll_date=row[2])
                p.num_responses = row[3]
                result_list.append(p)
        return result_list

class OpinionPoll(models.Model):
    question = models.CharField(max_length=200)
    poll_date = models.DateField()
    objects = PollManager()

class Response(models.Model):
    poll = models.ForeignKey(OpinionPoll, on_delete=models.CASCADE)
    person_name = models.CharField(max_length=50)
    response = models.TextField()
```

With this example, you'd use `OpinionPoll.objects.with_counts()` to return that list of `OpinionPoll` objects with `num_responses` attributes.

Another thing to note about this example is that `Manager` methods can access `self.model` to get the model class to which they're attached.

Modifying a manager's initial `QuerySet`

A `Manager`'s base `QuerySet` returns all objects in the system. For example, using this model:

```
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)
```

...the statement `Book.objects.all()` will return all books in the database.

You can override a `Manager`'s base `QuerySet` by overriding the `Manager.get_queryset()` method. `get_queryset()` should return a `QuerySet` with the properties you require.

For example, the following model has *two* `Managers` – one that returns all objects, and one that returns only the books by Roald Dahl:

```
# First, define the Manager subclass.
class DahlBookManager(models.Manager):
    def get_queryset(self):
        return super(DahlBookManager, self).get_queryset().filter(author='Roald Dahl')

# Then hook it into the Book model explicitly.
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)

    objects = models.Manager() # The default manager.
    dahl_objects = DahlBookManager() # The Dahl-specific manager.
```

With this sample model, `Book.objects.all()` will return all books in the database, but `Book.dahl_objects.all()` will only return the ones written by Roald Dahl.

Of course, because `get_queryset()` returns a `QuerySet` object, you can use `filter()`, `exclude()` and all the other `QuerySet` methods on it. So these statements are all legal:

```
Book.dahl_objects.all()
Book.dahl_objects.filter(title='Matilda')
Book.dahl_objects.count()
```

This example also pointed out another interesting technique: using multiple managers on the same model. You can attach as many `Manager()` instances to a model as you'd like. This is an easy way to define common "filters" for your models.

For example:

```
class AuthorManager(models.Manager):
    def get_queryset(self):
        return super(AuthorManager, self).get_queryset().filter(role='A')

class EditorManager(models.Manager):
    def get_queryset(self):
        return super(EditorManager, self).get_queryset().filter(role='E')

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    role = models.CharField(max_length=1, choices=(('A', _('Author')), ('E', _('Editor'))))
    people = models.Manager()
    authors = AuthorManager()
    editors = EditorManager()
```

This example allows you to request `Person.authors.all()`, `Person.editors.all()`, and `Person.people.all()`, yielding predictable results.

Default managers

`Model._default_manager`

If you use custom `Manager` objects, take note that the first `Manager` Django encounters (in the order in which they're defined in the model) has a special status. Django interprets the first `Manager` defined in a class as the "default" `Manager`, and several parts of Django (including *dumpdata*) will use that `Manager` exclusively for that model. As a result, it's a good idea to be careful in your choice of default manager in order to avoid a situation where overriding `get_queryset()` results in an inability to retrieve objects you'd like to work with.

You can specify a custom default manager using *Meta.default_manager_name*.

If you're writing some code that must handle an unknown model, for example, in a third-party app that implements a generic view, use this manager (or *_base_manager*) rather than assuming the model has an `objects` manager.

Base managers

`Model._base_manager`

Using managers for related object access By default, Django uses an instance of the `Model._base_manager` manager class when accessing related objects (i.e. `choice.question`), not the `_default_manager` on the related object. This is because Django needs to be able to retrieve the related object, even if it would otherwise be filtered out (and hence be inaccessible) by the default manager.

If the normal base manager class (*django.db.models.Manager*) isn't appropriate for your circumstances, you can tell Django which class to use by setting *Meta.base_manager_name*.

Manager's aren't used when querying on related models. For example, if the Question model *from the tutorial* had a deleted field and a base manager that filters out instances with deleted=True, a queryset like Choice.objects.filter(question__name__startswith='What') would include choices related to deleted questions.

Don't filter away any results in this type of manager subclass This manager is used to access objects that are related to from some other model. In those situations, Django has to be able to see all the objects for the model it is fetching, so that *anything* which is referred to can be retrieved.

If you override the get_queryset() method and filter out any rows, Django will return incorrect results. Don't do that. A manager that filters results in get_queryset() is not appropriate for use as a default manager.

Calling custom QuerySet methods from the manager

While most methods from the standard QuerySet are accessible directly from the Manager, this is only the case for the extra methods defined on a custom QuerySet if you also implement them on the Manager:

```
class PersonQuerySet(models.QuerySet):
    def authors(self):
        return self.filter(role='A')

    def editors(self):
        return self.filter(role='E')

class PersonManager(models.Manager):
    def get_queryset(self):
        return PersonQuerySet(self.model, using=self._db)

    def authors(self):
        return self.get_queryset().authors()

    def editors(self):
        return self.get_queryset().editors()

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    role = models.CharField(max_length=1, choices=(('A', _('Author')), ('E', _('Editor'))))
    people = PersonManager()
```

This example allows you to call both authors() and editors() directly from the manager Person.people.

Creating a manager with QuerySet methods

In lieu of the above approach which requires duplicating methods on both the QuerySet and the Manager, *QuerySet.as_manager()* can be used to create an instance of Manager with a copy of a custom QuerySet's methods:

```
class Person(models.Model):
    ...
    people = PersonQuerySet.as_manager()
```

The `Manager` instance created by *`QuerySet.as_manager()`* will be virtually identical to the `PersonManager` from the previous example.

Not every `QuerySet` method makes sense at the `Manager` level; for instance we intentionally prevent the *`QuerySet.delete()`* method from being copied onto the `Manager` class.

Methods are copied according to the following rules:

- Public methods are copied by default.
- Private methods (starting with an underscore) are not copied by default.
- Methods with a `queryset_only` attribute set to `False` are always copied.
- Methods with a `queryset_only` attribute set to `True` are never copied.

For example:

```
class CustomQuerySet(models.QuerySet):
    # Available on both Manager and QuerySet.
    def public_method(self):
        return

    # Available only on QuerySet.
    def _private_method(self):
        return

    # Available only on QuerySet.
    def opted_out_public_method(self):
        return
    opted_out_public_method.queryset_only = True

    # Available on both Manager and QuerySet.
    def _opted_in_private_method(self):
        return
    _opted_in_private_method.queryset_only = False
```

`from_queryset()`
classmethod `from_queryset`(*queryset_class*)
For advanced usage you might want both a custom `Manager` and a custom `QuerySet`. You can do that by calling `Manager.from_queryset()` which returns a *subclass* of your base `Manager` with a copy of the custom `QuerySet` methods:

```
class BaseManager(models.Manager):
    def manager_only_method(self):
        return

class CustomQuerySet(models.QuerySet):
    def manager_and_queryset_method(self):
        return

class MyModel(models.Model):
    objects = BaseManager.from_queryset(CustomQuerySet)()
```

You may also store the generated class into a variable:

```
CustomManager = BaseManager.from_queryset(CustomQuerySet)

class MyModel(models.Model):
    objects = CustomManager()
```

Custom managers and model inheritance

Here's how Django handles custom managers and *model inheritance*:

1. Managers from base classes are always inherited by the child class, using Python's normal name resolution order (names on the child class override all others; then come names on the first parent class, and so on).
2. If no managers are declared on a model and/or its parents, Django automatically creates the `objects` manager.
3. The default manager on a class is either the one chosen with *`Meta.default_manager_name`*, or the first manager declared on the model, or the default manager of the first parent model.

In older versions, manager inheritance varied depending on the type of model inheritance (i.e. *Abstract base classes*, *Multi-table inheritance*, or *Proxy models*), especially with regards to electing the default manager.

These rules provide the necessary flexibility if you want to install a collection of custom managers on a group of models, via an abstract base class, but still customize the default manager. For example, suppose you have this base class:

```
class AbstractBase(models.Model):
    # ...
    objects = CustomManager()

    class Meta:
        abstract = True
```

If you use this directly in a subclass, `objects` will be the default manager if you declare no managers in the base class:

```
class ChildA(AbstractBase):
    # ...
    # This class has CustomManager as the default manager.
    pass
```

If you want to inherit from `AbstractBase`, but provide a different default manager, you can provide the default manager on the child class:

```
class ChildB(AbstractBase):
    # ...
    # An explicit default manager.
    default_manager = OtherManager()
```

Here, `default_manager` is the default. The `objects` manager is still available, since it's inherited. It just isn't used as the default.

Finally for this example, suppose you want to add extra managers to the child class, but still use the default from `AbstractBase`. You can't add the new manager directly in the child class, as that would override the default and you would have to also explicitly include all the managers from the abstract base class. The solution is to put the extra managers in another base class and introduce it into the inheritance hierarchy *after* the defaults:

```
class ExtraManager(models.Model):
    extra_manager = OtherManager()

    class Meta:
        abstract = True

class ChildC(AbstractBase, ExtraManager):
    # ...
    # Default manager is CustomManager, but OtherManager is
    # also available via the "extra_manager" attribute.
    pass
```

Note that while you can *define* a custom manager on the abstract model, you can't *invoke* any methods using the abstract model. That is:

```
ClassA.objects.do_something()
```

is legal, but:

```
AbstractBase.objects.do_something()
```

will raise an exception. This is because managers are intended to encapsulate logic for managing collections of objects. Since you can't have a collection of abstract objects, it doesn't make sense to be managing them. If you have functionality that applies to the abstract model, you should put that functionality in a `staticmethod` or `classmethod` on the abstract model.

Implementation concerns

Whatever features you add to your custom `Manager`, it must be possible to make a shallow copy of a `Manager` instance; i.e., the following code must work:

```
>>> import copy
>>> manager = MyManager()
>>> my_copy = copy.copy(manager)
```

Django makes shallow copies of manager objects during certain queries; if your Manager cannot be copied, those queries will fail.

This won't be an issue for most custom managers. If you are just adding simple methods to your `Manager`, it is unlikely that you will inadvertently make instances of your `Manager` uncopyable. However, if you're overriding `__getattr__` or some other private method of your `Manager` object that controls object state, you should ensure that you don't affect the ability of your `Manager` to be copied.

3.2.6 Performing raw SQL queries

When the model query APIs don't go far enough, you can fall back to writing raw SQL. Django gives you two ways of performing raw SQL queries: you can use *`Manager.raw()`* to *perform raw queries and return model instances*, or you can avoid the model layer entirely and *execute custom SQL directly*.

: You should be very careful whenever you write raw SQL. Every time you use it, you should properly escape any parameters that the user can control by using `params` in order to protect against SQL injection attacks. Please read more about *SQL injection protection*.

Performing raw queries

The `raw()` manager method can be used to perform raw SQL queries that return model instances:

`Manager.`**`raw`**(*raw_query*, *params=None*, *translations=None*)

This method takes a raw SQL query, executes it, and returns a `django.db.models.query.RawQuerySet` instance. This `RawQuerySet` instance can be iterated over just like a normal *`QuerySet`* to provide object instances.

This is best illustrated with an example. Suppose you have the following model:

```
class Person(models.Model):
    first_name = models.CharField(...)
    last_name = models.CharField(...)
    birth_date = models.DateField(...)
```

You could then execute custom SQL like so:

```
>>> for p in Person.objects.raw('SELECT * FROM myapp_person'):
...     print(p)
John Smith
Jane Jones
```

Of course, this example isn't very exciting – it's exactly the same as running `Person.objects.all()`. However, `raw()` has a bunch of other options that make it very powerful.

Model table names

Where did the name of the `Person` table come from in that example?

By default, Django figures out a database table name by joining the model's "app label" – the name you used in `manage.py startapp` – to the model's class name, with an underscore between them. In the example we've assumed that the `Person` model lives in an app named `myapp`, so its table would be `myapp_person`.

For more details check out the documentation for the *`db_table`* option, which also lets you manually set the database table name.

: No checking is done on the SQL statement that is passed in to `.raw()`. Django expects that the statement will return a set of rows from the database, but does nothing to enforce that. If the query does not return rows, a (possibly cryptic) error will result.

: If you are performing queries on MySQL, note that MySQL's silent type coercion may cause unexpected results when mixing types. If you query on a string type column, but with an integer value, MySQL will coerce the types of all values in the table to an integer before performing the comparison. For example, if your table contains the values `'abc'`, `'def'` and you query for `WHERE mycolumn=0`, both rows will match. To prevent this, perform the correct typecasting before using the value in a query.

: While a `RawQuerySet` instance can be iterated over like a normal *`QuerySet`*, `RawQuerySet` doesn't implement all methods you can use with `QuerySet`. For example, `__bool__()` and `__len__()` are not defined in `RawQuerySet`, and thus all `RawQuerySet` instances are considered `True`. The reason these methods are not implemented in `RawQuerySet` is that implementing them without internal caching would be a performance drawback and adding such caching would be backward incompatible.

Mapping query fields to model fields

`raw()` automatically maps fields in the query to fields on the model.

The order of fields in your query doesn't matter. In other words, both of the following queries work identically:

```
>>> Person.objects.raw('SELECT id, first_name, last_name, birth_date FROM myapp_person')
...
>>> Person.objects.raw('SELECT last_name, birth_date, first_name, id FROM myapp_person')
...
```

Matching is done by name. This means that you can use SQL's `AS` clauses to map fields in the query to model fields. So if you had some other table that had `Person` data in it, you could easily map it into `Person` instances:

```
>>> Person.objects.raw('''SELECT first AS first_name,
...                              last AS last_name,
...                              bd AS birth_date,
...                              pk AS id,
...                       FROM some_other_table''')
```

As long as the names match, the model instances will be created correctly.

Alternatively, you can map fields in the query to model fields using the `translations` argument to `raw()`. This is a dictionary mapping names of fields in the query to names of fields on the model. For example, the above query could also be written:

```
>>> name_map = {'first': 'first_name', 'last': 'last_name', 'bd': 'birth_date', 'pk': 'id'}
>>> Person.objects.raw('SELECT * FROM some_other_table', translations=name_map)
```

Index lookups

`raw()` supports indexing, so if you need only the first result you can write:

```
>>> first_person = Person.objects.raw('SELECT * FROM myapp_person')[0]
```

However, the indexing and slicing are not performed at the database level. If you have a large number of `Person` objects in your database, it is more efficient to limit the query at the SQL level:

```
>>> first_person = Person.objects.raw('SELECT * FROM myapp_person LIMIT 1')[0]
```

Deferring model fields

Fields may also be left out:

```
>>> people = Person.objects.raw('SELECT id, first_name FROM myapp_person')
```

The `Person` objects returned by this query will be deferred model instances (see *`defer()`*). This means that the fields that are omitted from the query will be loaded on demand. For example:

```
>>> for p in Person.objects.raw('SELECT id, first_name FROM myapp_person'):
...     print(p.first_name, # This will be retrieved by the original query
...           p.last_name) # This will be retrieved on demand
...
John Smith
Jane Jones
```

From outward appearances, this looks like the query has retrieved both the first name and last name. However, this example actually issued 3 queries. Only the first names were retrieved by the raw() query – the last names were both retrieved on demand when they were printed.

There is only one field that you can't leave out - the primary key field. Django uses the primary key to identify model instances, so it must always be included in a raw query. An `InvalidQuery` exception will be raised if you forget to include the primary key.

Adding annotations

You can also execute queries containing fields that aren't defined on the model. For example, we could use PostgreSQL's age() function to get a list of people with their ages calculated by the database:

```
>>> people = Person.objects.raw('SELECT *, age(birth_date) AS age FROM myapp_person')
>>> for p in people:
...     print("%s is %s." % (p.first_name, p.age))
John is 37.
Jane is 42.
...
```

Passing parameters into `raw()`

If you need to perform parameterized queries, you can use the `params` argument to `raw()`:

```
>>> lname = 'Doe'
>>> Person.objects.raw('SELECT * FROM myapp_person WHERE last_name = %s', [lname])
```

`params` is a list or dictionary of parameters. You'll use `%s` placeholders in the query string for a list, or `%(key)s` placeholders for a dictionary (where `key` is replaced by a dictionary key, of course), regardless of your database engine. Such placeholders will be replaced with parameters from the `params` argument.

: Dictionary params are not supported with the SQLite backend; with this backend, you must pass parameters as a list.

: Do not use string formatting on raw queries!
It's tempting to write the above query as:

```
>>> query = 'SELECT * FROM myapp_person WHERE last_name = %s' % lname
>>> Person.objects.raw(query)
```

Don't.
Using the `params` argument completely protects you from SQL injection attacks, a common exploit where attackers inject arbitrary SQL into your database. If you use string interpolation, sooner or later you'll fall victim to SQL injection. As long as you remember to always use the `params` argument you'll be protected.

Executing custom SQL directly

Sometimes even *`Manager.raw()`* isn't quite enough: you might need to perform queries that don't map cleanly to models, or directly execute `UPDATE`, `INSERT`, or `DELETE` queries.

In these cases, you can always access the database directly, routing around the model layer entirely.

The object `django.db.connection` represents the default database connection. To use the database connection, call `connection.cursor()` to get a cursor object. Then, call `cursor.execute(sql, [params])` to execute the SQL and `cursor.fetchone()` or `cursor.fetchall()` to return the resulting rows.

For example:

```
from django.db import connection

def my_custom_sql(self):
    with connection.cursor() as cursor:
```

```
        cursor.execute("UPDATE bar SET foo = 1 WHERE baz = %s", [self.baz])
        cursor.execute("SELECT foo FROM bar WHERE baz = %s", [self.baz])
        row = cursor.fetchone()

    return row
```

Note that if you want to include literal percent signs in the query, you have to double them in the case you are passing parameters:

```
cursor.execute("SELECT foo FROM bar WHERE baz = '30%'")
cursor.execute("SELECT foo FROM bar WHERE baz = '30%%' AND id = %s", [self.id])
```

If you are using more than one database, you can use `django.db.connections` to obtain the connection (and cursor) for a specific database. `django.db.connections` is a dictionary-like object that allows you to retrieve a specific connection using its alias:

```
from django.db import connections
cursor = connections['my_db_alias'].cursor()
# Your code here...
```

By default, the Python DB API will return results without their field names, which means you end up with a `list` of values, rather than a `dict`. At a small performance and memory cost, you can return results as a `dict` by using something like this:

```
def dictfetchall(cursor):
    "Return all rows from a cursor as a dict"
    columns = [col[0] for col in cursor.description]
    return [
        dict(zip(columns, row))
        for row in cursor.fetchall()
    ]
```

Another option is to use `collections.namedtuple()` from the Python standard library. A `namedtuple` is a tuple-like object that has fields accessible by attribute lookup; it's also indexable and iterable. Results are immutable and accessible by field names or indices, which might be useful:

```
from collections import namedtuple

def namedtuplefetchall(cursor):
    "Return all rows from a cursor as a namedtuple"
    desc = cursor.description
    nt_result = namedtuple('Result', [col[0] for col in desc])
    return [nt_result(*row) for row in cursor.fetchall()]
```

Here is an example of the difference between the three:

```
>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2");
>>> cursor.fetchall()
((54360982, None), (54360880, None))

>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2");
>>> dictfetchall(cursor)
[{'parent_id': None, 'id': 54360982}, {'parent_id': None, 'id': 54360880}]

>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2");
>>> results = namedtuplefetchall(cursor)
>>> results
[Result(id=54360982, parent_id=None), Result(id=54360880, parent_id=None)]
>>> results[0].id
```

```
54360982
>>> results[0][0]
54360982
```

Connections and cursors

`connection` and `cursor` mostly implement the standard Python DB-API described in **PEP 249** — except when it comes to transaction handling.

If you're not familiar with the Python DB-API, note that the SQL statement in `cursor.execute()` uses placeholders, `"%s"`, rather than adding parameters directly within the SQL. If you use this technique, the underlying database library will automatically escape your parameters as necessary.

Also note that Django expects the `"%s"` placeholder, *not* the `"?"` placeholder, which is used by the SQLite Python bindings. This is for the sake of consistency and sanity.

Using a cursor as a context manager:

```
with connection.cursor() as c:
    c.execute(...)
```

is equivalent to:

```
c = connection.cursor()
try:
    c.execute(...)
finally:
    c.close()
```

3.2.7 Database transactions

Django gives you a few ways to control how database transactions are managed.

Managing database transactions

Django's default transaction behavior

Django's default behavior is to run in autocommit mode. Each query is immediately committed to the database, unless a transaction is active. *See below for details*.

Django uses transactions or savepoints automatically to guarantee the integrity of ORM operations that require multiple queries, especially *delete()* and *update()* queries.

Django's *TestCase* class also wraps each test in a transaction for performance reasons.

Tying transactions to HTTP requests

A common way to handle transactions on the web is to wrap each request in a transaction. Set *ATOMIC_REQUESTS* to `True` in the configuration of each database for which you want to enable this behavior.

It works like this. Before calling a view function, Django starts a transaction. If the response is produced without problems, Django commits the transaction. If the view produces an exception, Django rolls back the transaction.

You may perform subtransactions using savepoints in your view code, typically with the *atomic()* context manager. However, at the end of the view, either all or none of the changes will be committed.

: While the simplicity of this transaction model is appealing, it also makes it inefficient when traffic increases. Opening a transaction for every view has some overhead. The impact on performance depends on the query patterns of your application and on how well your database handles locking.

Per-request transactions and streaming responses

When a view returns a *StreamingHttpResponse*, reading the contents of the response will often execute code to generate the content. Since the view has already returned, such code runs outside of the transaction.

Generally speaking, it isn't advisable to write to the database while generating a streaming response, since there's no sensible way to handle errors after starting to send the response.

In practice, this feature simply wraps every view function in the *atomic()* decorator described below.

Note that only the execution of your view is enclosed in the transactions. Middleware runs outside of the transaction, and so does the rendering of template responses.

When *ATOMIC_REQUESTS* is enabled, it's still possible to prevent views from running in a transaction.

non_atomic_requests(*using=None*)

This decorator will negate the effect of *ATOMIC_REQUESTS* for a given view:

```
from django.db import transaction

@transaction.non_atomic_requests
def my_view(request):
    do_stuff()

@transaction.non_atomic_requests(using='other')
def my_other_view(request):
    do_stuff_on_the_other_database()
```

It only works if it's applied to the view itself.

Controlling transactions explicitly

Django provides a single API to control database transactions.

atomic(*using=None*, *savepoint=True*)

Atomicity is the defining property of database transactions. `atomic` allows us to create a block of code within which the atomicity on the database is guaranteed. If the block of code is successfully completed, the changes are committed to the database. If there is an exception, the changes are rolled back.

`atomic` blocks can be nested. In this case, when an inner block completes successfully, its effects can still be rolled back if an exception is raised in the outer block at a later point.

`atomic` is usable both as a decorator:

```
from django.db import transaction

@transaction.atomic
def viewfunc(request):
    # This code executes inside a transaction.
    do_stuff()
```

and as a context manager:

```
from django.db import transaction

def viewfunc(request):
    # This code executes in autocommit mode (Django's default).
    do_stuff()

    with transaction.atomic():
        # This code executes inside a transaction.
        do_more_stuff()
```

Wrapping `atomic` in a try/except block allows for natural handling of integrity errors:

```
from django.db import IntegrityError, transaction

@transaction.atomic
def viewfunc(request):
    create_parent()

    try:
        with transaction.atomic():
            generate_relationships()
    except IntegrityError:
        handle_exception()

    add_children()
```

In this example, even if `generate_relationships()` causes a database error by breaking an integrity constraint, you can execute queries in `add_children()`, and the changes from `create_parent()` are still there. Note that any operations attempted in `generate_relationships()` will already have been rolled back safely when `handle_exception()` is called, so the exception handler can also operate on the database if necessary.

Avoid catching exceptions inside `atomic`!

When exiting an `atomic` block, Django looks at whether it's exited normally or with an exception to determine whether to commit or roll back. If you catch and handle exceptions inside an `atomic` block, you may hide from Django the fact that a problem has happened. This can result in unexpected behavior.

This is mostly a concern for *`DatabaseError`* and its subclasses such as *`IntegrityError`*. After such an error, the transaction is broken and Django will perform a rollback at the end of the `atomic` block. If you attempt to run database queries before the rollback happens, Django will raise a *`TransactionManagementError`*. You may also encounter this behavior when an ORM-related signal handler raises an exception.

The correct way to catch database errors is around an `atomic` block as shown above. If necessary, add an extra `atomic` block for this purpose. This pattern has another advantage: it delimits explicitly which operations will be rolled back if an exception occurs.

If you catch exceptions raised by raw SQL queries, Django's behavior is unspecified and database-dependent.

In order to guarantee atomicity, `atomic` disables some APIs. Attempting to commit, roll back, or change the autocommit state of the database connection within an `atomic` block will raise an exception.

`atomic` takes a `using` argument which should be the name of a database. If this argument isn't provided, Django uses the `"default"` database.

Under the hood, Django's transaction management code:

- opens a transaction when entering the outermost `atomic` block;

- creates a savepoint when entering an inner `atomic` block;
- releases or rolls back to the savepoint when exiting an inner block;
- commits or rolls back the transaction when exiting the outermost block.

You can disable the creation of savepoints for inner blocks by setting the `savepoint` argument to `False`. If an exception occurs, Django will perform the rollback when exiting the first parent block with a savepoint if there is one, and the outermost block otherwise. Atomicity is still guaranteed by the outer transaction. This option should only be used if the overhead of savepoints is noticeable. It has the drawback of breaking the error handling described above.

You may use `atomic` when autocommit is turned off. It will only use savepoints, even for the outermost block.

Performance considerations

Open transactions have a performance cost for your database server. To minimize this overhead, keep your transactions as short as possible. This is especially important if you're using *`atomic()`* in long-running processes, outside of Django's request / response cycle.

Autocommit

Why Django uses autocommit

In the SQL standards, each SQL query starts a transaction, unless one is already active. Such transactions must then be explicitly committed or rolled back.

This isn't always convenient for application developers. To alleviate this problem, most databases provide an autocommit mode. When autocommit is turned on and no transaction is active, each SQL query gets wrapped in its own transaction. In other words, not only does each such query start a transaction, but the transaction also gets automatically committed or rolled back, depending on whether the query succeeded.

PEP 249, the Python Database API Specification v2.0, requires autocommit to be initially turned off. Django overrides this default and turns autocommit on.

To avoid this, you can *deactivate the transaction management*, but it isn't recommended.

Deactivating transaction management

You can totally disable Django's transaction management for a given database by setting *`AUTOCOMMIT`* to `False` in its configuration. If you do this, Django won't enable autocommit, and won't perform any commits. You'll get the regular behavior of the underlying database library.

This requires you to commit explicitly every transaction, even those started by Django or by third-party libraries. Thus, this is best used in situations where you want to run your own transaction-controlling middleware or do something really strange.

Performing actions after commit

Sometimes you need to perform an action related to the current database transaction, but only if the transaction successfully commits. Examples might include a Celery task, an email notification, or a cache invalidation.

Django provides the *`on_commit()`* function to register callback functions that should be executed after a transaction is successfully committed:

on_commit (*func*, *using=None*)

Pass any function (that takes no arguments) to *on_commit()*:

```
from django.db import transaction

def do_something():
    pass  # send a mail, invalidate a cache, fire off a Celery task, etc.

transaction.on_commit(do_something)
```

You can also wrap your function in a lambda:

```
transaction.on_commit(lambda: some_celery_task.delay('arg1'))
```

The function you pass in will be called immediately after a hypothetical database write made where `on_commit()` is called would be successfully committed.

If you call `on_commit()` while there isn't an active transaction, the callback will be executed immediately.

If that hypothetical database write is instead rolled back (typically when an unhandled exception is raised in an *atomic()* block), your function will be discarded and never called.

Savepoints

Savepoints (i.e. nested *atomic()* blocks) are handled correctly. That is, an *on_commit()* callable registered after a savepoint (in a nested *atomic()* block) will be called after the outer transaction is committed, but not if a rollback to that savepoint or any previous savepoint occurred during the transaction:

```
with transaction.atomic():  # Outer atomic, start a new transaction
    transaction.on_commit(foo)

    with transaction.atomic():  # Inner atomic block, create a savepoint
        transaction.on_commit(bar)

# foo() and then bar() will be called when leaving the outermost block
```

On the other hand, when a savepoint is rolled back (due to an exception being raised), the inner callable will not be called:

```
with transaction.atomic():  # Outer atomic, start a new transaction
    transaction.on_commit(foo)

    try:
        with transaction.atomic():  # Inner atomic block, create a savepoint
            transaction.on_commit(bar)
            raise SomeError()  # Raising an exception - abort the savepoint
    except SomeError:
        pass

# foo() will be called, but not bar()
```

Order of execution

On-commit functions for a given transaction are executed in the order they were registered.

Exception handling

If one on-commit function within a given transaction raises an uncaught exception, no later registered functions in that same transaction will run. This is, of course, the same behavior as if you'd executed the functions sequentially yourself without `on_commit()`.

Timing of execution

Your callbacks are executed *after* a successful commit, so a failure in a callback will not cause the transaction to roll back. They are executed conditionally upon the success of the transaction, but they are not *part* of the transaction. For the intended use cases (mail notifications, Celery tasks, etc.), this should be fine. If it's not (if your follow-up action is so critical that its failure should mean the failure of the transaction itself), then you don't want to use the `on_commit()` hook. Instead, you may want two-phase commit such as the psycopg Two-Phase Commit protocol support and the optional Two-Phase Commit Extensions in the Python DB-API specification.

Callbacks are not run until autocommit is restored on the connection following the commit (because otherwise any queries done in a callback would open an implicit transaction, preventing the connection from going back into autocommit mode).

When in autocommit mode and outside of an `atomic()` block, the function will run immediately, not on commit.

On-commit functions only work with *autocommit mode* and the `atomic()` (or `ATOMIC_REQUESTS`) transaction API. Calling `on_commit()` when autocommit is disabled and you are not within an atomic block will result in an error.

Use in tests

Django's `TestCase` class wraps each test in a transaction and rolls back that transaction after each test, in order to provide test isolation. This means that no transaction is ever actually committed, thus your `on_commit()` callbacks will never be run. If you need to test the results of an `on_commit()` callback, use a `TransactionTestCase` instead.

Why no rollback hook?

A rollback hook is harder to implement robustly than a commit hook, since a variety of things can cause an implicit rollback.

For instance, if your database connection is dropped because your process was killed without a chance to shut down gracefully, your rollback hook will never run.

The solution is simple: instead of doing something during the atomic block (transaction) and then undoing it if the transaction fails, use `on_commit()` to delay doing it in the first place until after the transaction succeeds. It's a lot easier to undo something you never did in the first place!

Low-level APIs

: Always prefer `atomic()` if possible at all. It accounts for the idiosyncrasies of each database and prevents invalid operations.
The low level APIs are only useful if you're implementing your own transaction management.

Autocommit

Django provides a straightforward API in the *django.db.transaction* module to manage the autocommit state of each database connection.

get_autocommit(*using=None*)

set_autocommit(*autocommit*, *using=None*)

These functions take a `using` argument which should be the name of a database. If it isn't provided, Django uses the `"default"` database.

Autocommit is initially turned on. If you turn it off, it's your responsibility to restore it.

Once you turn autocommit off, you get the default behavior of your database adapter, and Django won't help you. Although that behavior is specified in **PEP 249**, implementations of adapters aren't always consistent with one another. Review the documentation of the adapter you're using carefully.

You must ensure that no transaction is active, usually by issuing a *commit()* or a *rollback()*, before turning autocommit back on.

Django will refuse to turn autocommit off when an *atomic()* block is active, because that would break atomicity.

Transactions

A transaction is an atomic set of database queries. Even if your program crashes, the database guarantees that either all the changes will be applied, or none of them.

Django doesn't provide an API to start a transaction. The expected way to start a transaction is to disable autocommit with *set_autocommit()*.

Once you're in a transaction, you can choose either to apply the changes you've performed until this point with *commit()*, or to cancel them with *rollback()*. These functions are defined in *django.db.transaction*.

commit(*using=None*)

rollback(*using=None*)

These functions take a `using` argument which should be the name of a database. If it isn't provided, Django uses the `"default"` database.

Django will refuse to commit or to rollback when an *atomic()* block is active, because that would break atomicity.

Savepoints

A savepoint is a marker within a transaction that enables you to roll back part of a transaction, rather than the full transaction. Savepoints are available with the SQLite ($\geq$ 3.6.8), PostgreSQL, Oracle and MySQL (when using the InnoDB storage engine) backends. Other backends provide the savepoint functions, but they're empty operations – they don't actually do anything.

Savepoints aren't especially useful if you are using autocommit, the default behavior of Django. However, once you open a transaction with *atomic()*, you build up a series of database operations awaiting a commit or rollback. If you issue a rollback, the entire transaction is rolled back. Savepoints provide the ability to perform a fine-grained rollback, rather than the full rollback that would be performed by `transaction.rollback()`.

When the *atomic()* decorator is nested, it creates a savepoint to allow partial commit or rollback. You're strongly encouraged to use *atomic()* rather than the functions described below, but they're still part of the public API, and there's no plan to deprecate them.

Each of these functions takes a `using` argument which should be the name of a database for which the behavior applies. If no `using` argument is provided then the `"default"` database is used.

Savepoints are controlled by three functions in *`django.db.transaction`*:

`savepoint`(*using=None*)
Creates a new savepoint. This marks a point in the transaction that is known to be in a "good" state. Returns the savepoint ID (`sid`).

`savepoint_commit`(*sid*, *using=None*)
Releases savepoint `sid`. The changes performed since the savepoint was created become part of the transaction.

`savepoint_rollback`(*sid*, *using=None*)
Rolls back the transaction to savepoint `sid`.

These functions do nothing if savepoints aren't supported or if the database is in autocommit mode.

In addition, there's a utility function:

`clean_savepoints`(*using=None*)
Resets the counter used to generate unique savepoint IDs.

The following example demonstrates the use of savepoints:

```
from django.db import transaction

# open a transaction
@transaction.atomic
def viewfunc(request):

    a.save()
    # transaction now contains a.save()

    sid = transaction.savepoint()

    b.save()
    # transaction now contains a.save() and b.save()

    if want_to_keep_b:
        transaction.savepoint_commit(sid)
        # open transaction still contains a.save() and b.save()
    else:
        transaction.savepoint_rollback(sid)
        # open transaction now contains only a.save()
```

Savepoints may be used to recover from a database error by performing a partial rollback. If you're doing this inside an *`atomic()`* block, the entire block will still be rolled back, because it doesn't know you've handled the situation at a lower level! To prevent this, you can control the rollback behavior with the following functions.

`get_rollback`(*using=None*)

`set_rollback`(*rollback*, *using=None*)

Setting the rollback flag to `True` forces a rollback when exiting the innermost atomic block. This may be useful to trigger a rollback without raising an exception.

Setting it to `False` prevents such a rollback. Before doing that, make sure you've rolled back the transaction to a known-good savepoint within the current atomic block! Otherwise you're breaking atomicity and data corruption may occur.

Database-specific notes

Savepoints in SQLite

While SQLite $\geq$ 3.6.8 supports savepoints, a flaw in the design of the `sqlite3` module makes them hardly usable.

When autocommit is enabled, savepoints don't make sense. When it's disabled, `sqlite3` commits implicitly before savepoint statements. (In fact, it commits before any statement other than `SELECT`, `INSERT`, `UPDATE`, `DELETE` and `REPLACE`.) This bug has two consequences:

- The low level APIs for savepoints are only usable inside a transaction ie. inside an *`atomic()`* block.
- It's impossible to use *`atomic()`* when autocommit is turned off.

Transactions in MySQL

If you're using MySQL, your tables may or may not support transactions; it depends on your MySQL version and the table types you're using. (By "table types," we mean something like "InnoDB" or "MyISAM".) MySQL transaction peculiarities are outside the scope of this article, but the MySQL site has information on MySQL transactions.

If your MySQL setup does *not* support transactions, then Django will always function in autocommit mode: statements will be executed and committed as soon as they're called. If your MySQL setup *does* support transactions, Django will handle transactions as explained in this document.

Handling exceptions within PostgreSQL transactions

: This section is relevant only if you're implementing your own transaction management. This problem cannot occur in Django's default mode and *`atomic()`* handles it automatically.

Inside a transaction, when a call to a PostgreSQL cursor raises an exception (typically `IntegrityError`), all subsequent SQL in the same transaction will fail with the error "current transaction is aborted, queries ignored until end of transaction block". While simple use of `save()` is unlikely to raise an exception in PostgreSQL, there are more advanced usage patterns which might, such as saving objects with unique fields, saving using the force_insert/force_update flag, or invoking custom SQL.

There are several ways to recover from this sort of error.

Transaction rollback The first option is to roll back the entire transaction. For example:

```
a.save() # Succeeds, but may be undone by transaction rollback
try:
    b.save() # Could throw exception
except IntegrityError:
    transaction.rollback()
c.save() # Succeeds, but a.save() may have been undone
```

Calling `transaction.rollback()` rolls back the entire transaction. Any uncommitted database operations will be lost. In this example, the changes made by `a.save()` would be lost, even though that operation raised no error itself.

Savepoint rollback You can use *savepoints* to control the extent of a rollback. Before performing a database operation that could fail, you can set or update the savepoint; that way, if the operation fails, you can roll back the single offending operation, rather than the entire transaction. For example:

```
a.save() # Succeeds, and never undone by savepoint rollback
sid = transaction.savepoint()
try:
    b.save() # Could throw exception
    transaction.savepoint_commit(sid)
except IntegrityError:
    transaction.savepoint_rollback(sid)
c.save() # Succeeds, and a.save() is never undone
```

In this example, `a.save()` will not be undone in the case where `b.save()` raises an exception.

3.2.8 Multiple databases

This topic guide describes Django's support for interacting with multiple databases. Most of the rest of Django's documentation assumes you are interacting with a single database. If you want to interact with multiple databases, you'll need to take some additional steps.

Defining your databases

The first step to using more than one database with Django is to tell Django about the database servers you'll be using. This is done using the *DATABASES* setting. This setting maps database aliases, which are a way to refer to a specific database throughout Django, to a dictionary of settings for that specific connection. The settings in the inner dictionaries are described fully in the *DATABASES* documentation.

Databases can have any alias you choose. However, the alias `default` has special significance. Django uses the database with the alias of `default` when no other database has been selected.

The following is an example `settings.py` snippet defining two databases – a default PostgreSQL database and a MySQL database called `users`:

```
DATABASES = {
    'default': {
        'NAME': 'app_data',
        'ENGINE': 'django.db.backends.postgresql',
        'USER': 'postgres_user',
        'PASSWORD': 's3krit'
    },
    'users': {
        'NAME': 'user_data',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'priv4te'
    }
}
```

If the concept of a `default` database doesn't make sense in the context of your project, you need to be careful to always specify the database that you want to use. Django requires that a `default` database entry be defined, but the parameters dictionary can be left blank if it will not be used. To do this, you must set up *DATABASE_ROUTERS* for all of your apps' models, including those in any contrib and third-party apps you're using, so that no queries are routed to the default database. The following is an example `settings.py` snippet defining two non-default databases, with the `default` entry intentionally left empty:

```
DATABASES = {
    'default': {},
    'users': {
        'NAME': 'user_data',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'superS3cret'
    },
    'customers': {
        'NAME': 'customer_data',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_cust',
        'PASSWORD': 'veryPriv@ate'
    }
}
```

If you attempt to access a database that you haven't defined in your *DATABASES* setting, Django will raise a `django.db.utils.ConnectionDoesNotExist` exception.

Synchronizing your databases

The *migrate* management command operates on one database at a time. By default, it operates on the `default` database, but by providing the *--database* option, you can tell it to synchronize a different database. So, to synchronize all models onto all databases in the first example above, you would need to call:

```
$ ./manage.py migrate
$ ./manage.py migrate --database=users
```

If you don't want every application to be synchronized onto a particular database, you can define a *database router* that implements a policy constraining the availability of particular models.

If, as in the second example above, you've left the `default` database empty, you must provide a database name each time you run *migrate*. Omitting the database name would raise an error. For the second example:

```
$ ./manage.py migrate --database=users
$ ./manage.py migrate --database=customers
```

Using other management commands

Most other `django-admin` commands that interact with the database operate in the same way as *migrate* – they only ever operate on one database at a time, using `--database` to control the database used.

An exception to this rule is the *makemigrations* command. It validates the migration history in the databases to catch problems with the existing migration files (which could be caused by editing them) before creating new migrations. By default, it checks only the `default` database, but it consults the the *allow_migrate()* method of *routers* if any are installed.

Migration consistency checks were added. Checks based on database routers were added in 1.10.1.

Automatic database routing

The easiest way to use multiple databases is to set up a database routing scheme. The default routing scheme ensures that objects remain 'sticky' to their original database (i.e., an object retrieved from the `foo` database will be saved on the same database). The default routing scheme ensures that if a database isn't specified, all queries fall back to the `default` database.

You don't have to do anything to activate the default routing scheme – it is provided 'out of the box' on every Django project. However, if you want to implement more interesting database allocation behaviors, you can define and install your own database routers.

Database routers

A database Router is a class that provides up to four methods:

db_for_read(*model*, ***hints*)
Suggest the database that should be used for read operations for objects of type `model`.

If a database operation is able to provide any additional information that might assist in selecting a database, it will be provided in the `hints` dictionary. Details on valid hints are provided *below*.

Returns `None` if there is no suggestion.

db_for_write(*model*, ***hints*)
Suggest the database that should be used for writes of objects of type Model.

If a database operation is able to provide any additional information that might assist in selecting a database, it will be provided in the `hints` dictionary. Details on valid hints are provided *below*.

Returns `None` if there is no suggestion.

allow_relation(*obj1*, *obj2*, ***hints*)
Return `True` if a relation between `obj1` and `obj2` should be allowed, `False` if the relation should be prevented, or `None` if the router has no opinion. This is purely a validation operation, used by foreign key and many to many operations to determine if a relation should be allowed between two objects.

allow_migrate(*db*, *app_label*, *model_name=None*, ***hints*)
Determine if the migration operation is allowed to run on the database with alias `db`. Return `True` if the operation should run, `False` if it shouldn't run, or `None` if the router has no opinion.

The `app_label` positional argument is the label of the application being migrated.

`model_name` is set by most migration operations to the value of `model._meta.model_name` (the lowercased version of the model `__name__`) of the model being migrated. Its value is `None` for the *`RunPython`* and *`RunSQL`* operations unless they provide it using hints.

`hints` are used by certain operations to communicate additional information to the router.

When `model_name` is set, `hints` normally contains the model class under the key `'model'`. Note that it may be a *historical model*, and thus not have any custom attributes, methods, or managers. You should only rely on `_meta`.

This method can also be used to determine the availability of a model on a given database.

`makemigrations` always creates migrations for model changes, but if `allow_migrate()` returns `False`, any migration operations for the `model_name` will be silently skipped when running *`migrate`* on the `db`. Changing the behavior of `allow_migrate()` for models that already have migrations may result in broken foreign keys, extra tables, or missing tables. When *`makemigrations`* verifies the migration history, it skips databases where no app is allowed to migrate.

A router doesn't have to provide *all* these methods – it may omit one or more of them. If one of the methods is omitted, Django will skip that router when performing the relevant check.

Hints The hints received by the database router can be used to decide which database should receive a given request.

At present, the only hint that will be provided is `instance`, an object instance that is related to the read or write operation that is underway. This might be the instance that is being saved, or it might be an instance that is being

added in a many-to-many relation. In some cases, no instance hint will be provided at all. The router checks for the existence of an instance hint, and determine if that hint should be used to alter routing behavior.

Using routers

Database routers are installed using the *DATABASE_ROUTERS* setting. This setting defines a list of class names, each specifying a router that should be used by the master router (`django.db.router`).

The master router is used by Django's database operations to allocate database usage. Whenever a query needs to know which database to use, it calls the master router, providing a model and a hint (if available). Django then tries each router in turn until a database suggestion can be found. If no suggestion can be found, it tries the current `_state.db` of the hint instance. If a hint instance wasn't provided, or the instance doesn't currently have database state, the master router will allocate the `default` database.

An example

Example purposes only!

This example is intended as a demonstration of how the router infrastructure can be used to alter database usage. It intentionally ignores some complex issues in order to demonstrate how routers are used.

This example won't work if any of the models in `myapp` contain relationships to models outside of the `other` database. *Cross-database relationships* introduce referential integrity problems that Django can't currently handle.

The primary/replica (referred to as master/slave by some databases) configuration described is also flawed – it doesn't provide any solution for handling replication lag (i.e., query inconsistencies introduced because of the time taken for a write to propagate to the replicas). It also doesn't consider the interaction of transactions with the database utilization strategy.

So - what does this mean in practice? Let's consider another sample configuration. This one will have several databases: one for the `auth` application, and all other apps using a primary/replica setup with two read replicas. Here are the settings specifying these databases:

```
DATABASES = {
    'default': {},
    'auth_db': {
        'NAME': 'auth_db',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'swordfish',
    },
    'primary': {
        'NAME': 'primary',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'spam',
    },
    'replica1': {
        'NAME': 'replica1',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'eggs',
    },
    'replica2': {
        'NAME': 'replica2',
```

```
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'bacon',
    },
}
```

Now we'll need to handle routing. First we want a router that knows to send queries for the `auth` app to `auth_db`:

```
class AuthRouter(object):
    """
    A router to control all database operations on models in the
    auth application.
    """
    def db_for_read(self, model, **hints):
        """
        Attempts to read auth models go to auth_db.
        """
        if model._meta.app_label == 'auth':
            return 'auth_db'
        return None

    def db_for_write(self, model, **hints):
        """
        Attempts to write auth models go to auth_db.
        """
        if model._meta.app_label == 'auth':
            return 'auth_db'
        return None

    def allow_relation(self, obj1, obj2, **hints):
        """
        Allow relations if a model in the auth app is involved.
        """
        if obj1._meta.app_label == 'auth' or \
           obj2._meta.app_label == 'auth':
           return True
        return None

    def allow_migrate(self, db, app_label, model_name=None, **hints):
        """
        Make sure the auth app only appears in the 'auth_db'
        database.
        """
        if app_label == 'auth':
            return db == 'auth_db'
        return None
```

And we also want a router that sends all other apps to the primary/replica configuration, and randomly chooses a replica to read from:

```
import random

class PrimaryReplicaRouter(object):
    def db_for_read(self, model, **hints):
        """
        Reads go to a randomly-chosen replica.
        """
        return random.choice(['replica1', 'replica2'])
```

```
    def db_for_write(self, model, **hints):
        """
        Writes always go to primary.
        """
        return 'primary'

    def allow_relation(self, obj1, obj2, **hints):
        """
        Relations between objects are allowed if both objects are
        in the primary/replica pool.
        """
        db_list = ('primary', 'replica1', 'replica2')
        if obj1._state.db in db_list and obj2._state.db in db_list:
            return True
        return None

    def allow_migrate(self, db, app_label, model_name=None, **hints):
        """
        All non-auth models end up in this pool.
        """
        return True
```

Finally, in the settings file, we add the following (substituting `path.to.` with the actual Python path to the module(s) where the routers are defined):

```
DATABASE_ROUTERS = ['path.to.AuthRouter', 'path.to.PrimaryReplicaRouter']
```

The order in which routers are processed is significant. Routers will be queried in the order they are listed in the *DATABASE_ROUTERS* setting. In this example, the `AuthRouter` is processed before the `PrimaryReplicaRouter`, and as a result, decisions concerning the models in `auth` are processed before any other decision is made. If the *DATABASE_ROUTERS* setting listed the two routers in the other order, `PrimaryReplicaRouter.allow_migrate()` would be processed first. The catch-all nature of the PrimaryReplicaRouter implementation would mean that all models would be available on all databases.

With this setup installed, lets run some Django code:

```
>>> # This retrieval will be performed on the 'auth_db' database
>>> fred = User.objects.get(username='fred')
>>> fred.first_name = 'Frederick'

>>> # This save will also be directed to 'auth_db'
>>> fred.save()

>>> # These retrieval will be randomly allocated to a replica database
>>> dna = Person.objects.get(name='Douglas Adams')

>>> # A new object has no database allocation when created
>>> mh = Book(title='Mostly Harmless')

>>> # This assignment will consult the router, and set mh onto
>>> # the same database as the author object
>>> mh.author = dna

>>> # This save will force the 'mh' instance onto the primary database...
>>> mh.save()

>>> # ... but if we re-retrieve the object, it will come back on a replica
>>> mh = Book.objects.get(title='Mostly Harmless')
```

This example defined a router to handle interaction with models from the `auth` app, and other routers to handle interaction with all other apps. If you left your `default` database empty and don't want to define a catch-all database router to handle all apps not otherwise specified, your routers must handle the names of all apps in *`INSTALLED_APPS`* before you migrate. See *Behavior of contrib apps* for information about contrib apps that must be together in one database.

Manually selecting a database

Django also provides an API that allows you to maintain complete control over database usage in your code. A manually specified database allocation will take priority over a database allocated by a router.

Manually selecting a database for a `QuerySet`

You can select the database for a `QuerySet` at any point in the `QuerySet` "chain." Just call `using()` on the `QuerySet` to get another `QuerySet` that uses the specified database.

`using()` takes a single argument: the alias of the database on which you want to run the query. For example:

```
>>> # This will run on the 'default' database.
>>> Author.objects.all()

>>> # So will this.
>>> Author.objects.using('default').all()

>>> # This will run on the 'other' database.
>>> Author.objects.using('other').all()
```

Selecting a database for `save()`

Use the `using` keyword to `Model.save()` to specify to which database the data should be saved.

For example, to save an object to the `legacy_users` database, you'd use this:

```
>>> my_object.save(using='legacy_users')
```

If you don't specify `using`, the `save()` method will save into the default database allocated by the routers.

Moving an object from one database to another If you've saved an instance to one database, it might be tempting to use `save(using=...)` as a way to migrate the instance to a new database. However, if you don't take appropriate steps, this could have some unexpected consequences.

Consider the following example:

```
>>> p = Person(name='Fred')
>>> p.save(using='first')  # (statement 1)
>>> p.save(using='second') # (statement 2)
```

In statement 1, a new `Person` object is saved to the `first` database. At this time, `p` doesn't have a primary key, so Django issues an SQL `INSERT` statement. This creates a primary key, and Django assigns that primary key to `p`.

When the save occurs in statement 2, `p` already has a primary key value, and Django will attempt to use that primary key on the new database. If the primary key value isn't in use in the `second` database, then you won't have any problems – the object will be copied to the new database.

However, if the primary key of p is already in use on the second database, the existing object in the second database will be overridden when p is saved.

You can avoid this in two ways. First, you can clear the primary key of the instance. If an object has no primary key, Django will treat it as a new object, avoiding any loss of data on the second database:

```
>>> p = Person(name='Fred')
>>> p.save(using='first')
>>> p.pk = None # Clear the primary key.
>>> p.save(using='second') # Write a completely new object.
```

The second option is to use the force_insert option to save() to ensure that Django does an SQL INSERT:

```
>>> p = Person(name='Fred')
>>> p.save(using='first')
>>> p.save(using='second', force_insert=True)
```

This will ensure that the person named Fred will have the same primary key on both databases. If that primary key is already in use when you try to save onto the second database, an error will be raised.

Selecting a database to delete from

By default, a call to delete an existing object will be executed on the same database that was used to retrieve the object in the first place:

```
>>> u = User.objects.using('legacy_users').get(username='fred')
>>> u.delete() # will delete from the `legacy_users` database
```

To specify the database from which a model will be deleted, pass a using keyword argument to the Model.delete() method. This argument works just like the using keyword argument to save().

For example, if you're migrating a user from the legacy_users database to the new_users database, you might use these commands:

```
>>> user_obj.save(using='new_users')
>>> user_obj.delete(using='legacy_users')
```

Using managers with multiple databases

Use the db_manager() method on managers to give managers access to a non-default database.

For example, say you have a custom manager method that touches the database – User.objects.create_user(). Because create_user() is a manager method, not a QuerySet method, you can't do User.objects.using('new_users').create_user(). (The create_user() method is only available on User.objects, the manager, not on QuerySet objects derived from the manager.) The solution is to use db_manager(), like this:

```
User.objects.db_manager('new_users').create_user(...)
```

db_manager() returns a copy of the manager bound to the database you specify.

Using `get_queryset()` with multiple databases If you're overriding get_queryset() on your manager, be sure to either call the method on the parent (using super()) or do the appropriate handling of the _db attribute on the manager (a string containing the name of the database to use).

For example, if you want to return a custom QuerySet class from the get_queryset method, you could do this:

```
class MyManager(models.Manager):
    def get_queryset(self):
        qs = CustomQuerySet(self.model)
        if self._db is not None:
            qs = qs.using(self._db)
        return qs
```

Exposing multiple databases in Django's admin interface

Django's admin doesn't have any explicit support for multiple databases. If you want to provide an admin interface for a model on a database other than that specified by your router chain, you'll need to write custom *ModelAdmin* classes that will direct the admin to use a specific database for content.

`ModelAdmin` objects have five methods that require customization for multiple-database support:

```
class MultiDBModelAdmin(admin.ModelAdmin):
    # A handy constant for the name of the alternate database.
    using = 'other'

    def save_model(self, request, obj, form, change):
        # Tell Django to save objects to the 'other' database.
        obj.save(using=self.using)

    def delete_model(self, request, obj):
        # Tell Django to delete objects from the 'other' database
        obj.delete(using=self.using)

    def get_queryset(self, request):
        # Tell Django to look for objects on the 'other' database.
        return super(MultiDBModelAdmin, self).get_queryset(request).using(self.using)

    def formfield_for_foreignkey(self, db_field, request, **kwargs):
        # Tell Django to populate ForeignKey widgets using a query
        # on the 'other' database.
        return super(MultiDBModelAdmin, self).formfield_for_foreignkey(db_field, request, using=self.

    def formfield_for_manytomany(self, db_field, request, **kwargs):
        # Tell Django to populate ManyToMany widgets using a query
        # on the 'other' database.
        return super(MultiDBModelAdmin, self).formfield_for_manytomany(db_field, request, using=self.
```

The implementation provided here implements a multi-database strategy where all objects of a given type are stored on a specific database (e.g., all `User` objects are in the `other` database). If your usage of multiple databases is more complex, your `ModelAdmin` will need to reflect that strategy.

InlineModelAdmin objects can be handled in a similar fashion. They require three customized methods:

```
class MultiDBTabularInline(admin.TabularInline):
    using = 'other'

    def get_queryset(self, request):
        # Tell Django to look for inline objects on the 'other' database.
        return super(MultiDBTabularInline, self).get_queryset(request).using(self.using)

    def formfield_for_foreignkey(self, db_field, request, **kwargs):
        # Tell Django to populate ForeignKey widgets using a query
        # on the 'other' database.
        return super(MultiDBTabularInline, self).formfield_for_foreignkey(db_field, request, using=se
```

```
    def formfield_for_manytomany(self, db_field, request, **kwargs):
        # Tell Django to populate ManyToMany widgets using a query
        # on the 'other' database.
        return super(MultiDBTabularInline, self).formfield_for_manytomany(db_field, request, using=se
```

Once you've written your model admin definitions, they can be registered with any `Admin` instance:

```
from django.contrib import admin

# Specialize the multi-db admin objects for use with specific models.
class BookInline(MultiDBTabularInline):
    model = Book

class PublisherAdmin(MultiDBModelAdmin):
    inlines = [BookInline]

admin.site.register(Author, MultiDBModelAdmin)
admin.site.register(Publisher, PublisherAdmin)

othersite = admin.AdminSite('othersite')
othersite.register(Publisher, MultiDBModelAdmin)
```

This example sets up two admin sites. On the first site, the `Author` and `Publisher` objects are exposed; `Publisher` objects have an tabular inline showing books published by that publisher. The second site exposes just publishers, without the inlines.

Using raw cursors with multiple databases

If you are using more than one database you can use `django.db.connections` to obtain the connection (and cursor) for a specific database. `django.db.connections` is a dictionary-like object that allows you to retrieve a specific connection using its alias:

```
from django.db import connections
cursor = connections['my_db_alias'].cursor()
```

Limitations of multiple databases

Cross-database relations

Django doesn't currently provide any support for foreign key or many-to-many relationships spanning multiple databases. If you have used a router to partition models to different databases, any foreign key and many-to-many relationships defined by those models must be internal to a single database.

This is because of referential integrity. In order to maintain a relationship between two objects, Django needs to know that the primary key of the related object is valid. If the primary key is stored on a separate database, it's not possible to easily evaluate the validity of a primary key.

If you're using Postgres, Oracle, or MySQL with InnoDB, this is enforced at the database integrity level – database level key constraints prevent the creation of relations that can't be validated.

However, if you're using SQLite or MySQL with MyISAM tables, there is no enforced referential integrity; as a result, you may be able to 'fake' cross database foreign keys. However, this configuration is not officially supported by Django.

Behavior of contrib apps

Several contrib apps include models, and some apps depend on others. Since cross-database relationships are impossible, this creates some restrictions on how you can split these models across databases:

- each one of `contenttypes.ContentType`, `sessions.Session` and `sites.Site` can be stored in any database, given a suitable router.
- `auth` models — `User`, `Group` and `Permission` — are linked together and linked to `ContentType`, so they must be stored in the same database as `ContentType`.
- `admin` depends on `auth`, so its models must be in the same database as `auth`.
- `flatpages` and `redirects` depend on `sites`, so their models must be in the same database as `sites`.

In addition, some objects are automatically created just after *`migrate`* creates a table to hold them in a database:

- a default `Site`,
- a `ContentType` for each model (including those not stored in that database),
- three `Permission` for each model (including those not stored in that database).

For common setups with multiple databases, it isn't useful to have these objects in more than one database. Common setups include primary/replica and connecting to external databases. Therefore, it's recommended to write a *database router* that allows synchronizing these three models to only one database. Use the same approach for contrib and third-party apps that don't need their tables in multiple databases.

: If you're synchronizing content types to more than one database, be aware that their primary keys may not match across databases. This may result in data corruption or data loss.

3.2.9 Tablespaces

A common paradigm for optimizing performance in database systems is the use of tablespaces to organize disk layout.

: Django does not create the tablespaces for you. Please refer to your database engine's documentation for details on creating and managing tablespaces.

Declaring tablespaces for tables

A tablespace can be specified for the table generated by a model by supplying the *`db_tablespace`* option inside the model's `class Meta`. This option also affects tables automatically created for *`ManyToManyField`*s in the model.

You can use the *`DEFAULT_TABLESPACE`* setting to specify a default value for *`db_tablespace`*. This is useful for setting a tablespace for the built-in Django apps and other applications whose code you cannot control.

Declaring tablespaces for indexes

You can pass the *`db_tablespace`* option to a `Field` constructor to specify an alternate tablespace for the `Field`'s column index. If no index would be created for the column, the option is ignored.

You can use the *`DEFAULT_INDEX_TABLESPACE`* setting to specify a default value for *`db_tablespace`*.

If *`db_tablespace`* isn't specified and you didn't set *`DEFAULT_INDEX_TABLESPACE`*, the index is created in the same tablespace as the tables.

An example

```
class TablespaceExample(models.Model):
    name = models.CharField(max_length=30, db_index=True, db_tablespace="indexes")
    data = models.CharField(max_length=255, db_index=True)
    edges = models.ManyToManyField(to="self", db_tablespace="indexes")

    class Meta:
        db_tablespace = "tables"
```

In this example, the tables generated by the `TablespaceExample` model (i.e. the model table and the many-to-many table) would be stored in the `tables` tablespace. The index for the name field and the indexes on the many-to-many table would be stored in the `indexes` tablespace. The `data` field would also generate an index, but no tablespace for it is specified, so it would be stored in the model tablespace `tables` by default.

Database support

PostgreSQL and Oracle support tablespaces. SQLite and MySQL don't.

When you use a backend that lacks support for tablespaces, Django ignores all tablespace-related options.

3.2.10 Database access optimization

Django's database layer provides various ways to help developers get the most out of their databases. This document gathers together links to the relevant documentation, and adds various tips, organized under a number of headings that outline the steps to take when attempting to optimize your database usage.

Profile first

As general programming practice, this goes without saying. Find out *what queries you are doing and what they are costing you*. You may also want to use an external project like django-debug-toolbar, or a tool that monitors your database directly.

Remember that you may be optimizing for speed or memory or both, depending on your requirements. Sometimes optimizing for one will be detrimental to the other, but sometimes they will help each other. Also, work that is done by the database process might not have the same cost (to you) as the same amount of work done in your Python process. It is up to you to decide what your priorities are, where the balance must lie, and profile all of these as required since this will depend on your application and server.

With everything that follows, remember to profile after every change to ensure that the change is a benefit, and a big enough benefit given the decrease in readability of your code. **All** of the suggestions below come with the caveat that in your circumstances the general principle might not apply, or might even be reversed.

Use standard DB optimization techniques

...including:

- Indexes. This is a number one priority, *after* you have determined from profiling what indexes should be added. Use `Field.db_index` or `Meta.index_together` to add these from Django. Consider adding indexes to fields that you frequently query using `filter()`, `exclude()`, `order_by()`, etc. as indexes may help to speed up lookups. Note that determining the best indexes is a complex database-dependent topic that will depend on your particular application. The overhead of maintaining an index may outweigh any gains in query speed.

- Appropriate use of field types.

We will assume you have done the obvious things above. The rest of this document focuses on how to use Django in such a way that you are not doing unnecessary work. This document also does not address other optimization techniques that apply to all expensive operations, such as general purpose caching.

Understand `QuerySets`

Understanding QuerySets is vital to getting good performance with simple code. In particular:

Understand `QuerySet` evaluation

To avoid performance problems, it is important to understand:

- that *QuerySets are lazy*.
- when *they are evaluated*.
- how *the data is held in memory*.

Understand cached attributes

As well as caching of the whole `QuerySet`, there is caching of the result of attributes on ORM objects. In general, attributes that are not callable will be cached. For example, assuming the *example Weblog models*:

```
>>> entry = Entry.objects.get(id=1)
>>> entry.blog   # Blog object is retrieved at this point
>>> entry.blog   # cached version, no DB access
```

But in general, callable attributes cause DB lookups every time:

```
>>> entry = Entry.objects.get(id=1)
>>> entry.authors.all()   # query performed
>>> entry.authors.all()   # query performed again
```

Be careful when reading template code - the template system does not allow use of parentheses, but will call callables automatically, hiding the above distinction.

Be careful with your own custom properties - it is up to you to implement caching when required, for example using the *`cached_property`* decorator.

Use the `with` template tag

To make use of the caching behavior of `QuerySet`, you may need to use the *`with`* template tag.

Use `iterator()`

When you have a lot of objects, the caching behavior of the `QuerySet` can cause a large amount of memory to be used. In this case, *`iterator()`* may help.

Do database work in the database rather than in Python

For instance:

- At the most basic level, use *filter and exclude* to do filtering in the database.
- Use *F expressions* to filter based on other fields within the same model.
- Use annotate to do aggregation in the database.

If these aren't enough to generate the SQL you need:

Use RawSQL

A less portable but more powerful method is the *RawSQL* expression, which allows some SQL to be explicitly added to the query. If that still isn't powerful enough:

Use raw SQL

Write your own custom SQL to retrieve data or populate models. Use `django.db.connection.queries` to find out what Django is writing for you and start from there.

Retrieve individual objects using a unique, indexed column

There are two reasons to use a column with *unique* or *db_index* when using *get()* to retrieve individual objects. First, the query will be quicker because of the underlying database index. Also, the query could run much slower if multiple objects match the lookup; having a unique constraint on the column guarantees this will never happen.

So using the *example Weblog models*:

```
>>> entry = Entry.objects.get(id=10)
```

will be quicker than:

```
>>> entry = Entry.objects.get(headline="News Item Title")
```

because `id` is indexed by the database and is guaranteed to be unique.

Doing the following is potentially quite slow:

```
>>> entry = Entry.objects.get(headline__startswith="News")
```

First of all, `headline` is not indexed, which will make the underlying database fetch slower.

Second, the lookup doesn't guarantee that only one object will be returned. If the query matches more than one object, it will retrieve and transfer all of them from the database. This penalty could be substantial if hundreds or thousands of records are returned. The penalty will be compounded if the database lives on a separate server, where network overhead and latency also play a factor.

Retrieve everything at once if you know you will need it

Hitting the database multiple times for different parts of a single 'set' of data that you will need all parts of is, in general, less efficient than retrieving it all in one query. This is particularly important if you have a query that is executed in a loop, and could therefore end up doing many database queries, when only one was needed. So:

Use `QuerySet.select_related()` and `prefetch_related()`

Understand *`select_related()`* and *`prefetch_related()`* thoroughly, and use them:

- in view code,
- and in managers and default managers where appropriate. Be aware when your manager is and is not used; sometimes this is tricky so don't make assumptions.

Don't retrieve things you don't need

Use `QuerySet.values()` and `values_list()`

When you just want a `dict` or `list` of values, and don't need ORM model objects, make appropriate usage of *`values()`*. These can be useful for replacing model objects in template code - as long as the dicts you supply have the same attributes as those used in the template, you are fine.

Use `QuerySet.defer()` and `only()`

Use *`defer()`* and *`only()`* if there are database columns you know that you won't need (or won't need in most cases) to avoid loading them. Note that if you *do* use them, the ORM will have to go and get them in a separate query, making this a pessimization if you use it inappropriately.

Also, be aware that there is some (small extra) overhead incurred inside Django when constructing a model with deferred fields. Don't be too aggressive in deferring fields without profiling as the database has to read most of the non-text, non-VARCHAR data from the disk for a single row in the results, even if it ends up only using a few columns. The `defer()` and `only()` methods are most useful when you can avoid loading a lot of text data or for fields that might take a lot of processing to convert back to Python. As always, profile first, then optimize.

Use `QuerySet.count()`

...if you only want the count, rather than doing `len(queryset)`.

Use `QuerySet.exists()`

...if you only want to find out if at least one result exists, rather than `if queryset`.

But:

Don't overuse `count()` and `exists()`

If you are going to need other data from the QuerySet, just evaluate it.

For example, assuming an Email model that has a `body` attribute and a many-to-many relation to User, the following template code is optimal:

```
{% if display_inbox %}
  {% with emails=user.emails.all %}
    {% if emails %}
      <p>You have {{ emails|length }} email(s)</p>
      {% for email in emails %}
        <p>{{ email.body }}</p>
```

```
      {% endfor %}
    {% else %}
      <p>No messages today.</p>
    {% endif %}
  {% endwith %}
{% endif %}
```

It is optimal because:

1. Since QuerySets are lazy, this does no database queries if 'display_inbox' is False.
2. Use of *with* means that we store `user.emails.all` in a variable for later use, allowing its cache to be re-used.
3. The line `{% if emails %}` causes `QuerySet.__bool__()` to be called, which causes the `user.emails.all()` query to be run on the database, and at the least the first line to be turned into an ORM object. If there aren't any results, it will return False, otherwise True.
4. The use of `{{ emails|length }}` calls `QuerySet.__len__()`, filling out the rest of the cache without doing another query.
5. The *for* loop iterates over the already filled cache.

In total, this code does either one or zero database queries. The only deliberate optimization performed is the use of the *with* tag. Using `QuerySet.exists()` or `QuerySet.count()` at any point would cause additional queries.

Use `QuerySet.update()` and `delete()`

Rather than retrieve a load of objects, set some values, and save them individual, use a bulk SQL UPDATE statement, via *QuerySet.update()*. Similarly, do *bulk deletes* where possible.

Note, however, that these bulk update methods cannot call the `save()` or `delete()` methods of individual instances, which means that any custom behavior you have added for these methods will not be executed, including anything driven from the normal database object signals.

Use foreign key values directly

If you only need a foreign key value, use the foreign key value that is already on the object you've got, rather than getting the whole related object and taking its primary key. i.e. do:

```
entry.blog_id
```

instead of:

```
entry.blog.id
```

Don't order results if you don't care

Ordering is not free; each field to order by is an operation the database must perform. If a model has a default ordering (*Meta.ordering*) and you don't need it, remove it on a `QuerySet` by calling *order_by()* with no parameters.

Adding an index to your database may help to improve ordering performance.

Insert in bulk

When creating objects, where possible, use the *bulk_create()* method to reduce the number of SQL queries. For example:

```
Entry.objects.bulk_create([
    Entry(headline="Python 3.0 Released"),
    Entry(headline="Python 3.1 Planned")
])
```

...is preferable to:

```
Entry.objects.create(headline="Python 3.0 Released")
Entry.objects.create(headline="Python 3.1 Planned")
```

Note that there are a number of *caveats to this method*, so make sure it's appropriate for your use case.

This also applies to *ManyToManyFields*, so doing:

```
my_band.members.add(me, my_friend)
```

...is preferable to:

```
my_band.members.add(me)
my_band.members.add(my_friend)
```

...where `Bands` and `Artists` have a many-to-many relationship.

3.2.11 Examples of model relationship API usage

Many-to-many relationships

To define a many-to-many relationship, use *ManyToManyField*.

In this example, an `Article` can be published in multiple `Publication` objects, and a `Publication` has multiple `Article` objects:

```
from django.db import models

class Publication(models.Model):
    title = models.CharField(max_length=30)

    def __str__(self):              # __unicode__ on Python 2
        return self.title

    class Meta:
        ordering = ('title',)

class Article(models.Model):
    headline = models.CharField(max_length=100)
    publications = models.ManyToManyField(Publication)

    def __str__(self):              # __unicode__ on Python 2
        return self.headline

    class Meta:
        ordering = ('headline',)
```

What follows are examples of operations that can be performed using the Python API facilities. Note that if you are using *an intermediate model* for a many-to-many relationship, some of the related manager's methods are disabled, so some of these examples won't work with such models.

Create a couple of `Publications`:

```
>>> p1 = Publication(title='The Python Journal')
>>> p1.save()
>>> p2 = Publication(title='Science News')
>>> p2.save()
>>> p3 = Publication(title='Science Weekly')
>>> p3.save()
```

Create an `Article`:

```
>>> a1 = Article(headline='Django lets you build Web apps easily')
```

You can't associate it with a `Publication` until it's been saved:

```
>>> a1.publications.add(p1)
Traceback (most recent call last):
...
ValueError: 'Article' instance needs to have a primary key value before a many-to-many relationship c
```

Save it!

```
>>> a1.save()
```

Associate the `Article` with a `Publication`:

```
>>> a1.publications.add(p1)
```

Create another `Article`, and set it to appear in both `Publications`:

```
>>> a2 = Article(headline='NASA uses Python')
>>> a2.save()
>>> a2.publications.add(p1, p2)
>>> a2.publications.add(p3)
```

Adding a second time is OK:

```
>>> a2.publications.add(p3)
```

Adding an object of the wrong type raises `TypeError`:

```
>>> a2.publications.add(a1)
Traceback (most recent call last):
...
TypeError: 'Publication' instance expected
```

Create and add a `Publication` to an `Article` in one step using *`create()`*:

```
>>> new_publication = a2.publications.create(title='Highlights for Children')
```

`Article` objects have access to their related `Publication` objects:

```
>>> a1.publications.all()
<QuerySet [<Publication: The Python Journal>]>
>>> a2.publications.all()
<QuerySet [<Publication: Highlights for Children>, <Publication: Science News>, <Publication: Science
```

`Publication` objects have access to their related `Article` objects:

```
>>> p2.article_set.all()
<QuerySet [<Article: NASA uses Python>]>
>>> p1.article_set.all()
<QuerySet [<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]>
>>> Publication.objects.get(id=4).article_set.all()
<QuerySet [<Article: NASA uses Python>]>
```

Many-to-many relationships can be queried using *lookups across relationships*:

```
>>> Article.objects.filter(publications__id=1)
<QuerySet [<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]>
>>> Article.objects.filter(publications__pk=1)
<QuerySet [<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]>
>>> Article.objects.filter(publications=1)
<QuerySet [<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]>
>>> Article.objects.filter(publications=p1)
<QuerySet [<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]>

>>> Article.objects.filter(publications__title__startswith="Science")
<QuerySet [<Article: NASA uses Python>, <Article: NASA uses Python>]>

>>> Article.objects.filter(publications__title__startswith="Science").distinct()
<QuerySet [<Article: NASA uses Python>]>
```

The *count()* function respects *distinct()* as well:

```
>>> Article.objects.filter(publications__title__startswith="Science").count()
2

>>> Article.objects.filter(publications__title__startswith="Science").distinct().count()
1

>>> Article.objects.filter(publications__in=[1,2]).distinct()
<QuerySet [<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]>
>>> Article.objects.filter(publications__in=[p1,p2]).distinct()
<QuerySet [<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]>
```

Reverse m2m queries are supported (i.e., starting at the table that doesn't have a *ManyToManyField*):

```
>>> Publication.objects.filter(id=1)
<QuerySet [<Publication: The Python Journal>]>
>>> Publication.objects.filter(pk=1)
<QuerySet [<Publication: The Python Journal>]>

>>> Publication.objects.filter(article__headline__startswith="NASA")
<QuerySet [<Publication: Highlights for Children>, <Publication: Science News>, <Publication: Science

>>> Publication.objects.filter(article__id=1)
<QuerySet [<Publication: The Python Journal>]>
>>> Publication.objects.filter(article__pk=1)
<QuerySet [<Publication: The Python Journal>]>
>>> Publication.objects.filter(article=1)
<QuerySet [<Publication: The Python Journal>]>
>>> Publication.objects.filter(article=a1)
<QuerySet [<Publication: The Python Journal>]>

>>> Publication.objects.filter(article__in=[1,2]).distinct()
<QuerySet [<Publication: Highlights for Children>, <Publication: Science News>, <Publication: Science
>>> Publication.objects.filter(article__in=[a1,a2]).distinct()
```

```
<QuerySet [<Publication: Highlights for Children>, <Publication: Science News>, <Publication: Science
```

Excluding a related item works as you would expect, too (although the SQL involved is a little complex):

```
>>> Article.objects.exclude(publications=p2)
<QuerySet [<Article: Django lets you build Web apps easily>]>
```

If we delete a `Publication`, its `Articles` won't be able to access it:

```
>>> p1.delete()
>>> Publication.objects.all()
<QuerySet [<Publication: Highlights for Children>, <Publication: Science News>, <Publication: Science
>>> a1 = Article.objects.get(pk=1)
>>> a1.publications.all()
<QuerySet []>
```

If we delete an `Article`, its `Publications` won't be able to access it:

```
>>> a2.delete()
>>> Article.objects.all()
<QuerySet [<Article: Django lets you build Web apps easily>]>
>>> p2.article_set.all()
<QuerySet []>
```

Adding via the 'other' end of an m2m:

```
>>> a4 = Article(headline='NASA finds intelligent life on Earth')
>>> a4.save()
>>> p2.article_set.add(a4)
>>> p2.article_set.all()
<QuerySet [<Article: NASA finds intelligent life on Earth>]>
>>> a4.publications.all()
<QuerySet [<Publication: Science News>]>
```

Adding via the other end using keywords:

```
>>> new_article = p2.article_set.create(headline='Oxygen-free diet works wonders')
>>> p2.article_set.all()
<QuerySet [<Article: NASA finds intelligent life on Earth>, <Article: Oxygen-free diet works wonders>
>>> a5 = p2.article_set.all()[1]
>>> a5.publications.all()
<QuerySet [<Publication: Science News>]>
```

Removing `Publication` from an `Article`:

```
>>> a4.publications.remove(p2)
>>> p2.article_set.all()
<QuerySet [<Article: Oxygen-free diet works wonders>]>
>>> a4.publications.all()
<QuerySet []>
```

And from the other end:

```
>>> p2.article_set.remove(a5)
>>> p2.article_set.all()
<QuerySet []>
>>> a5.publications.all()
<QuerySet []>
```

Relation sets can be set:

```
>>> a4.publications.all()
<QuerySet [<Publication: Science News>]>
>>> a4.publications.set([p3])
>>> a4.publications.all()
<QuerySet [<Publication: Science Weekly>]>
```

Relation sets can be cleared:

```
>>> p2.article_set.clear()
>>> p2.article_set.all()
<QuerySet []>
```

And you can clear from the other end:

```
>>> p2.article_set.add(a4, a5)
>>> p2.article_set.all()
<QuerySet [<Article: NASA finds intelligent life on Earth>, <Article: Oxygen-free diet works wonders>
>>> a4.publications.all()
<QuerySet [<Publication: Science News>, <Publication: Science Weekly>]>
>>> a4.publications.clear()
>>> a4.publications.all()
<QuerySet []>
>>> p2.article_set.all()
<QuerySet [<Article: Oxygen-free diet works wonders>]>
```

Recreate the `Article` and `Publication` we have deleted:

```
>>> p1 = Publication(title='The Python Journal')
>>> p1.save()
>>> a2 = Article(headline='NASA uses Python')
>>> a2.save()
>>> a2.publications.add(p1, p2, p3)
```

Bulk delete some `Publications` - references to deleted publications should go:

```
>>> Publication.objects.filter(title__startswith='Science').delete()
>>> Publication.objects.all()
<QuerySet [<Publication: Highlights for Children>, <Publication: The Python Journal>]>
>>> Article.objects.all()
<QuerySet [<Article: Django lets you build Web apps easily>, <Article: NASA finds intelligent life o
>>> a2.publications.all()
<QuerySet [<Publication: The Python Journal>]>
```

Bulk delete some articles - references to deleted objects should go:

```
>>> q = Article.objects.filter(headline__startswith='Django')
>>> print(q)
<QuerySet [<Article: Django lets you build Web apps easily>]>
>>> q.delete()
```

After the *`delete()`*, the *`QuerySet`* cache needs to be cleared, and the referenced objects should be gone:

```
>>> print(q)
<QuerySet []>
>>> p1.article_set.all()
<QuerySet [<Article: NASA uses Python>]>
```

Many-to-one relationships

To define a many-to-one relationship, use *ForeignKey*:

```
from django.db import models

class Reporter(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    email = models.EmailField()

    def __str__(self):              # __unicode__ on Python 2
        return "%s %s" % (self.first_name, self.last_name)

class Article(models.Model):
    headline = models.CharField(max_length=100)
    pub_date = models.DateField()
    reporter = models.ForeignKey(Reporter, on_delete=models.CASCADE)

    def __str__(self):              # __unicode__ on Python 2
        return self.headline

    class Meta:
        ordering = ('headline',)
```

What follows are examples of operations that can be performed using the Python API facilities.

Create a few Reporters:

```
>>> r = Reporter(first_name='John', last_name='Smith', email='john@example.com')
>>> r.save()

>>> r2 = Reporter(first_name='Paul', last_name='Jones', email='paul@example.com')
>>> r2.save()
```

Create an Article:

```
>>> from datetime import date
>>> a = Article(id=None, headline="This is a test", pub_date=date(2005, 7, 27), reporter=r)
>>> a.save()

>>> a.reporter.id
1

>>> a.reporter
<Reporter: John Smith>
```

Note that you must save an object before it can be assigned to a foreign key relationship. For example, creating an `Article` with unsaved `Reporter` raises `ValueError`:

```
>>> r3 = Reporter(first_name='John', last_name='Smith', email='john@example.com')
>>> Article.objects.create(headline="This is a test", pub_date=date(2005, 7, 27), reporter=r3)
Traceback (most recent call last):
...
ValueError: save() prohibited to prevent data loss due to unsaved related object 'reporter'.
```

Article objects have access to their related Reporter objects:

```
>>> r = a.reporter
```

On Python 2, these are strings of type `str` instead of unicode strings because that's what was used in the creation of this reporter (and we haven't refreshed the data from the database, which always returns unicode strings):

```
>>> r.first_name, r.last_name
('John', 'Smith')
```

Create an Article via the Reporter object:

```
>>> new_article = r.article_set.create(headline="John's second story", pub_date=date(2005, 7, 29))
>>> new_article
<Article: John's second story>
>>> new_article.reporter
<Reporter: John Smith>
>>> new_article.reporter.id
1
```

Create a new article, and add it to the article set:

```
>>> new_article2 = Article(headline="Paul's story", pub_date=date(2006, 1, 17))
>>> r.article_set.add(new_article2)
>>> new_article2.reporter
<Reporter: John Smith>
>>> new_article2.reporter.id
1
>>> r.article_set.all()
<QuerySet [<Article: John's second story>, <Article: Paul's story>, <Article: This is a test>]>
```

Add the same article to a different article set - check that it moves:

```
>>> r2.article_set.add(new_article2)
>>> new_article2.reporter.id
2
>>> new_article2.reporter
<Reporter: Paul Jones>
```

Adding an object of the wrong type raises TypeError:

```
>>> r.article_set.add(r2)
Traceback (most recent call last):
...
TypeError: 'Article' instance expected

>>> r.article_set.all()
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
>>> r2.article_set.all()
<QuerySet [<Article: Paul's story>]>

>>> r.article_set.count()
2

>>> r2.article_set.count()
1
```

Note that in the last example the article has moved from John to Paul.

Related managers support field lookups as well. The API automatically follows relationships as far as you need. Use double underscores to separate relationships. This works as many levels deep as you want. There's no limit. For example:

```
>>> r.article_set.filter(headline__startswith='This')
<QuerySet [<Article: This is a test>]>

# Find all Articles for any Reporter whose first name is "John".
>>> Article.objects.filter(reporter__first_name='John')
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
```

Exact match is implied here:

```
>>> Article.objects.filter(reporter__first_name='John')
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
```

Query twice over the related field. This translates to an AND condition in the WHERE clause:

```
>>> Article.objects.filter(reporter__first_name='John', reporter__last_name='Smith')
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
```

For the related lookup you can supply a primary key value or pass the related object explicitly:

```
>>> Article.objects.filter(reporter__pk=1)
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
>>> Article.objects.filter(reporter=1)
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
>>> Article.objects.filter(reporter=r)
<QuerySet [<Article: John's second story>, <Article: This is a test>]>

>>> Article.objects.filter(reporter__in=[1,2]).distinct()
<QuerySet [<Article: John's second story>, <Article: Paul's story>, <Article: This is a test>]>
>>> Article.objects.filter(reporter__in=[r,r2]).distinct()
<QuerySet [<Article: John's second story>, <Article: Paul's story>, <Article: This is a test>]>
```

You can also use a queryset instead of a literal list of instances:

```
>>> Article.objects.filter(reporter__in=Reporter.objects.filter(first_name='John')).distinct()
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
```

Querying in the opposite direction:

```
>>> Reporter.objects.filter(article__pk=1)
<QuerySet [<Reporter: John Smith>]>
>>> Reporter.objects.filter(article=1)
<QuerySet [<Reporter: John Smith>]>
>>> Reporter.objects.filter(article=a)
<QuerySet [<Reporter: John Smith>]>

>>> Reporter.objects.filter(article__headline__startswith='This')
<QuerySet [<Reporter: John Smith>, <Reporter: John Smith>, <Reporter: John Smith>]>
>>> Reporter.objects.filter(article__headline__startswith='This').distinct()
<QuerySet [<Reporter: John Smith>]>
```

Counting in the opposite direction works in conjunction with distinct():

```
>>> Reporter.objects.filter(article__headline__startswith='This').count()
3
>>> Reporter.objects.filter(article__headline__startswith='This').distinct().count()
1
```

Queries can go round in circles:

```
>>> Reporter.objects.filter(article__reporter__first_name__startswith='John')
<QuerySet [<Reporter: John Smith>, <Reporter: John Smith>, <Reporter: John Smith>, <Reporter: John Sm
>>> Reporter.objects.filter(article__reporter__first_name__startswith='John').distinct()
<QuerySet [<Reporter: John Smith>]>
>>> Reporter.objects.filter(article__reporter=r).distinct()
<QuerySet [<Reporter: John Smith>]>
```

If you delete a reporter, his articles will be deleted (assuming that the ForeignKey was defined with *django.db.models.ForeignKey.on_delete* set to CASCADE, which is the default):

```
>>> Article.objects.all()
<QuerySet [<Article: John's second story>, <Article: Paul's story>, <Article: This is a test>]>
>>> Reporter.objects.order_by('first_name')
<QuerySet [<Reporter: John Smith>, <Reporter: Paul Jones>]>
>>> r2.delete()
>>> Article.objects.all()
<QuerySet [<Article: John's second story>, <Article: This is a test>]>
>>> Reporter.objects.order_by('first_name')
<QuerySet [<Reporter: John Smith>]>
```

You can delete using a JOIN in the query:

```
>>> Reporter.objects.filter(article__headline__startswith='This').delete()
>>> Reporter.objects.all()
<QuerySet []>
>>> Article.objects.all()
<QuerySet []>
```

One-to-one relationships

To define a one-to-one relationship, use *OneToOneField*.

In this example, a Place optionally can be a Restaurant:

```
from django.db import models

class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

    def __str__(self):              # __unicode__ on Python 2
        return "%s the place" % self.name

class Restaurant(models.Model):
    place = models.OneToOneField(
        Place,
        on_delete=models.CASCADE,
        primary_key=True,
    )
    serves_hot_dogs = models.BooleanField(default=False)
    serves_pizza = models.BooleanField(default=False)

    def __str__(self):              # __unicode__ on Python 2
        return "%s the restaurant" % self.place.name

class Waiter(models.Model):
    restaurant = models.ForeignKey(Restaurant, on_delete=models.CASCADE)
    name = models.CharField(max_length=50)
```

```
    def __str__(self):              # __unicode__ on Python 2
        return "%s the waiter at %s" % (self.name, self.restaurant)
```

What follows are examples of operations that can be performed using the Python API facilities.

Create a couple of Places:

```
>>> p1 = Place(name='Demon Dogs', address='944 W. Fullerton')
>>> p1.save()
>>> p2 = Place(name='Ace Hardware', address='1013 N. Ashland')
>>> p2.save()
```

Create a Restaurant. Pass the ID of the "parent" object as this object's ID:

```
>>> r = Restaurant(place=p1, serves_hot_dogs=True, serves_pizza=False)
>>> r.save()
```

A Restaurant can access its place:

```
>>> r.place
<Place: Demon Dogs the place>
```

A Place can access its restaurant, if available:

```
>>> p1.restaurant
<Restaurant: Demon Dogs the restaurant>
```

p2 doesn't have an associated restaurant:

```
>>> from django.core.exceptions import ObjectDoesNotExist
>>> try:
>>>     p2.restaurant
>>> except ObjectDoesNotExist:
>>>     print("There is no restaurant here.")
There is no restaurant here.
```

You can also use `hasattr` to avoid the need for exception catching:

```
>>> hasattr(p2, 'restaurant')
False
```

Set the place using assignment notation. Because place is the primary key on Restaurant, the save will create a new restaurant:

```
>>> r.place = p2
>>> r.save()
>>> p2.restaurant
<Restaurant: Ace Hardware the restaurant>
>>> r.place
<Place: Ace Hardware the place>
```

Set the place back again, using assignment in the reverse direction:

```
>>> p1.restaurant = r
>>> p1.restaurant
<Restaurant: Demon Dogs the restaurant>
```

Note that you must save an object before it can be assigned to a one-to-one relationship. For example, creating a `Restaurant` with unsaved `Place` raises `ValueError`:

```
>>> p3 = Place(name='Demon Dogs', address='944 W. Fullerton')
>>> Restaurant.objects.create(place=p3, serves_hot_dogs=True, serves_pizza=False)
Traceback (most recent call last):
...
ValueError: save() prohibited to prevent data loss due to unsaved related object 'place'.
```

Restaurant.objects.all() just returns the Restaurants, not the Places. Note that there are two restaurants - Ace Hardware the Restaurant was created in the call to r.place = p2:

```
>>> Restaurant.objects.all()
<QuerySet [<Restaurant: Demon Dogs the restaurant>, <Restaurant: Ace Hardware the restaurant>]>
```

Place.objects.all() returns all Places, regardless of whether they have Restaurants:

```
>>> Place.objects.order_by('name')
<QuerySet [<Place: Ace Hardware the place>, <Place: Demon Dogs the place>]>
```

You can query the models using *lookups across relationships*:

```
>>> Restaurant.objects.get(place=p1)
<Restaurant: Demon Dogs the restaurant>
>>> Restaurant.objects.get(place__pk=1)
<Restaurant: Demon Dogs the restaurant>
>>> Restaurant.objects.filter(place__name__startswith="Demon")
<QuerySet [<Restaurant: Demon Dogs the restaurant>]>
>>> Restaurant.objects.exclude(place__address__contains="Ashland")
<QuerySet [<Restaurant: Demon Dogs the restaurant>]>
```

This of course works in reverse:

```
>>> Place.objects.get(pk=1)
<Place: Demon Dogs the place>
>>> Place.objects.get(restaurant__place=p1)
<Place: Demon Dogs the place>
>>> Place.objects.get(restaurant=r)
<Place: Demon Dogs the place>
>>> Place.objects.get(restaurant__place__name__startswith="Demon")
<Place: Demon Dogs the place>
```

Add a Waiter to the Restaurant:

```
>>> w = r.waiter_set.create(name='Joe')
>>> w
<Waiter: Joe the waiter at Demon Dogs the restaurant>
```

Query the waiters:

```
>>> Waiter.objects.filter(restaurant__place=p1)
<QuerySet [<Waiter: Joe the waiter at Demon Dogs the restaurant>]>
>>> Waiter.objects.filter(restaurant__place__name__startswith="Demon")
<QuerySet [<Waiter: Joe the waiter at Demon Dogs the restaurant>]>
```

3.3 Handling HTTP requests

Information on handling HTTP requests in Django:

3.3.1 URL dispatcher

A clean, elegant URL scheme is an important detail in a high-quality Web application. Django lets you design URLs however you want, with no framework limitations.

There's no `.php` or `.cgi` required, and certainly none of that `0,2097,1-1-1928,00` nonsense.

See Cool URIs don't change, by World Wide Web creator Tim Berners-Lee, for excellent arguments on why URLs should be clean and usable.

Overview

To design URLs for an app, you create a Python module informally called a **URLconf** (URL configuration). This module is pure Python code and is a simple mapping between URL patterns (simple regular expressions) to Python functions (your views).

This mapping can be as short or as long as needed. It can reference other mappings. And, because it's pure Python code, it can be constructed dynamically.

Django also provides a way to translate URLs according to the active language. See the *internationalization documentation* for more information.

How Django processes a request

When a user requests a page from your Django-powered site, this is the algorithm the system follows to determine which Python code to execute:

1. Django determines the root URLconf module to use. Ordinarily, this is the value of the *`ROOT_URLCONF`* setting, but if the incoming `HttpRequest` object has a *`urlconf`* attribute (set by middleware), its value will be used in place of the *`ROOT_URLCONF`* setting.
2. Django loads that Python module and looks for the variable `urlpatterns`. This should be a Python list of *`django.conf.urls.url()`* instances.
3. Django runs through each URL pattern, in order, and stops at the first one that matches the requested URL.
4. Once one of the regexes matches, Django imports and calls the given view, which is a simple Python function (or a class-based view). The view gets passed the following arguments:
   - An instance of *`HttpRequest`*.
   - If the matched regular expression returned no named groups, then the matches from the regular expression are provided as positional arguments.
   - The keyword arguments are made up of any named groups matched by the regular expression, overridden by any arguments specified in the optional `kwargs` argument to *`django.conf.urls.url()`*.
5. If no regex matches, or if an exception is raised during any point in this process, Django invokes an appropriate error-handling view. See *Error handling* below.

Example

Here's a sample URLconf:

```
from django.conf.urls import url

from . import views
```

```
urlpatterns = [
    url(r'^articles/2003/$', views.special_case_2003),
    url(r'^articles/([0-9]{4})/$', views.year_archive),
    url(r'^articles/([0-9]{4})/([0-9]{2})/$', views.month_archive),
    url(r'^articles/([0-9]{4})/([0-9]{2})/([0-9]+)/$', views.article_detail),
]
```

Notes:

- To capture a value from the URL, just put parenthesis around it.
- There's no need to add a leading slash, because every URL has that. For example, it's `^articles`, not `^/articles`.
- The `'r'` in front of each regular expression string is optional but recommended. It tells Python that a string is "raw" – that nothing in the string should be escaped. See Dive Into Python's explanation.

Example requests:

- A request to `/articles/2005/03/` would match the third entry in the list. Django would call the function `views.month_archive(request, '2005', '03')`.
- `/articles/2005/3/` would not match any URL patterns, because the third entry in the list requires two digits for the month.
- `/articles/2003/` would match the first pattern in the list, not the second one, because the patterns are tested in order, and the first one is the first test to pass. Feel free to exploit the ordering to insert special cases like this. Here, Django would call the function `views.special_case_2003(request)`
- `/articles/2003` would not match any of these patterns, because each pattern requires that the URL end with a slash.
- `/articles/2003/03/03/` would match the final pattern. Django would call the function `views.article_detail(request, '2003', '03', '03')`.

Named groups

The above example used simple, *non-named* regular-expression groups (via parenthesis) to capture bits of the URL and pass them as *positional* arguments to a view. In more advanced usage, it's possible to use *named* regular-expression groups to capture URL bits and pass them as *keyword* arguments to a view.

In Python regular expressions, the syntax for named regular-expression groups is `(?P<name>pattern)`, where `name` is the name of the group and `pattern` is some pattern to match.

Here's the above example URLconf, rewritten to use named groups:

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^articles/2003/$', views.special_case_2003),
    url(r'^articles/(?P<year>[0-9]{4})/$', views.year_archive),
    url(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/$', views.month_archive),
    url(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<day>[0-9]{2})/$', views.article_detail),
]
```

This accomplishes exactly the same thing as the previous example, with one subtle difference: The captured values are passed to view functions as keyword arguments rather than positional arguments. For example:

- A request to `/articles/2005/03/` would call the function `views.month_archive(request, year='2005', month='03')`, instead of `views.month_archive(request, '2005', '03')`.
- A request to `/articles/2003/03/03/` would call the function `views.article_detail(request, year='2003', month='03', day='03')`.

In practice, this means your URLconfs are slightly more explicit and less prone to argument-order bugs – and you can reorder the arguments in your views' function definitions. Of course, these benefits come at the cost of brevity; some developers find the named-group syntax ugly and too verbose.

The matching/grouping algorithm

Here's the algorithm the URLconf parser follows, with respect to named groups vs. non-named groups in a regular expression:

1. If there are any named arguments, it will use those, ignoring non-named arguments.
2. Otherwise, it will pass all non-named arguments as positional arguments.

In both cases, any extra keyword arguments that have been given as per *Passing extra options to view functions* (below) will also be passed to the view.

What the URLconf searches against

The URLconf searches against the requested URL, as a normal Python string. This does not include GET or POST parameters, or the domain name.

For example, in a request to `https://www.example.com/myapp/`, the URLconf will look for `myapp/`.

In a request to `https://www.example.com/myapp/?page=3`, the URLconf will look for `myapp/`.

The URLconf doesn't look at the request method. In other words, all request methods – `POST`, `GET`, `HEAD`, etc. – will be routed to the same function for the same URL.

Captured arguments are always strings

Each captured argument is sent to the view as a plain Python string, regardless of what sort of match the regular expression makes. For example, in this URLconf line:

```
url(r'^articles/(?P<year>[0-9]{4})/$', views.year_archive),
```

...the `year` argument passed to `views.year_archive()` will be a string, not an integer, even though the `[0-9]{4}` will only match integer strings.

Specifying defaults for view arguments

A convenient trick is to specify default parameters for your views' arguments. Here's an example URLconf and view:

```
# URLconf
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^blog/$', views.page),
    url(r'^blog/page(?P<num>[0-9]+)/$', views.page),
```

```
]

# View (in blog/views.py)
def page(request, num="1"):
    # Output the appropriate page of blog entries, according to num.
    ...
```

In the above example, both URL patterns point to the same view – `views.page` – but the first pattern doesn't capture anything from the URL. If the first pattern matches, the `page()` function will use its default argument for `num`, `"1"`. If the second pattern matches, `page()` will use whatever `num` value was captured by the regex.

Performance

Each regular expression in a `urlpatterns` is compiled the first time it's accessed. This makes the system blazingly fast.

Syntax of the `urlpatterns` variable

`urlpatterns` should be a Python list of *`url()`* instances.

Error handling

When Django can't find a regex matching the requested URL, or when an exception is raised, Django will invoke an error-handling view.

The views to use for these cases are specified by four variables. Their default values should suffice for most projects, but further customization is possible by overriding their default values.

See the documentation on *customizing error views* for the full details.

Such values can be set in your root URLconf. Setting these variables in any other URLconf will have no effect.

Values must be callables, or strings representing the full Python import path to the view that should be called to handle the error condition at hand.

The variables are:

- `handler400` – See *`django.conf.urls.handler400`*.
- `handler403` – See *`django.conf.urls.handler403`*.
- `handler404` – See *`django.conf.urls.handler404`*.
- `handler500` – See *`django.conf.urls.handler500`*.

Including other URLconfs

At any point, your `urlpatterns` can "include" other URLconf modules. This essentially "roots" a set of URLs below other ones.

For example, here's an excerpt of the URLconf for the Django website itself. It includes a number of other URLconfs:

```
from django.conf.urls import include, url

urlpatterns = [
    # ... snip ...
    url(r'^community/', include('django_website.aggregator.urls')),
```

```
    url(r'^contact/', include('django_website.contact.urls')),
    # ... snip ...
]
```

Note that the regular expressions in this example don't have a `$` (end-of-string match character) but do include a trailing slash. Whenever Django encounters `include()` (*`django.conf.urls.include()`*), it chops off whatever part of the URL matched up to that point and sends the remaining string to the included URLconf for further processing.

Another possibility is to include additional URL patterns by using a list of *`url()`* instances. For example, consider this URLconf:

```
from django.conf.urls import include, url

from apps.main import views as main_views
from credit import views as credit_views

extra_patterns = [
    url(r'^reports/$', credit_views.report),
    url(r'^reports/(?P<id>[0-9]+)/$', credit_views.report),
    url(r'^charge/$', credit_views.charge),
]

urlpatterns = [
    url(r'^$', main_views.homepage),
    url(r'^help/', include('apps.help.urls')),
    url(r'^credit/', include(extra_patterns)),
]
```

In this example, the `/credit/reports/` URL will be handled by the `credit_views.report()` Django view.

This can be used to remove redundancy from URLconfs where a single pattern prefix is used repeatedly. For example, consider this URLconf:

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^(?P<page_slug>[\w-]+)-(?P<page_id>\w+)/history/$', views.history),
    url(r'^(?P<page_slug>[\w-]+)-(?P<page_id>\w+)/edit/$', views.edit),
    url(r'^(?P<page_slug>[\w-]+)-(?P<page_id>\w+)/discuss/$', views.discuss),
    url(r'^(?P<page_slug>[\w-]+)-(?P<page_id>\w+)/permissions/$', views.permissions),
]
```

We can improve this by stating the common path prefix only once and grouping the suffixes that differ:

```
from django.conf.urls import include, url
from . import views

urlpatterns = [
    url(r'^(?P<page_slug>[\w-]+)-(?P<page_id>\w+)/', include([
        url(r'^history/$', views.history),
        url(r'^edit/$', views.edit),
        url(r'^discuss/$', views.discuss),
        url(r'^permissions/$', views.permissions),
    ])),
]
```

Captured parameters

An included URLconf receives any captured parameters from parent URLconfs, so the following example is valid:

```
# In settings/urls/main.py
from django.conf.urls import include, url

urlpatterns = [
    url(r'^(?P<username>\w+)/blog/', include('foo.urls.blog')),
]

# In foo/urls/blog.py
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^$', views.blog.index),
    url(r'^archive/$', views.blog.archive),
]
```

In the above example, the captured "`username`" variable is passed to the included URLconf, as expected.

Nested arguments

Regular expressions allow nested arguments, and Django will resolve them and pass them to the view. When reversing, Django will try to fill in all outer captured arguments, ignoring any nested captured arguments. Consider the following URL patterns which optionally take a page argument:

```
from django.conf.urls import url

urlpatterns = [
    url(r'blog/(page-(\d+)/)?$', blog_articles),                  # bad
    url(r'comments/(?:page-(?P<page_number>\d+)/)?$', comments),  # good
]
```

Both patterns use nested arguments and will resolve: for example, `blog/page-2/` will result in a match to `blog_articles` with two positional arguments: `page-2/` and `2`. The second pattern for `comments` will match `comments/page-2/` with keyword argument `page_number` set to 2. The outer argument in this case is a non-capturing argument `(?:...)`.

The `blog_articles` view needs the outermost captured argument to be reversed, `page-2/` or no arguments in this case, while `comments` can be reversed with either no arguments or a value for `page_number`.

Nested captured arguments create a strong coupling between the view arguments and the URL as illustrated by `blog_articles`: the view receives part of the URL (`page-2/`) instead of only the value the view is interested in. This coupling is even more pronounced when reversing, since to reverse the view we need to pass the piece of URL instead of the page number.

As a rule of thumb, only capture the values the view needs to work with and use non-capturing arguments when the regular expression needs an argument but the view ignores it.

Passing extra options to view functions

URLconfs have a hook that lets you pass extra arguments to your view functions, as a Python dictionary.

The *`django.conf.urls.url()`* function can take an optional third argument which should be a dictionary of extra keyword arguments to pass to the view function.

For example:

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^blog/(?P<year>[0-9]{4})/$', views.year_archive, {'foo': 'bar'}),
]
```

In this example, for a request to `/blog/2005/`, Django will call `views.year_archive(request, year='2005', foo='bar')`.

This technique is used in the syndication framework to pass metadata and options to views.

Dealing with conflicts

It's possible to have a URL pattern which captures named keyword arguments, and also passes arguments with the same names in its dictionary of extra arguments. When this happens, the arguments in the dictionary will be used instead of the arguments captured in the URL.

Passing extra options to `include()`

Similarly, you can pass extra options to *`include()`*. When you pass extra options to `include()`, *each* line in the included URLconf will be passed the extra options.

For example, these two URLconf sets are functionally identical:

Set one:

```
# main.py
from django.conf.urls import include, url

urlpatterns = [
    url(r'^blog/', include('inner'), {'blogid': 3}),
]

# inner.py
from django.conf.urls import url
from mysite import views

urlpatterns = [
    url(r'^archive/$', views.archive),
    url(r'^about/$', views.about),
]
```

Set two:

```
# main.py
from django.conf.urls import include, url
from mysite import views

urlpatterns = [
    url(r'^blog/', include('inner')),
]

# inner.py
from django.conf.urls import url
```

```
urlpatterns = [
    url(r'^archive/$', views.archive, {'blogid': 3}),
    url(r'^about/$', views.about, {'blogid': 3}),
]
```

Note that extra options will *always* be passed to *every* line in the included URLconf, regardless of whether the line's view actually accepts those options as valid. For this reason, this technique is only useful if you're certain that every view in the included URLconf accepts the extra options you're passing.

Reverse resolution of URLs

A common need when working on a Django project is the possibility to obtain URLs in their final forms either for embedding in generated content (views and assets URLs, URLs shown to the user, etc.) or for handling of the navigation flow on the server side (redirections, etc.)

It is strongly desirable to avoid hard-coding these URLs (a laborious, non-scalable and error-prone strategy). Equally dangerous is devising ad-hoc mechanisms to generate URLs that are parallel to the design described by the URLconf, which can result in the production of URLs that become stale over time.

In other words, what's needed is a DRY mechanism. Among other advantages it would allow evolution of the URL design without having to go over all the project source code to search and replace outdated URLs.

The primary piece of information we have available to get a URL is an identification (e.g. the name) of the view in charge of handling it. Other pieces of information that necessarily must participate in the lookup of the right URL are the types (positional, keyword) and values of the view arguments.

Django provides a solution such that the URL mapper is the only repository of the URL design. You feed it with your URLconf and then it can be used in both directions:

- Starting with a URL requested by the user/browser, it calls the right Django view providing any arguments it might need with their values as extracted from the URL.
- Starting with the identification of the corresponding Django view plus the values of arguments that would be passed to it, obtain the associated URL.

The first one is the usage we've been discussing in the previous sections. The second one is what is known as *reverse resolution of URLs*, *reverse URL matching*, *reverse URL lookup*, or simply *URL reversing*.

Django provides tools for performing URL reversing that match the different layers where URLs are needed:

- In templates: Using the *url* template tag.
- In Python code: Using the *reverse()* function.
- In higher level code related to handling of URLs of Django model instances: The *get_absolute_url()* method.

Examples

Consider again this URLconf entry:

```
from django.conf.urls import url

from . import views

urlpatterns = [
    #...
    url(r'^articles/([0-9]{4})/$', views.year_archive, name='news-year-archive'),
```

```
    #...
]
```

According to this design, the URL for the archive corresponding to year *nnnn* is `/articles/nnnn/`.

You can obtain these in template code by using:

```
<a href="{% url 'news-year-archive' 2012 %}">2012 Archive</a>
{# Or with the year in a template context variable: #}
<ul>
{% for yearvar in year_list %}
<li><a href="{% url 'news-year-archive' yearvar %}">{{ yearvar }} Archive</a></li>
{% endfor %}
</ul>
```

Or in Python code:

```
from django.urls import reverse
from django.http import HttpResponseRedirect

def redirect_to_year(request):
    # ...
    year = 2006
    # ...
    return HttpResponseRedirect(reverse('news-year-archive', args=(year,)))
```

If, for some reason, it was decided that the URLs where content for yearly article archives are published at should be changed then you would only need to change the entry in the URLconf.

In some scenarios where views are of a generic nature, a many-to-one relationship might exist between URLs and views. For these cases the view name isn't a good enough identifier for it when comes the time of reversing URLs. Read the next section to know about the solution Django provides for this.

Naming URL patterns

In order to perform URL reversing, you'll need to use **named URL patterns** as done in the examples above. The string used for the URL name can contain any characters you like. You are not restricted to valid Python names.

When you name your URL patterns, make sure you use names that are unlikely to clash with any other application's choice of names. If you call your URL pattern `comment`, and another application does the same thing, there's no guarantee which URL will be inserted into your template when you use this name.

Putting a prefix on your URL names, perhaps derived from the application name, will decrease the chances of collision. We recommend something like `myapp-comment` instead of `comment`.

URL namespaces

Introduction

URL namespaces allow you to uniquely reverse *named URL patterns* even if different applications use the same URL names. It's a good practice for third-party apps to always use namespaced URLs (as we did in the tutorial). Similarly, it also allows you to reverse URLs if multiple instances of an application are deployed. In other words, since multiple instances of a single application will share named URLs, namespaces provide a way to tell these named URLs apart.

Django applications that make proper use of URL namespacing can be deployed more than once for a particular site. For example *`django.contrib.admin`* has an *`AdminSite`* class which allows you to easily *deploy more than*

one instance of the admin. In a later example, we'll discuss the idea of deploying the polls application from the tutorial in two different locations so we can serve the same functionality to two different audiences (authors and publishers).

A URL namespace comes in two parts, both of which are strings:

application namespace This describes the name of the application that is being deployed. Every instance of a single application will have the same application namespace. For example, Django's admin application has the somewhat predictable application namespace of `'admin'`.

instance namespace This identifies a specific instance of an application. Instance namespaces should be unique across your entire project. However, an instance namespace can be the same as the application namespace. This is used to specify a default instance of an application. For example, the default Django admin instance has an instance namespace of `'admin'`.

Namespaced URLs are specified using the `':'` operator. For example, the main index page of the admin application is referenced using `'admin:index'`. This indicates a namespace of `'admin'`, and a named URL of `'index'`.

Namespaces can also be nested. The named URL `'sports:polls:index'` would look for a pattern named `'index'` in the namespace `'polls'` that is itself defined within the top-level namespace `'sports'`.

Reversing namespaced URLs

When given a namespaced URL (e.g. `'polls:index'`) to resolve, Django splits the fully qualified name into parts and then tries the following lookup:

1. First, Django looks for a matching *application namespace* (in this example, `'polls'`). This will yield a list of instances of that application.
2. If there is a current application defined, Django finds and returns the URL resolver for that instance. The current application can be specified with the `current_app` argument to the *`reverse()`* function.

   The *`url`* template tag uses the namespace of the currently resolved view as the current application in a *`RequestContext`*. You can override this default by setting the current application on the *`request.current_app`* attribute.

   Previously, the *`url`* template tag did not use the namespace of the currently resolved view and you had to set the `current_app` attribute on the request.
3. If there is no current application. Django looks for a default application instance. The default application instance is the instance that has an *instance namespace* matching the *application namespace* (in this example, an instance of `polls` called `'polls'`).
4. If there is no default application instance, Django will pick the last deployed instance of the application, whatever its instance name may be.
5. If the provided namespace doesn't match an *application namespace* in step 1, Django will attempt a direct lookup of the namespace as an *instance namespace*.

If there are nested namespaces, these steps are repeated for each part of the namespace until only the view name is unresolved. The view name will then be resolved into a URL in the namespace that has been found.

Example To show this resolution strategy in action, consider an example of two instances of the `polls` application from the tutorial: one called `'author-polls'` and one called `'publisher-polls'`. Assume we have enhanced that application so that it takes the instance namespace into consideration when creating and displaying polls.

urls.py

```
from django.conf.urls import include, url

urlpatterns = [
```

```
    url(r'^author-polls/', include('polls.urls', namespace='author-polls')),
    url(r'^publisher-polls/', include('polls.urls', namespace='publisher-polls')),
]
```

polls/urls.py

```
from django.conf.urls import url

from . import views

app_name = 'polls'
urlpatterns = [
    url(r'^$', views.IndexView.as_view(), name='index'),
    url(r'^(?P<pk>\d+)/$', views.DetailView.as_view(), name='detail'),
    ...
]
```

Using this setup, the following lookups are possible:

- If one of the instances is current - say, if we were rendering the detail page in the instance `'author-polls'` - `'polls:index'` will resolve to the index page of the `'author-polls'` instance; i.e. both of the following will result in `"/author-polls/"`.

  In the method of a class-based view:

```
    reverse('polls:index', current_app=self.request.resolver_match.namespace)
```

  and in the template:

```
    {% url 'polls:index' %}
```

- If there is no current instance - say, if we were rendering a page somewhere else on the site - `'polls:index'` will resolve to the last registered instance of `polls`. Since there is no default instance (instance namespace of `'polls'`), the last instance of `polls` that is registered will be used. This would be `'publisher-polls'` since it's declared last in the `urlpatterns`.
- `'author-polls:index'` will always resolve to the index page of the instance `'author-polls'` (and likewise for `'publisher-polls'`) .

If there were also a default instance - i.e., an instance named `'polls'` - the only change from above would be in the case where there is no current instance (the second item in the list above). In this case `'polls:index'` would resolve to the index page of the default instance instead of the instance declared last in `urlpatterns`.

URL namespaces and included URLconfs

Application namespaces of included URLconfs can be specified in two ways.

Firstly, you can set an `app_name` attribute in the included URLconf module, at the same level as the `urlpatterns` attribute. You have to pass the actual module, or a string reference to the module, to *`include()`*, not the list of `urlpatterns` itself.

polls/urls.py

```
from django.conf.urls import url

from . import views

app_name = 'polls'
urlpatterns = [
```

```
    url(r'^$', views.IndexView.as_view(), name='index'),
    url(r'^(?P<pk>\d+)/$', views.DetailView.as_view(), name='detail'),
    ...
]
```

urls.py

```
from django.conf.urls import include, url

urlpatterns = [
    url(r'^polls/', include('polls.urls')),
]
```

The URLs defined in `polls.urls` will have an application namespace `polls`.

Secondly, you can include an object that contains embedded namespace data. If you `include()` a list of *`url()`* instances, the URLs contained in that object will be added to the global namespace. However, you can also `include()` a 2-tuple containing:

```
(<list of url() instances>, <application namespace>)
```

For example:

```
from django.conf.urls import include, url

from . import views

polls_patterns = ([
    url(r'^$', views.IndexView.as_view(), name='index'),
    url(r'^(?P<pk>\d+)/$', views.DetailView.as_view(), name='detail'),
], 'polls')

urlpatterns = [
    url(r'^polls/', include(polls_patterns)),
]
```

This will include the nominated URL patterns into the given application namespace.

The instance namespace can be specified using the `namespace` argument to *`include()`*. If the instance namespace is not specified, it will default to the included URLconf's application namespace. This means it will also be the default instance for that namespace.

In previous versions, you had to specify both the application namespace and the instance namespace in a single place, either by passing them as parameters to *`include()`* or by including a 3-tuple containing `(<list of url() instances>, <application namespace>, <instance namespace>)`.

3.3.2 Writing views

A view function, or *view* for short, is simply a Python function that takes a Web request and returns a Web response. This response can be the HTML contents of a Web page, or a redirect, or a 404 error, or an XML document, or an image . . . or anything, really. The view itself contains whatever arbitrary logic is necessary to return that response. This code can live anywhere you want, as long as it's on your Python path. There's no other requirement–no "magic," so to speak. For the sake of putting the code *somewhere*, the convention is to put views in a file called `views.py`, placed in your project or application directory.

A simple view

Here's a view that returns the current date and time, as an HTML document:

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

Let's step through this code one line at a time:

- First, we import the class *HttpResponse* from the *django.http* module, along with Python's `datetime` library.
- Next, we define a function called `current_datetime`. This is the view function. Each view function takes an *HttpRequest* object as its first parameter, which is typically named `request`.

  Note that the name of the view function doesn't matter; it doesn't have to be named in a certain way in order for Django to recognize it. We're calling it `current_datetime` here, because that name clearly indicates what it does.
- The view returns an *HttpResponse* object that contains the generated response. Each view function is responsible for returning an *HttpResponse* object. (There are exceptions, but we'll get to those later.)

Django's Time Zone

Django includes a *TIME_ZONE* setting that defaults to `America/Chicago`. This probably isn't where you live, so you might want to change it in your settings file.

Mapping URLs to views

So, to recap, this view function returns an HTML page that includes the current date and time. To display this view at a particular URL, you'll need to create a *URLconf*; see URL dispatcher for instructions.

Returning errors

Returning HTTP error codes in Django is easy. There are subclasses of *HttpResponse* for a number of common HTTP status codes other than 200 (which means *"OK"*). You can find the full list of available subclasses in the *request/response* documentation. Just return an instance of one of those subclasses instead of a normal *HttpResponse* in order to signify an error. For example:

```
from django.http import HttpResponse, HttpResponseNotFound

def my_view(request):
    # ...
    if foo:
        return HttpResponseNotFound('<h1>Page not found</h1>')
    else:
        return HttpResponse('<h1>Page was found</h1>')
```

There isn't a specialized subclass for every possible HTTP response code, since many of them aren't going to be that common. However, as documented in the *HttpResponse* documentation, you can also pass the HTTP status code into the constructor for *HttpResponse* to create a return class for any status code you like. For example:

```
from django.http import HttpResponse

def my_view(request):
    # ...

    # Return a "created" (201) response code.
    return HttpResponse(status=201)
```

Because 404 errors are by far the most common HTTP error, there's an easier way to handle those errors.

The `Http404` exception

class `django.http.`**`Http404`**

When you return an error such as *`HttpResponseNotFound`*, you're responsible for defining the HTML of the resulting error page:

```
return HttpResponseNotFound('<h1>Page not found</h1>')
```

For convenience, and because it's a good idea to have a consistent 404 error page across your site, Django provides an `Http404` exception. If you raise `Http404` at any point in a view function, Django will catch it and return the standard error page for your application, along with an HTTP error code 404.

Example usage:

```
from django.http import Http404
from django.shortcuts import render
from polls.models import Poll

def detail(request, poll_id):
    try:
        p = Poll.objects.get(pk=poll_id)
    except Poll.DoesNotExist:
        raise Http404("Poll does not exist")
    return render(request, 'polls/detail.html', {'poll': p})
```

In order to show customized HTML when Django returns a 404, you can create an HTML template named `404.html` and place it in the top level of your template tree. This template will then be served when *`DEBUG`* is set to `False`.

When *`DEBUG`* is `True`, you can provide a message to `Http404` and it will appear in the standard 404 debug template. Use these messages for debugging purposes; they generally aren't suitable for use in a production 404 template.

Customizing error views

The default error views in Django should suffice for most Web applications, but can easily be overridden if you need any custom behavior. Simply specify the handlers as seen below in your URLconf (setting them anywhere else will have no effect).

The *`page_not_found()`* view is overridden by *`handler404`*:

```
handler404 = 'mysite.views.my_custom_page_not_found_view'
```

The *`server_error()`* view is overridden by *`handler500`*:

```
handler500 = 'mysite.views.my_custom_error_view'
```

The *`permission_denied()`* view is overridden by *`handler403`*:

```
handler403 = 'mysite.views.my_custom_permission_denied_view'
```

The *bad_request()* view is overridden by *handler400*:

```
handler400 = 'mysite.views.my_custom_bad_request_view'
```

:

Use the *CSRF_FAILURE_VIEW* setting to override the CSRF error view.

3.3.3 View decorators

Django provides several decorators that can be applied to views to support various HTTP features.

Allowed HTTP methods

The decorators in *django.views.decorators.http* can be used to restrict access to views based on the request method. These decorators will return a *django.http.HttpResponseNotAllowed* if the conditions are not met.

require_http_methods(*request_method_list*)

Decorator to require that a view only accepts particular request methods. Usage:

```
from django.views.decorators.http import require_http_methods

@require_http_methods(["GET", "POST"])
def my_view(request):
    # I can assume now that only GET or POST requests make it this far
    # ...
    pass
```

Note that request methods should be in uppercase.

require_GET()

Decorator to require that a view only accepts the GET method.

require_POST()

Decorator to require that a view only accepts the POST method.

require_safe()

Decorator to require that a view only accepts the GET and HEAD methods. These methods are commonly considered "safe" because they should not have the significance of taking an action other than retrieving the requested resource.

: Web servers should automatically strip the content of responses to HEAD requests while leaving the headers unchanged, so you may handle HEAD requests exactly like GET requests in your views. Since some software, such as link checkers, rely on HEAD requests, you might prefer using `require_safe` instead of `require_GET`.

Conditional view processing

The following decorators in *django.views.decorators.http* can be used to control caching behavior on particular views.

condition(*etag_func=None*, *last_modified_func=None*)

etag(*etag_func*)

last_modified(*last_modified_func*)
These decorators can be used to generate `ETag` and `Last-Modified` headers; see conditional view processing.

GZip compression

The decorators in *`django.views.decorators.gzip`* control content compression on a per-view basis.

gzip_page()
This decorator compresses content if the browser allows gzip compression. It sets the `Vary` header accordingly, so that caches will base their storage on the `Accept-Encoding` header.

Vary headers

The decorators in *`django.views.decorators.vary`* can be used to control caching based on specific request headers.

vary_on_cookie(*func*)

vary_on_headers(**headers*)
The `Vary` header defines which request headers a cache mechanism should take into account when building its cache key.

See *using vary headers*.

Caching

The decorators in *`django.views.decorators.cache`* control server and client-side caching.

cache_control(***kwargs*)
This decorator patches the response's `Cache-Control` header by adding all of the keyword arguments to it. See *`patch_cache_control()`* for the details of the transformation.

never_cache(*view_func*)
This decorator adds a `Cache-Control: max-age=0, no-cache, no-store, must-revalidate` header to a response to indicate that a page should never be cached.

Before Django 1.9, `Cache-Control: max-age=0` was sent. This didn't reliably prevent caching in all browsers.

3.3.4 File Uploads

When Django handles a file upload, the file data ends up placed in *`request.FILES`* (for more on the `request` object see the documentation for request and response objects). This document explains how files are stored on disk and in memory, and how to customize the default behavior.

: There are security risks if you are accepting uploaded content from untrusted users! See the security guide's topic on *User-uploaded content* for mitigation details.

Basic file uploads

Consider a simple form containing a *FileField*:

forms.py

```
from django import forms

class UploadFileForm(forms.Form):
    title = forms.CharField(max_length=50)
    file = forms.FileField()
```

A view handling this form will receive the file data in *request.FILES*, which is a dictionary containing a key for each *FileField* (or *ImageField*, or other *FileField* subclass) in the form. So the data from the above form would be accessible as `request.FILES['file']`.

Note that *request.FILES* will only contain data if the request method was `POST` and the `<form>` that posted the request has the attribute `enctype="multipart/form-data"`. Otherwise, `request.FILES` will be empty.

Most of the time, you'll simply pass the file data from `request` into the form as described in *Binding uploaded files to a form*. This would look something like:

views.py

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import UploadFileForm

# Imaginary function to handle an uploaded file.
from somewhere import handle_uploaded_file

def upload_file(request):
    if request.method == 'POST':
        form = UploadFileForm(request.POST, request.FILES)
        if form.is_valid():
            handle_uploaded_file(request.FILES['file'])
            return HttpResponseRedirect('/success/url/')
    else:
        form = UploadFileForm()
    return render(request, 'upload.html', {'form': form})
```

Notice that we have to pass *request.FILES* into the form's constructor; this is how file data gets bound into a form.

Here's a common way you might handle an uploaded file:

```
def handle_uploaded_file(f):
    with open('some/file/name.txt', 'wb+') as destination:
        for chunk in f.chunks():
            destination.write(chunk)
```

Looping over `UploadedFile.chunks()` instead of using `read()` ensures that large files don't overwhelm your system's memory.

There are a few other methods and attributes available on `UploadedFile` objects; see *UploadedFile* for a complete reference.

Handling uploaded files with a model

If you're saving a file on a *Model* with a *FileField*, using a *ModelForm* makes this process much easier. The file object will be saved to the location specified by the *upload_to* argument of the corresponding *FileField* when calling `form.save()`:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import ModelFormWithFileField

def upload_file(request):
    if request.method == 'POST':
        form = ModelFormWithFileField(request.POST, request.FILES)
        if form.is_valid():
            # file is saved
            form.save()
            return HttpResponseRedirect('/success/url/')
    else:
        form = ModelFormWithFileField()
    return render(request, 'upload.html', {'form': form})
```

If you are constructing an object manually, you can simply assign the file object from *request.FILES* to the file field in the model:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from .forms import UploadFileForm
from .models import ModelWithFileField

def upload_file(request):
    if request.method == 'POST':
        form = UploadFileForm(request.POST, request.FILES)
        if form.is_valid():
            instance = ModelWithFileField(file_field=request.FILES['file'])
            instance.save()
            return HttpResponseRedirect('/success/url/')
    else:
        form = UploadFileForm()
    return render(request, 'upload.html', {'form': form})
```

Uploading multiple files

If you want to upload multiple files using one form field, set the `multiple` HTML attribute of field's widget:

forms.py

```
from django import forms

class FileFieldForm(forms.Form):
    file_field = forms.FileField(widget=forms.ClearableFileInput(attrs={'multiple': True}))
```

Then override the `post` method of your *FormView* subclass to handle multiple file uploads:

views.py

```
from django.views.generic.edit import FormView
from .forms import FileFieldForm
```

```
class FileFieldView(FormView):
    form_class = FileFieldForm
    template_name = 'upload.html'  # Replace with your template.
    success_url = '...'  # Replace with your URL or reverse().

    def post(self, request, *args, **kwargs):
        form_class = self.get_form_class()
        form = self.get_form(form_class)
        files = request.FILES.getlist('file_field')
        if form.is_valid():
            for f in files:
                ...  # Do something with each file.
            return self.form_valid(form)
        else:
            return self.form_invalid(form)
```

Upload Handlers

When a user uploads a file, Django passes off the file data to an *upload handler* – a small class that handles file data as it gets uploaded. Upload handlers are initially defined in the *FILE_UPLOAD_HANDLERS* setting, which defaults to:

```
["django.core.files.uploadhandler.MemoryFileUploadHandler",
 "django.core.files.uploadhandler.TemporaryFileUploadHandler"]
```

Together *MemoryFileUploadHandler* and *TemporaryFileUploadHandler* provide Django's default file upload behavior of reading small files into memory and large ones onto disk.

You can write custom handlers that customize how Django handles files. You could, for example, use custom handlers to enforce user-level quotas, compress data on the fly, render progress bars, and even send data to another storage location directly without storing it locally. See *Writing custom upload handlers* for details on how you can customize or completely replace upload behavior.

Where uploaded data is stored

Before you save uploaded files, the data needs to be stored somewhere.

By default, if an uploaded file is smaller than 2.5 megabytes, Django will hold the entire contents of the upload in memory. This means that saving the file involves only a read from memory and a write to disk and thus is very fast.

However, if an uploaded file is too large, Django will write the uploaded file to a temporary file stored in your system's temporary directory. On a Unix-like platform this means you can expect Django to generate a file called something like `/tmp/tmpzfp6I6.upload`. If an upload is large enough, you can watch this file grow in size as Django streams the data onto disk.

These specifics – 2.5 megabytes; `/tmp`; etc. – are simply "reasonable defaults" which can be customized as described in the next section.

Changing upload handler behavior

There are a few settings which control Django's file upload behavior. See *File Upload Settings* for details.

Modifying upload handlers on the fly

Sometimes particular views require different upload behavior. In these cases, you can override upload handlers on a per-request basis by modifying `request.upload_handlers`. By default, this list will contain the upload handlers given by *FILE_UPLOAD_HANDLERS*, but you can modify the list as you would any other list.

For instance, suppose you've written a `ProgressBarUploadHandler` that provides feedback on upload progress to some sort of AJAX widget. You'd add this handler to your upload handlers like this:

```
request.upload_handlers.insert(0, ProgressBarUploadHandler(request))
```

You'd probably want to use `list.insert()` in this case (instead of `append()`) because a progress bar handler would need to run *before* any other handlers. Remember, the upload handlers are processed in order.

If you want to replace the upload handlers completely, you can just assign a new list:

```
request.upload_handlers = [ProgressBarUploadHandler(request)]
```

: You can only modify upload handlers *before* accessing `request.POST` or `request.FILES` – it doesn't make sense to change upload handlers after upload handling has already started. If you try to modify `request.upload_handlers` after reading from `request.POST` or `request.FILES` Django will throw an error.

Thus, you should always modify uploading handlers as early in your view as possible.

Also, `request.POST` is accessed by *CsrfViewMiddleware* which is enabled by default. This means you will need to use *csrf_exempt()* on your view to allow you to change the upload handlers. You will then need to use *csrf_protect()* on the function that actually processes the request. Note that this means that the handlers may start receiving the file upload before the CSRF checks have been done. Example code:

```
from django.views.decorators.csrf import csrf_exempt, csrf_protect

@csrf_exempt
def upload_file_view(request):
    request.upload_handlers.insert(0, ProgressBarUploadHandler(request))
    return _upload_file_view(request)

@csrf_protect
def _upload_file_view(request):
    ... # Process request
```

3.3.5 Django shortcut functions

The package `django.shortcuts` collects helper functions and classes that "span" multiple levels of MVC. In other words, these functions/classes introduce controlled coupling for convenience's sake.

render()

render(*request*, *template_name*, *context=None*, *content_type=None*, *status=None*, *using=None*)

Combines a given template with a given context dictionary and returns an *HttpResponse* object with that rendered text.

Django does not provide a shortcut function which returns a *TemplateResponse* because the constructor of *TemplateResponse* offers the same level of convenience as *render()*.

Required arguments

request The request object used to generate this response.

template_name The full name of a template to use or sequence of template names. If a sequence is given, the first template that exists will be used. See the *template loading documentation* for more information on how templates are found.

Optional arguments

context A dictionary of values to add to the template context. By default, this is an empty dictionary. If a value in the dictionary is callable, the view will call it just before rendering the template.

content_type The MIME type to use for the resulting document. Defaults to the value of the *DEFAULT_CONTENT_TYPE* setting.

status The status code for the response. Defaults to `200`.

using The *NAME* of a template engine to use for loading the template.

Example

The following example renders the template `myapp/index.html` with the MIME type *application/xhtml+xml*:

```
from django.shortcuts import render

def my_view(request):
    # View code here...
    return render(request, 'myapp/index.html', {
        'foo': 'bar',
    }, content_type='application/xhtml+xml')
```

This example is equivalent to:

```
from django.http import HttpResponse
from django.template import loader

def my_view(request):
    # View code here...
    t = loader.get_template('myapp/index.html')
    c = {'foo': 'bar'}
    return HttpResponse(t.render(c, request), content_type='application/xhtml+xml')
```

render_to_response()

render_to_response(*template_name*, *context=None*, *content_type=None*, *status=None*, *using=None*)
This function preceded the introduction of *render()* and works similarly except that it doesn't make the `request` available in the response. It's not recommended and is likely to be deprecated in the future.

redirect()

redirect(*to*, *permanent=False*, **args*, ***kwargs*)
Returns an *HttpResponseRedirect* to the appropriate URL for the arguments passed.

The arguments could be:

- A model: the model's *get_absolute_url()* function will be called.
- A view name, possibly with arguments: *reverse()* will be used to reverse-resolve the name.
- An absolute or relative URL, which will be used as-is for the redirect location.

By default issues a temporary redirect; pass `permanent=True` to issue a permanent redirect.

Examples

You can use the *redirect()* function in a number of ways.

1. By passing some object; that object's *get_absolute_url()* method will be called to figure out the redirect URL:

```
from django.shortcuts import redirect

def my_view(request):
    ...
    object = MyModel.objects.get(...)
    return redirect(object)
```

2. By passing the name of a view and optionally some positional or keyword arguments; the URL will be reverse resolved using the *reverse()* method:

```
def my_view(request):
    ...
    return redirect('some-view-name', foo='bar')
```

3. By passing a hardcoded URL to redirect to:

```
def my_view(request):
    ...
    return redirect('/some/url/')
```

This also works with full URLs:

```
def my_view(request):
    ...
    return redirect('https://example.com/')
```

By default, *redirect()* returns a temporary redirect. All of the above forms accept a `permanent` argument; if set to `True` a permanent redirect will be returned:

```
def my_view(request):
    ...
    object = MyModel.objects.get(...)
    return redirect(object, permanent=True)
```

get_object_or_404()

get_object_or_404(*klass*, **args*, ***kwargs*)

Calls *get()* on a given model manager, but it raises *Http404* instead of the model's *DoesNotExist* exception.

Required arguments

klass A *Model* class, a *Manager*, or a *QuerySet* instance from which to get the object.

****kwargs** Lookup parameters, which should be in the format accepted by `get()` and `filter()`.

Example

The following example gets the object with the primary key of 1 from `MyModel`:

```
from django.shortcuts import get_object_or_404

def my_view(request):
    my_object = get_object_or_404(MyModel, pk=1)
```

This example is equivalent to:

```
from django.http import Http404

def my_view(request):
    try:
        my_object = MyModel.objects.get(pk=1)
    except MyModel.DoesNotExist:
        raise Http404("No MyModel matches the given query.")
```

The most common use case is to pass a *Model*, as shown above. However, you can also pass a *QuerySet* instance:

```
queryset = Book.objects.filter(title__startswith='M')
get_object_or_404(queryset, pk=1)
```

The above example is a bit contrived since it's equivalent to doing:

```
get_object_or_404(Book, title__startswith='M', pk=1)
```

but it can be useful if you are passed the `queryset` variable from somewhere else.

Finally, you can also use a *Manager*. This is useful for example if you have a *custom manager*:

```
get_object_or_404(Book.dahl_objects, title='Matilda')
```

You can also use *related managers*:

```
author = Author.objects.get(name='Roald Dahl')
get_object_or_404(author.book_set, title='Matilda')
```

Note: As with `get()`, a *MultipleObjectsReturned* exception will be raised if more than one object is found.

get_list_or_404()

get_list_or_404(*klass*, **args*, ***kwargs*)
Returns the result of *filter()* on a given model manager cast to a list, raising *Http404* if the resulting list is empty.

Required arguments

klass A *Model*, *Manager* or *QuerySet* instance from which to get the list.

****kwargs** Lookup parameters, which should be in the format accepted by `get()` and `filter()`.

Example

The following example gets all published objects from `MyModel`:

```
from django.shortcuts import get_list_or_404

def my_view(request):
    my_objects = get_list_or_404(MyModel, published=True)
```

This example is equivalent to:

```
from django.http import Http404

def my_view(request):
    my_objects = list(MyModel.objects.filter(published=True))
    if not my_objects:
        raise Http404("No MyModel matches the given query.")
```

3.3.6 Generic views

See Built-in class-based views API.

3.3.7 Middleware

Middleware is a framework of hooks into Django's request/response processing. It's a light, low-level "plugin" system for globally altering Django's input or output.

Each middleware component is responsible for doing some specific function. For example, Django includes a middleware component, *`AuthenticationMiddleware`*, that associates users with requests using sessions.

This document explains how middleware works, how you activate middleware, and how to write your own middleware. Django ships with some built-in middleware you can use right out of the box. They're documented in the built-in middleware reference.

A new style of middleware was introduced for use with the new *`MIDDLEWARE`* setting. If you're using the old *`MIDDLEWARE_CLASSES`* setting, you'll need to *adapt old, custom middleware* before using the new setting. This document describes new-style middleware. Refer to this page in older versions of the documentation for a description of how old-style middleware works.

Writing your own middleware

A middleware factory is a callable that takes a `get_response` callable and returns a middleware. A middleware is a callable that takes a request and returns a response, just like a view.

A middleware can be written as a function that looks like this:

```
def simple_middleware(get_response):
    # One-time configuration and initialization.

    def middleware(request):
        # Code to be executed for each request before
        # the view (and later middleware) are called.
```

```
        response = get_response(request)

        # Code to be executed for each request/response after
        # the view is called.

        return response

    return middleware
```

Or it can be written as a class whose instances are callable, like this:

```
class SimpleMiddleware(object):
    def __init__(self, get_response):
        self.get_response = get_response
        # One-time configuration and initialization.

    def __call__(self, request):
        # Code to be executed for each request before
        # the view (and later middleware) are called.

        response = self.get_response(request)

        # Code to be executed for each request/response after
        # the view is called.

        return response
```

The `get_response` callable provided by Django might be the actual view (if this is the last listed middleware) or it might be the next middleware in the chain. The current middleware doesn't need to know or care what exactly it is, just that it represents whatever comes next.

The above is a slight simplification – the `get_response` callable for the last middleware in the chain won't be the actual view but rather a wrapper method from the handler which takes care of applying *view middleware*, calling the view with appropriate URL arguments, and applying *template-response* and *exception* middleware.

Middleware can live anywhere on your Python path.

`__init__(get_response)`

Middleware factories must accept a `get_response` argument. You can also initialize some global state for the middleware. Keep in mind a couple of caveats:

- Django initializes your middleware with only the `get_response` argument, so you can't define `__init__()` as requiring any other arguments.
- Unlike the `__call__()` method which is called once per request, `__init__()` is called only *once*, when the Web server starts.

In older versions, `__init__()` wasn't called until the Web server responded to its first request.

In older versions, `__init__()` didn't accept any arguments. To allow your middleware to be used in Django 1.9 and earlier, make `get_response` an optional argument (`get_response=None`).

Marking middleware as unused

It's sometimes useful to determine at startup time whether a piece of middleware should be used. In these cases, your middleware's `__init__()` method may raise *`MiddlewareNotUsed`*. Django will then remove that middleware from the middleware process and log a debug message to the *django.request* logger when *`DEBUG`* is `True`.

Activating middleware

To activate a middleware component, add it to the *MIDDLEWARE* list in your Django settings.

In *MIDDLEWARE*, each middleware component is represented by a string: the full Python path to the middleware factory's class or function name. For example, here's the default value created by *django-admin startproject*:

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

A Django installation doesn't require any middleware — *MIDDLEWARE* can be empty, if you'd like — but it's strongly suggested that you at least use *CommonMiddleware*.

The order in *MIDDLEWARE* matters because a middleware can depend on other middleware. For instance, *AuthenticationMiddleware* stores the authenticated user in the session; therefore, it must run after *SessionMiddleware*. See *Middleware ordering* for some common hints about ordering of Django middleware classes.

Middleware order and layering

During the request phase, before calling the view, Django applies middleware in the order it's defined in *MIDDLEWARE*, top-down.

You can think of it like an onion: each middleware class is a "layer" that wraps the view, which is in the core of the onion. If the request passes through all the layers of the onion (each one calls `get_response` to pass the request in to the next layer), all the way to the view at the core, the response will then pass through every layer (in reverse order) on the way back out.

If one of the layers decides to short-circuit and return a response without ever calling its `get_response`, none of the layers of the onion inside that layer (including the view) will see the request or the response. The response will only return through the same layers that the request passed in through.

Other middleware hooks

Besides the basic request/response middleware pattern described earlier, you can add three other special methods to class-based middleware:

process_view()

process_view(*request*, *view_func*, *view_args*, *view_kwargs*)

`request` is an *HttpRequest* object. `view_func` is the Python function that Django is about to use. (It's the actual function object, not the name of the function as a string.) `view_args` is a list of positional arguments that will be passed to the view, and `view_kwargs` is a dictionary of keyword arguments that will be passed to the view. Neither `view_args` nor `view_kwargs` include the first view argument (`request`).

`process_view()` is called just before Django calls the view.

It should return either `None` or an *`HttpResponse`* object. If it returns `None`, Django will continue processing this request, executing any other `process_view()` middleware and, then, the appropriate view. If it returns an *`HttpResponse`* object, Django won't bother calling the appropriate view; it'll apply response middleware to that *`HttpResponse`* and return the result.

: Accessing *`request.POST`* inside middleware before the view runs or in `process_view()` will prevent any view running after the middleware from being able to *modify the upload handlers for the request*, and should normally be avoided.

The *`CsrfViewMiddleware`* class can be considered an exception, as it provides the *`csrf_exempt()`* and *`csrf_protect()`* decorators which allow views to explicitly control at what point the CSRF validation should occur.

---

process_exception()

process_exception(*request*, *exception*)

`request` is an *`HttpRequest`* object. `exception` is an `Exception` object raised by the view function.

Django calls `process_exception()` when a view raises an exception. `process_exception()` should return either `None` or an *`HttpResponse`* object. If it returns an *`HttpResponse`* object, the template response and response middleware will be applied and the resulting response returned to the browser. Otherwise, *default exception handling* kicks in.

Again, middleware are run in reverse order during the response phase, which includes `process_exception`. If an exception middleware returns a response, the `process_exception` methods of the middleware classes above that middleware won't be called at all.

process_template_response()

process_template_response(*request*, *response*)

`request` is an *`HttpRequest`* object. `response` is the *`TemplateResponse`* object (or equivalent) returned by a Django view or by a middleware.

`process_template_response()` is called just after the view has finished executing, if the response instance has a `render()` method, indicating that it is a *`TemplateResponse`* or equivalent.

It must return a response object that implements a `render` method. It could alter the given `response` by changing `response.template_name` and `response.context_data`, or it could create and return a brand-new *`TemplateResponse`* or equivalent.

You don't need to explicitly render responses – responses will be automatically rendered once all template response middleware has been called.

Middleware are run in reverse order during the response phase, which includes `process_template_response()`.

Dealing with streaming responses

Unlike *`HttpResponse`*, *`StreamingHttpResponse`* does not have a `content` attribute. As a result, middleware can no longer assume that all responses will have a `content` attribute. If they need access to the content, they must test for streaming responses and adjust their behavior accordingly:

```
if response.streaming:
    response.streaming_content = wrap_streaming_content(response.streaming_content)
else:
    response.content = alter_content(response.content)
```

: `streaming_content` should be assumed to be too large to hold in memory. Response middleware may wrap it in a new generator, but must not consume it. Wrapping is typically implemented as follows:

```
def wrap_streaming_content(content):
    for chunk in content:
        yield alter_content(chunk)
```

Exception handling

Django automatically converts exceptions raised by the view or by middleware into an appropriate HTTP response with an error status code. *Certain exceptions* are converted to 4xx status codes, while an unknown exception is converted to a 500 status code.

This conversion takes place before and after each middleware (you can think of it as the thin film in between each layer of the onion), so that every middleware can always rely on getting some kind of HTTP response back from calling its `get_response` callable. Middleware don't need to worry about wrapping their call to `get_response` in a `try/except` and handling an exception that might have been raised by a later middleware or the view. Even if the very next middleware in the chain raises an *`Http404`* exception, for example, your middleware won't see that exception; instead it will get an *`HttpResponse`* object with a *`status_code`* of 404.

Upgrading pre-Django 1.10-style middleware

class `django.utils.deprecation.`**`MiddlewareMixin`**

Django provides `django.utils.deprecation.MiddlewareMixin` to ease creating middleware classes that are compatible with both *`MIDDLEWARE`* and the old *`MIDDLEWARE_CLASSES`*. All middleware classes included with Django are compatible with both settings.

The mixin provides an `__init__()` method that accepts an optional `get_response` argument and stores it in `self.get_response`.

The `__call__()` method:

1. Calls `self.process_request(request)` (if defined).
2. Calls `self.get_response(request)` to get the response from later middleware and the view.
3. Calls `self.process_response(request, response)` (if defined).
4. Returns the response.

If used with *`MIDDLEWARE_CLASSES`*, the `__call__()` method will never be used; Django calls `process_request()` and `process_response()` directly.

In most cases, inheriting from this mixin will be sufficient to make an old-style middleware compatible with the new system with sufficient backwards-compatibility. The new short-circuiting semantics will be harmless or even beneficial to the existing middleware. In a few cases, a middleware class may need some changes to adjust to the new semantics.

These are the behavioral differences between using *`MIDDLEWARE`* and *`MIDDLEWARE_CLASSES`*:

1. Under *MIDDLEWARE_CLASSES*, every middleware will always have its `process_response` method called, even if an earlier middleware short-circuited by returning a response from its `process_request` method. Under *MIDDLEWARE*, middleware behaves more like an onion: the layers that a response goes through on the way out are the same layers that saw the request on the way in. If a middleware short-circuits, only that middleware and the ones before it in *MIDDLEWARE* will see the response.
2. Under *MIDDLEWARE_CLASSES*, `process_exception` is applied to exceptions raised from a middleware `process_request` method. Under *MIDDLEWARE*, `process_exception` applies only to exceptions raised from the view (or from the `render` method of a *TemplateResponse*). Exceptions raised from a middleware are converted to the appropriate HTTP response and then passed to the next middleware.
3. Under *MIDDLEWARE_CLASSES*, if a `process_response` method raises an exception, the `process_response` methods of all earlier middleware are skipped and a `500 Internal Server Error` HTTP response is always returned (even if the exception raised was e.g. an *Http404*). Under *MIDDLEWARE*, an exception raised from a middleware will immediately be converted to the appropriate HTTP response, and then the next middleware in line will see that response. Middleware are never skipped due to a middleware raising an exception.

3.3.8 How to use sessions

Django provides full support for anonymous sessions. The session framework lets you store and retrieve arbitrary data on a per-site-visitor basis. It stores data on the server side and abstracts the sending and receiving of cookies. Cookies contain a session ID – not the data itself (unless you're using the *cookie based backend*).

Enabling sessions

Sessions are implemented via a piece of middleware.

To enable session functionality, do the following:

- Edit the *MIDDLEWARE* setting and make sure it contains `'django.contrib.sessions.middleware.SessionMiddle`
  The default `settings.py` created by `django-admin startproject` has `SessionMiddleware` activated.

If you don't want to use sessions, you might as well remove the `SessionMiddleware` line from *MIDDLEWARE* and `'django.contrib.sessions'` from your *INSTALLED_APPS*. It'll save you a small bit of overhead.

Configuring the session engine

By default, Django stores sessions in your database (using the model `django.contrib.sessions.models.Session`). Though this is convenient, in some setups it's faster to store session data elsewhere, so Django can be configured to store session data on your filesystem or in your cache.

Using database-backed sessions

If you want to use a database-backed session, you need to add `'django.contrib.sessions'` to your *INSTALLED_APPS* setting.

Once you have configured your installation, run `manage.py migrate` to install the single database table that stores session data.

Using cached sessions

For better performance, you may want to use a cache-based session backend.

To store session data using Django's cache system, you'll first need to make sure you've configured your cache; see the cache documentation for details.

: You should only use cache-based sessions if you're using the Memcached cache backend. The local-memory cache backend doesn't retain data long enough to be a good choice, and it'll be faster to use file or database sessions directly instead of sending everything through the file or database cache backends. Additionally, the local-memory cache backend is NOT multi-process safe, therefore probably not a good choice for production environments.

If you have multiple caches defined in *CACHES*, Django will use the default cache. To use another cache, set *SESSION_CACHE_ALIAS* to the name of that cache.

Once your cache is configured, you've got two choices for how to store data in the cache:

- Set *SESSION_ENGINE* to `"django.contrib.sessions.backends.cache"` for a simple caching session store. Session data will be stored directly in your cache. However, session data may not be persistent: cached data can be evicted if the cache fills up or if the cache server is restarted.
- For persistent, cached data, set *SESSION_ENGINE* to `"django.contrib.sessions.backends.cached_db"`. This uses a write-through cache – every write to the cache will also be written to the database. Session reads only use the database if the data is not already in the cache.

Both session stores are quite fast, but the simple cache is faster because it disregards persistence. In most cases, the `cached_db` backend will be fast enough, but if you need that last bit of performance, and are willing to let session data be expunged from time to time, the `cache` backend is for you.

If you use the `cached_db` session backend, you also need to follow the configuration instructions for the *using database-backed sessions*.

Using file-based sessions

To use file-based sessions, set the *SESSION_ENGINE* setting to `"django.contrib.sessions.backends.file"`.

You might also want to set the *SESSION_FILE_PATH* setting (which defaults to output from `tempfile.gettempdir()`, most likely `/tmp`) to control where Django stores session files. Be sure to check that your Web server has permissions to read and write to this location.

Using cookie-based sessions

To use cookies-based sessions, set the *SESSION_ENGINE* setting to `"django.contrib.sessions.backends.signed_cook`
The session data will be stored using Django's tools for cryptographic signing and the *SECRET_KEY* setting.

: It's recommended to leave the *SESSION_COOKIE_HTTPONLY* setting on `True` to prevent access to the stored data from JavaScript.

: If the SECRET_KEY is not kept secret and you are using the *PickleSerializer*, this can lead to arbitrary remote code execution.
An attacker in possession of the *SECRET_KEY* can not only generate falsified session data, which your site will trust, but also remotely execute arbitrary code, as the data is serialized using pickle.
If you use cookie-based sessions, pay extra care that your secret key is always kept completely secret, for any system which might be remotely accessible.
The session data is signed but not encrypted
When using the cookies backend the session data can be read by the client.
A MAC (Message Authentication Code) is used to protect the data against changes by the client, so that the session data will be invalidated when being tampered with. The same invalidation happens if the client storing the cookie (e.g. your user's browser) can't store all of the session cookie and drops data. Even though Django compresses the data, it's still entirely possible to exceed the common limit of 4096 bytes per cookie.
No freshness guarantee
Note also that while the MAC can guarantee the authenticity of the data (that it was generated by your site, and not someone else), and the integrity of the data (that it is all there and correct), it cannot guarantee freshness i.e. that you are being sent back the last thing you sent to the client. This means that for some uses of session data, the cookie backend might open you up to replay attacks. Unlike other session backends which keep a server-side record of each session and invalidate it when a user logs out, cookie-based sessions are not invalidated when a user logs out. Thus if an attacker steals a user's cookie, they can use that cookie to login as that user even if the user logs out. Cookies will only be detected as 'stale' if they are older than your *SESSION_COOKIE_AGE*.
Performance
Finally, the size of a cookie can have an impact on the speed of your site.

Using sessions in views

When `SessionMiddleware` is activated, each *HttpRequest* object – the first argument to any Django view function – will have a `session` attribute, which is a dictionary-like object.

You can read it and write to `request.session` at any point in your view. You can edit it multiple times.

class `backends.base.`**SessionBase**
This is the base class for all session objects. It has the following standard dictionary methods:

__getitem__(*key*)
Example: `fav_color = request.session['fav_color']`

__setitem__(*key*, *value*)
Example: `request.session['fav_color'] = 'blue'`

__delitem__(*key*)
Example: `del request.session['fav_color']`. This raises `KeyError` if the given `key` isn't already in the session.

__contains__(*key*)
Example: `'fav_color' in request.session`

get(*key*, *default=None*)
Example: `fav_color = request.session.get('fav_color', 'red')`

pop(*key*, *default=__not_given*)
Example: `fav_color = request.session.pop('fav_color', 'blue')`

keys()

items()

setdefault()

clear()

It also has these methods:

flush()
Deletes the current session data from the session and deletes the session cookie. This is used if you want to ensure that the previous session data can't be accessed again from the user's browser (for example, the *django.contrib.auth.logout()* function calls it).

set_test_cookie()
Sets a test cookie to determine whether the user's browser supports cookies. Due to the way cookies work, you won't be able to test this until the user's next page request. See *Setting test cookies* below for more information.

test_cookie_worked()
Returns either `True` or `False`, depending on whether the user's browser accepted the test cookie. Due to the way cookies work, you'll have to call `set_test_cookie()` on a previous, separate page request. See *Setting test cookies* below for more information.

delete_test_cookie()
Deletes the test cookie. Use this to clean up after yourself.

set_expiry(*value*)
Sets the expiration time for the session. You can pass a number of different values:

- If `value` is an integer, the session will expire after that many seconds of inactivity. For example, calling `request.session.set_expiry(300)` would make the session expire in 5 minutes.
- If `value` is a `datetime` or `timedelta` object, the session will expire at that specific date/time. Note that `datetime` and `timedelta` values are only serializable if you are using the *PickleSerializer*.
- If `value` is `0`, the user's session cookie will expire when the user's Web browser is closed.
- If `value` is `None`, the session reverts to using the global session expiry policy.

Reading a session is not considered activity for expiration purposes. Session expiration is computed from the last time the session was *modified*.

get_expiry_age()
Returns the number of seconds until this session expires. For sessions with no custom expiration (or those set to expire at browser close), this will equal *SESSION_COOKIE_AGE*.

This function accepts two optional keyword arguments:

- `modification`: last modification of the session, as a `datetime` object. Defaults to the current time.
- `expiry`: expiry information for the session, as a `datetime` object, an `int` (in seconds), or `None`. Defaults to the value stored in the session by *set_expiry()*, if there is one, or `None`.

get_expiry_date()
Returns the date this session will expire. For sessions with no custom expiration (or those set to expire at browser close), this will equal the date *SESSION_COOKIE_AGE* seconds from now.

This function accepts the same keyword arguments as *get_expiry_age()*.

get_expire_at_browser_close()
Returns either `True` or `False`, depending on whether the user's session cookie will expire when the user's Web browser is closed.

clear_expired()
Removes expired sessions from the session store. This class method is called by *clearsessions*.

cycle_key()
Creates a new session key while retaining the current session data. *django.contrib.auth.login()* calls this method to mitigate against session fixation.

Session serialization

By default, Django serializes session data using JSON. You can use the *SESSION_SERIALIZER* setting to customize the session serialization format. Even with the caveats described in *Write your own serializer*, we highly recommend sticking with JSON serialization *especially if you are using the cookie backend*.

For example, here's an attack scenario if you use `pickle` to serialize session data. If you're using the *signed cookie session backend* and *SECRET_KEY* is known by an attacker (there isn't an inherent vulnerability in Django that would cause it to leak), the attacker could insert a string into their session which, when unpickled, executes arbitrary code on the server. The technique for doing so is simple and easily available on the internet. Although the cookie session storage signs the cookie-stored data to prevent tampering, a *SECRET_KEY* leak immediately escalates to a remote code execution vulnerability.

Bundled serializers

class `serializers.`**JSONSerializer**
A wrapper around the JSON serializer from *django.core.signing*. Can only serialize basic data types.

In addition, as JSON supports only string keys, note that using non-string keys in `request.session` won't work as expected:

```
>>> # initial assignment
>>> request.session[0] = 'bar'
>>> # subsequent requests following serialization & deserialization
>>> # of session data
>>> request.session[0]  # KeyError
>>> request.session['0']
'bar'
```

Similarly, data that can't be encoded in JSON, such as non-UTF8 bytes like `'\xd9'` (which raises `UnicodeDecodeError`), can't be stored.

See the *Write your own serializer* section for more details on limitations of JSON serialization.

class `serializers.`**PickleSerializer**
Supports arbitrary Python objects, but, as described above, can lead to a remote code execution vulnerability if *SECRET_KEY* becomes known by an attacker.

Write your own serializer Note that unlike *PickleSerializer*, the *JSONSerializer* cannot handle arbitrary Python data types. As is often the case, there is a trade-off between convenience and security. If you wish to store more advanced data types including `datetime` and `Decimal` in JSON backed sessions, you will need to write a custom serializer (or convert such values to a JSON serializable object before storing them in `request.session`). While serializing these values is fairly straightforward (*DjangoJSONEncoder* may be helpful), writing a decoder that can reliably get back the same thing that you put in is more fragile. For example, you run the risk of returning a `datetime` that was actually a string that just happened to be in the same format chosen for `datetimes`).

Your serializer class must implement two methods, `dumps(self, obj)` and `loads(self, data)`, to serialize and deserialize the dictionary of session data, respectively.

Session object guidelines

- Use normal Python strings as dictionary keys on `request.session`. This is more of a convention than a hard-and-fast rule.
- Session dictionary keys that begin with an underscore are reserved for internal use by Django.
- Don't override `request.session` with a new object, and don't access or set its attributes. Use it like a Python dictionary.

Examples

This simplistic view sets a `has_commented` variable to `True` after a user posts a comment. It doesn't let a user post a comment more than once:

```
def post_comment(request, new_comment):
    if request.session.get('has_commented', False):
        return HttpResponse("You've already commented.")
    c = comments.Comment(comment=new_comment)
    c.save()
    request.session['has_commented'] = True
    return HttpResponse('Thanks for your comment!')
```

This simplistic view logs in a "member" of the site:

```
def login(request):
    m = Member.objects.get(username=request.POST['username'])
    if m.password == request.POST['password']:
        request.session['member_id'] = m.id
        return HttpResponse("You're logged in.")
    else:
        return HttpResponse("Your username and password didn't match.")
```

...And this one logs a member out, according to `login()` above:

```
def logout(request):
    try:
        del request.session['member_id']
    except KeyError:
        pass
    return HttpResponse("You're logged out.")
```

The standard *`django.contrib.auth.logout()`* function actually does a bit more than this to prevent inadvertent data leakage. It calls the *`flush()`* method of `request.session`. We are using this example as a demonstration of how to work with session objects, not as a full `logout()` implementation.

Setting test cookies

As a convenience, Django provides an easy way to test whether the user's browser accepts cookies. Just call the *`set_test_cookie()`* method of `request.session` in a view, and call *`test_cookie_worked()`* in a subsequent view – not in the same view call.

This awkward split between `set_test_cookie()` and `test_cookie_worked()` is necessary due to the way cookies work. When you set a cookie, you can't actually tell whether a browser accepted it until the browser's next request.

It's good practice to use *delete_test_cookie()* to clean up after yourself. Do this after you've verified that the test cookie worked.

Here's a typical usage example:

```
from django.http import HttpResponse
from django.shortcuts import render

def login(request):
    if request.method == 'POST':
        if request.session.test_cookie_worked():
            request.session.delete_test_cookie()
            return HttpResponse("You're logged in.")
        else:
            return HttpResponse("Please enable cookies and try again.")
    request.session.set_test_cookie()
    return render(request, 'foo/login_form.html')
```

Using sessions out of views

: The examples in this section import the `SessionStore` object directly from the `django.contrib.sessions.backends.db` backend. In your own code, you should consider importing `SessionStore` from the session engine designated by *SESSION_ENGINE*, as below:

```
>>> from importlib import import_module
>>> from django.conf import settings
>>> SessionStore = import_module(settings.SESSION_ENGINE).SessionStore
```

An API is available to manipulate session data outside of a view:

```
>>> from django.contrib.sessions.backends.db import SessionStore
>>> s = SessionStore()
>>> # stored as seconds since epoch since datetimes are not serializable in JSON.
>>> s['last_login'] = 1376587691
>>> s.create()
>>> s.session_key
'2b1189a188b44ad18c35e113ac6ceead'
>>> s = SessionStore(session_key='2b1189a188b44ad18c35e113ac6ceead')
>>> s['last_login']
1376587691
```

`SessionStore.create()` is designed to create a new session (i.e. one not loaded from the session store and with `session_key=None`). `save()` is designed to save an existing session (i.e. one loaded from the session store). Calling `save()` on a new session may also work but has a small chance of generating a `session_key` that collides with an existing one. `create()` calls `save()` and loops until an unused `session_key` is generated.

If you're using the `django.contrib.sessions.backends.db` backend, each session is just a normal Django model. The `Session` model is defined in `django/contrib/sessions/models.py`. Because it's a normal model, you can access sessions using the normal Django database API:

```
>>> from django.contrib.sessions.models import Session
>>> s = Session.objects.get(pk='2b1189a188b44ad18c35e113ac6ceead')
>>> s.expire_date
datetime.datetime(2005, 8, 20, 13, 35, 12)
```

Note that you'll need to call *get_decoded()* to get the session dictionary. This is necessary because the dictionary is stored in an encoded format:

```
>>> s.session_data
'KGRwMQpTJ19hdXRoX3VzZXJfaWQnCnAyCkkxCnMuMTExY2ZjODI2Yj...'
>>> s.get_decoded()
{'user_id': 42}
```

When sessions are saved

By default, Django only saves to the session database when the session has been modified – that is if any of its dictionary values have been assigned or deleted:

```
# Session is modified.
request.session['foo'] = 'bar'

# Session is modified.
del request.session['foo']

# Session is modified.
request.session['foo'] = {}

# Gotcha: Session is NOT modified, because this alters
# request.session['foo'] instead of request.session.
request.session['foo']['bar'] = 'baz'
```

In the last case of the above example, we can tell the session object explicitly that it has been modified by setting the `modified` attribute on the session object:

```
request.session.modified = True
```

To change this default behavior, set the *SESSION_SAVE_EVERY_REQUEST* setting to `True`. When set to `True`, Django will save the session to the database on every single request.

Note that the session cookie is only sent when a session has been created or modified. If *SESSION_SAVE_EVERY_REQUEST* is `True`, the session cookie will be sent on every request.

Similarly, the `expires` part of a session cookie is updated each time the session cookie is sent.

The session is not saved if the response's status code is 500.

Browser-length sessions vs. persistent sessions

You can control whether the session framework uses browser-length sessions vs. persistent sessions with the *SESSION_EXPIRE_AT_BROWSER_CLOSE* setting.

By default, *SESSION_EXPIRE_AT_BROWSER_CLOSE* is set to `False`, which means session cookies will be stored in users' browsers for as long as *SESSION_COOKIE_AGE*. Use this if you don't want people to have to log in every time they open a browser.

If *SESSION_EXPIRE_AT_BROWSER_CLOSE* is set to `True`, Django will use browser-length cookies – cookies that expire as soon as the user closes their browser. Use this if you want people to have to log in every time they open a browser.

This setting is a global default and can be overwritten at a per-session level by explicitly calling the *set_expiry()* method of `request.session` as described above in *using sessions in views*.

: Some browsers (Chrome, for example) provide settings that allow users to continue browsing sessions after closing and re-opening the browser. In some cases, this can interfere with the *SESSION_EXPIRE_AT_BROWSER_CLOSE*

setting and prevent sessions from expiring on browser close. Please be aware of this while testing Django applications which have the *SESSION_EXPIRE_AT_BROWSER_CLOSE* setting enabled.

Clearing the session store

As users create new sessions on your website, session data can accumulate in your session store. If you're using the database backend, the `django_session` database table will grow. If you're using the file backend, your temporary directory will contain an increasing number of files.

To understand this problem, consider what happens with the database backend. When a user logs in, Django adds a row to the `django_session` database table. Django updates this row each time the session data changes. If the user logs out manually, Django deletes the row. But if the user does *not* log out, the row never gets deleted. A similar process happens with the file backend.

Django does *not* provide automatic purging of expired sessions. Therefore, it's your job to purge expired sessions on a regular basis. Django provides a clean-up management command for this purpose: *clearsessions*. It's recommended to call this command on a regular basis, for example as a daily cron job.

Note that the cache backend isn't vulnerable to this problem, because caches automatically delete stale data. Neither is the cookie backend, because the session data is stored by the users' browsers.

Settings

A few *Django settings* give you control over session behavior:

- *SESSION_CACHE_ALIAS*
- *SESSION_COOKIE_AGE*
- *SESSION_COOKIE_DOMAIN*
- *SESSION_COOKIE_HTTPONLY*
- *SESSION_COOKIE_NAME*
- *SESSION_COOKIE_PATH*
- *SESSION_COOKIE_SECURE*
- *SESSION_ENGINE*
- *SESSION_EXPIRE_AT_BROWSER_CLOSE*
- *SESSION_FILE_PATH*
- *SESSION_SAVE_EVERY_REQUEST*
- *SESSION_SERIALIZER*

Session security

Subdomains within a site are able to set cookies on the client for the whole domain. This makes session fixation possible if cookies are permitted from subdomains not controlled by trusted users.

For example, an attacker could log into `good.example.com` and get a valid session for their account. If the attacker has control over `bad.example.com`, they can use it to send their session key to you since a subdomain is permitted to set cookies on `*.example.com`. When you visit `good.example.com`, you'll be logged in as the attacker and might inadvertently enter your sensitive personal data (e.g. credit card info) into the attackers account.

Another possible attack would be if `good.example.com` sets its *SESSION_COOKIE_DOMAIN* to `".example.com"` which would cause session cookies from that site to be sent to `bad.example.com`.

Technical details

- The session dictionary accepts any `json` serializable value when using *JSONSerializer* or any picklable Python object when using *PickleSerializer*. See the `pickle` module for more information.
- Session data is stored in a database table named `django_session` .
- Django only sends a cookie if it needs to. If you don't set any session data, it won't send a session cookie.

The `SessionStore` object

When working with sessions internally, Django uses a session store object from the corresponding session engine. By convention, the session store object class is named `SessionStore` and is located in the module designated by *SESSION_ENGINE*.

All `SessionStore` classes available in Django inherit from *SessionBase* and implement data manipulation methods, namely:

- `exists()`
- `create()`
- `save()`
- `delete()`
- `load()`
- *clear_expired()*

In order to build a custom session engine or to customize an existing one, you may create a new class inheriting from *SessionBase* or any other existing `SessionStore` class.

Extending most of the session engines is quite straightforward, but doing so with database-backed session engines generally requires some extra effort (see the next section for details).

Extending database-backed session engines

Creating a custom database-backed session engine built upon those included in Django (namely `db` and `cached_db`) may be done by inheriting *AbstractBaseSession* and either `SessionStore` class.

`AbstractBaseSession` and `BaseSessionManager` are importable from `django.contrib.sessions.base_session` so that they can be imported without including `django.contrib.sessions` in *INSTALLED_APPS*.

class `base_session.`**AbstractBaseSession**
The abstract base session model.

session_key
Primary key. The field itself may contain up to 40 characters. The current implementation generates a 32-character string (a random sequence of digits and lowercase ASCII letters).

session_data
A string containing an encoded and serialized session dictionary.

expire_date
A datetime designating when the session expires.

Expired sessions are not available to a user, however, they may still be stored in the database until the *clearsessions* management command is run.

classmethod get_session_store_class()
Returns a session store class to be used with this session model.

get_decoded()
Returns decoded session data.

Decoding is performed by the session store class.

You can also customize the model manager by subclassing *BaseSessionManager*:

class base_session.**BaseSessionManager**

encode(*session_dict*)
Returns the given session dictionary serialized and encoded as a string.

Encoding is performed by the session store class tied to a model class.

save(*session_key*, *session_dict*, *expire_date*)
Saves session data for a provided session key, or deletes the session in case the data is empty.

Customization of SessionStore classes is achieved by overriding methods and properties described below:

class backends.db.**SessionStore**
Implements database-backed session store.

classmethod get_model_class()
Override this method to return a custom session model if you need one.

create_model_instance(*data*)
Returns a new instance of the session model object, which represents the current session state.

Overriding this method provides the ability to modify session model data before it's saved to database.

class backends.cached_db.**SessionStore**
Implements cached database-backed session store.

cache_key_prefix
A prefix added to a session key to build a cache key string.

Example

The example below shows a custom database-backed session engine that includes an additional database column to store an account ID (thus providing an option to query the database for all active sessions for an account):

```
from django.contrib.sessions.backends.db import SessionStore as DBStore
from django.contrib.sessions.base_session import AbstractBaseSession
from django.db import models

class CustomSession(AbstractBaseSession):
    account_id = models.IntegerField(null=True, db_index=True)

    @classmethod
    def get_session_store_class(cls):
        return SessionStore
```

```
class SessionStore(DBStore):
    @classmethod
    def get_model_class(cls):
        return CustomSession

    def create_model_instance(self, data):
        obj = super(SessionStore, self).create_model_instance(data)
        try:
            account_id = int(data.get('_auth_user_id'))
        except (ValueError, TypeError):
            account_id = None
        obj.account_id = account_id
        return obj
```

If you are migrating from the Django's built-in `cached_db` session store to a custom one based on `cached_db`, you should override the cache key prefix in order to prevent a namespace clash:

```
class SessionStore(CachedDBStore):
    cache_key_prefix = 'mysessions.custom_cached_db_backend'

    # ...
```

Session IDs in URLs

The Django sessions framework is entirely, and solely, cookie-based. It does not fall back to putting session IDs in URLs as a last resort, as PHP does. This is an intentional design decision. Not only does that behavior make URLs ugly, it makes your site vulnerable to session-ID theft via the "Referer" header.

3.4 Working with forms

About this document

This document provides an introduction to the basics of web forms and how they are handled in Django. For a more detailed look at specific areas of the forms API, see The Forms API, Form fields, and Form and field validation.

Unless you're planning to build websites and applications that do nothing but publish content, and don't accept input from your visitors, you're going to need to understand and use forms.

Django provides a range of tools and libraries to help you build forms to accept input from site visitors, and then process and respond to the input.

3.4.1 HTML forms

In HTML, a form is a collection of elements inside `<form>...</form>` that allow a visitor to do things like enter text, select options, manipulate objects or controls, and so on, and then send that information back to the server.

Some of these form interface elements - text input or checkboxes - are fairly simple and are built into HTML itself. Others are much more complex; an interface that pops up a date picker or allows you to move a slider or manipulate controls will typically use JavaScript and CSS as well as HTML form `<input>` elements to achieve these effects.

As well as its `<input>` elements, a form must specify two things:

- *where*: the URL to which the data corresponding to the user's input should be returned

- *how*: the HTTP method the data should be returned by

As an example, the login form for the Django admin contains several `<input>` elements: one of `type="text"` for the username, one of `type="password"` for the password, and one of `type="submit"` for the "Log in" button. It also contains some hidden text fields that the user doesn't see, which Django uses to determine what to do next.

It also tells the browser that the form data should be sent to the URL specified in the `<form>`'s `action` attribute - `/admin/` - and that it should be sent using the HTTP mechanism specified by the `method` attribute - `post`.

When the `<input type="submit" value="Log in">` element is triggered, the data is returned to `/admin/`.

`GET` and `POST`

`GET` and `POST` are the only HTTP methods to use when dealing with forms.

Django's login form is returned using the `POST` method, in which the browser bundles up the form data, encodes it for transmission, sends it to the server, and then receives back its response.

`GET`, by contrast, bundles the submitted data into a string, and uses this to compose a URL. The URL contains the address where the data must be sent, as well as the data keys and values. You can see this in action if you do a search in the Django documentation, which will produce a URL of the form `https://docs.djangoproject.com/search/?q=forms&release=1`.

`GET` and `POST` are typically used for different purposes.

Any request that could be used to change the state of the system - for example, a request that makes changes in the database - should use `POST`. `GET` should be used only for requests that do not affect the state of the system.

`GET` would also be unsuitable for a password form, because the password would appear in the URL, and thus, also in browser history and server logs, all in plain text. Neither would it be suitable for large quantities of data, or for binary data, such as an image. A Web application that uses `GET` requests for admin forms is a security risk: it can be easy for an attacker to mimic a form's request to gain access to sensitive parts of the system. `POST`, coupled with other protections like Django's CSRF protection offers more control over access.

On the other hand, `GET` is suitable for things like a web search form, because the URLs that represent a `GET` request can easily be bookmarked, shared, or resubmitted.

3.4.2 Django's role in forms

Handling forms is a complex business. Consider Django's admin, where numerous items of data of several different types may need to be prepared for display in a form, rendered as HTML, edited using a convenient interface, returned to the server, validated and cleaned up, and then saved or passed on for further processing.

Django's form functionality can simplify and automate vast portions of this work, and can also do it more securely than most programmers would be able to do in code they wrote themselves.

Django handles three distinct parts of the work involved in forms:

- preparing and restructuring data to make it ready for rendering
- creating HTML forms for the data
- receiving and processing submitted forms and data from the client

It is *possible* to write code that does all of this manually, but Django can take care of it all for you.

3.4.3 Forms in Django

We've described HTML forms briefly, but an HTML `<form>` is just one part of the machinery required.

In the context of a Web application, 'form' might refer to that HTML `<form>`, or to the Django *Form* that produces it, or to the structured data returned when it is submitted, or to the end-to-end working collection of these parts.

The Django `Form` class

At the heart of this system of components is Django's *Form* class. In much the same way that a Django model describes the logical structure of an object, its behavior, and the way its parts are represented to us, a *Form* class describes a form and determines how it works and appears.

In a similar way that a model class's fields map to database fields, a form class's fields map to HTML form `<input>` elements. (A *ModelForm* maps a model class's fields to HTML form `<input>` elements via a *Form*; this is what the Django admin is based upon.)

A form's fields are themselves classes; they manage form data and perform validation when a form is submitted. A *DateField* and a *FileField* handle very different kinds of data and have to do different things with it.

A form field is represented to a user in the browser as an HTML "widget" - a piece of user interface machinery. Each field type has an appropriate default Widget class, but these can be overridden as required.

Instantiating, processing, and rendering forms

When rendering an object in Django, we generally:

1. get hold of it in the view (fetch it from the database, for example)
2. pass it to the template context
3. expand it to HTML markup using template variables

Rendering a form in a template involves nearly the same work as rendering any other kind of object, but there are some key differences.

In the case of a model instance that contained no data, it would rarely if ever be useful to do anything with it in a template. On the other hand, it makes perfect sense to render an unpopulated form - that's what we do when we want the user to populate it.

So when we handle a model instance in a view, we typically retrieve it from the database. When we're dealing with a form we typically instantiate it in the view.

When we instantiate a form, we can opt to leave it empty or pre-populate it, for example with:

- data from a saved model instance (as in the case of admin forms for editing)
- data that we have collated from other sources
- data received from a previous HTML form submission

The last of these cases is the most interesting, because it's what makes it possible for users not just to read a website, but to send information back to it too.

3.4.4 Building a form

The work that needs to be done

Suppose you want to create a simple form on your website, in order to obtain the user's name. You'd need something like this in your template:

```
<form action="/your-name/" method="post">
    <label for="your_name">Your name: </label>
    <input id="your_name" type="text" name="your_name" value="{{ current_name }}">
    <input type="submit" value="OK">
</form>
```

This tells the browser to return the form data to the URL `/your-name/`, using the `POST` method. It will display a text field, labeled "Your name:", and a button marked "OK". If the template context contains a `current_name` variable, that will be used to pre-fill the `your_name` field.

You'll need a view that renders the template containing the HTML form, and that can supply the `current_name` field as appropriate.

When the form is submitted, the `POST` request which is sent to the server will contain the form data.

Now you'll also need a view corresponding to that `/your-name/` URL which will find the appropriate key/value pairs in the request, and then process them.

This is a very simple form. In practice, a form might contain dozens or hundreds of fields, many of which might need to be pre-populated, and we might expect the user to work through the edit-submit cycle several times before concluding the operation.

We might require some validation to occur in the browser, even before the form is submitted; we might want to use much more complex fields, that allow the user to do things like pick dates from a calendar and so on.

At this point it's much easier to get Django to do most of this work for us.

Building a form in Django

The `Form` class

We already know what we want our HTML form to look like. Our starting point for it in Django is this:

forms.py

```
from django import forms

class NameForm(forms.Form):
    your_name = forms.CharField(label='Your name', max_length=100)
```

This defines a *`Form`* class with a single field (`your_name`). We've applied a human-friendly label to the field, which will appear in the `<label>` when it's rendered (although in this case, the *`label`* we specified is actually the same one that would be generated automatically if we had omitted it).

The field's maximum allowable length is defined by *`max_length`*. This does two things. It puts a `maxlength="100"` on the HTML `<input>` (so the browser should prevent the user from entering more than that number of characters in the first place). It also means that when Django receives the form back from the browser, it will validate the length of the data.

A *`Form`* instance has an *`is_valid()`* method, which runs validation routines for all its fields. When this method is called, if all fields contain valid data, it will:

- return `True`
- place the form's data in its *`cleaned_data`* attribute.

The whole form, when rendered for the first time, will look like:

```
<label for="your_name">Your name: </label>
<input id="your_name" type="text" name="your_name" maxlength="100" required />
```

Note that it **does not** include the `<form>` tags, or a submit button. We'll have to provide those ourselves in the template.

The view

Form data sent back to a Django website is processed by a view, generally the same view which published the form. This allows us to reuse some of the same logic.

To handle the form we need to instantiate it in the view for the URL where we want it to be published:

views.py

```
from django.shortcuts import render
from django.http import HttpResponseRedirect

from .forms import NameForm

def get_name(request):
    # if this is a POST request we need to process the form data
    if request.method == 'POST':
        # create a form instance and populate it with data from the request:
        form = NameForm(request.POST)
        # check whether it's valid:
        if form.is_valid():
            # process the data in form.cleaned_data as required
            # ...
            # redirect to a new URL:
            return HttpResponseRedirect('/thanks/')

    # if a GET (or any other method) we'll create a blank form
    else:
        form = NameForm()

    return render(request, 'name.html', {'form': form})
```

If we arrive at this view with a `GET` request, it will create an empty form instance and place it in the template context to be rendered. This is what we can expect to happen the first time we visit the URL.

If the form is submitted using a `POST` request, the view will once again create a form instance and populate it with data from the request: `form = NameForm(request.POST)` This is called "binding data to the form" (it is now a *bound* form).

We call the form's `is_valid()` method; if it's not `True`, we go back to the template with the form. This time the form is no longer empty (*unbound*) so the HTML form will be populated with the data previously submitted, where it can be edited and corrected as required.

If `is_valid()` is `True`, we'll now be able to find all the validated form data in its `cleaned_data` attribute. We can use this data to update the database or do other processing before sending an HTTP redirect to the browser telling it where to go next.

The template

We don't need to do much in our `name.html` template. The simplest example is:

```
<form action="/your-name/" method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Submit" />
</form>
```

All the form's fields and their attributes will be unpacked into HTML markup from that `{{ form }}` by Django's template language.

Forms and Cross Site Request Forgery protection

Django ships with an easy-to-use protection against Cross Site Request Forgeries. When submitting a form via `POST` with CSRF protection enabled you must use the *`csrf_token`* template tag as in the preceding example. However, since CSRF protection is not directly tied to forms in templates, this tag is omitted from the following examples in this document.

HTML5 input types and browser validation

If your form includes a *`URLField`*, an *`EmailField`* or any integer field type, Django will use the `url`, `email` and `number` HTML5 input types. By default, browsers may apply their own validation on these fields, which may be stricter than Django's validation. If you would like to disable this behavior, set the *novalidate* attribute on the `form` tag, or specify a different widget on the field, like *`TextInput`*.

We now have a working web form, described by a Django *`Form`*, processed by a view, and rendered as an HTML `<form>`.

That's all you need to get started, but the forms framework puts a lot more at your fingertips. Once you understand the basics of the process described above, you should be prepared to understand other features of the forms system and ready to learn a bit more about the underlying machinery.

3.4.5 More about Django `Form` classes

All form classes are created as subclasses of *`django.forms.Form`*, including the ModelForm, which you encounter in Django's admin.

Models and Forms

In fact if your form is going to be used to directly add or edit a Django model, a ModelForm can save you a great deal of time, effort, and code, because it will build a form, along with the appropriate fields and their attributes, from a `Model` class.

Bound and unbound form instances

The distinction between *Bound and unbound forms* is important:

- An unbound form has no data associated with it. When rendered to the user, it will be empty or will contain default values.

- A bound form has submitted data, and hence can be used to tell if that data is valid. If an invalid bound form is rendered, it can include inline error messages telling the user what data to correct.

The form's *is_bound* attribute will tell you whether a form has data bound to it or not.

More on fields

Consider a more useful form than our minimal example above, which we could use to implement "contact me" functionality on a personal website:

forms.py

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField(widget=forms.Textarea)
    sender = forms.EmailField()
    cc_myself = forms.BooleanField(required=False)
```

Our earlier form used a single field, `your_name`, a *CharField*. In this case, our form has four fields: `subject`, `message`, `sender` and `cc_myself`. *CharField*, *EmailField* and *BooleanField* are just three of the available field types; a full list can be found in Form fields.

Widgets

Each form field has a corresponding Widget class, which in turn corresponds to an HTML form widget such as `<input type="text">`.

In most cases, the field will have a sensible default widget. For example, by default, a *CharField* will have a *TextInput* widget, that produces an `<input type="text">` in the HTML. If you needed `<textarea>` instead, you'd specify the appropriate widget when defining your form field, as we have done for the `message` field.

Field data

Whatever the data submitted with a form, once it has been successfully validated by calling `is_valid()` (and `is_valid()` has returned `True`), the validated form data will be in the `form.cleaned_data` dictionary. This data will have been nicely converted into Python types for you.

: You can still access the unvalidated data directly from `request.POST` at this point, but the validated data is better.

In the contact form example above, `cc_myself` will be a boolean value. Likewise, fields such as *IntegerField* and *FloatField* convert values to a Python `int` and `float` respectively.

Here's how the form data could be processed in the view that handles this form:

views.py

```
from django.core.mail import send_mail

if form.is_valid():
    subject = form.cleaned_data['subject']
    message = form.cleaned_data['message']
    sender = form.cleaned_data['sender']
    cc_myself = form.cleaned_data['cc_myself']
```

```
    recipients = ['info@example.com']
    if cc_myself:
        recipients.append(sender)

    send_mail(subject, message, sender, recipients)
    return HttpResponseRedirect('/thanks/')
```

: For more on sending email from Django, see Sending email.

Some field types need some extra handling. For example, files that are uploaded using a form need to be handled differently (they can be retrieved from `request.FILES`, rather than `request.POST`). For details of how to handle file uploads with your form, see *Binding uploaded files to a form*.

3.4.6 Working with form templates

All you need to do to get your form into a template is to place the form instance into the template context. So if your form is called `form` in the context, `{{ form }}` will render its `<label>` and `<input>` elements appropriately.

Form rendering options

Additional form template furniture

Don't forget that a form's output does *not* include the surrounding `<form>` tags, or the form's `submit` control. You will have to provide these yourself.

There are other output options though for the `<label>`/`<input>` pairs:

- `{{ form.as_table }}` will render them as table cells wrapped in `<tr>` tags
- `{{ form.as_p }}` will render them wrapped in `<p>` tags
- `{{ form.as_ul }}` will render them wrapped in `<li>` tags

Note that you'll have to provide the surrounding `<table>` or `<ul>` elements yourself.

Here's the output of `{{ form.as_p }}` for our `ContactForm` instance:

```
<p><label for="id_subject">Subject:</label>
    <input id="id_subject" type="text" name="subject" maxlength="100" required /></p>
<p><label for="id_message">Message:</label>
    <textarea name="message" id="id_message" required></textarea></p>
<p><label for="id_sender">Sender:</label>
    <input type="email" name="sender" id="id_sender" required /></p>
<p><label for="id_cc_myself">Cc myself:</label>
    <input type="checkbox" name="cc_myself" id="id_cc_myself" /></p>
```

Note that each form field has an ID attribute set to `id_<field-name>`, which is referenced by the accompanying label tag. This is important in ensuring that forms are accessible to assistive technology such as screen reader software. You can also *customize the way in which labels and ids are generated*.

See *Outputting forms as HTML* for more on this.

Rendering fields manually

We don't have to let Django unpack the form's fields; we can do it manually if we like (allowing us to reorder the fields, for example). Each field is available as an attribute of the form using `{{ form.name_of_field }}`, and in a Django template, will be rendered appropriately. For example:

```
{{ form.non_field_errors }}
<div class="fieldWrapper">
    {{ form.subject.errors }}
    <label for="{{ form.subject.id_for_label }}">Email subject:</label>
    {{ form.subject }}
</div>
<div class="fieldWrapper">
    {{ form.message.errors }}
    <label for="{{ form.message.id_for_label }}">Your message:</label>
    {{ form.message }}
</div>
<div class="fieldWrapper">
    {{ form.sender.errors }}
    <label for="{{ form.sender.id_for_label }}">Your email address:</label>
    {{ form.sender }}
</div>
<div class="fieldWrapper">
    {{ form.cc_myself.errors }}
    <label for="{{ form.cc_myself.id_for_label }}">CC yourself?</label>
    {{ form.cc_myself }}
</div>
```

Complete `<label>` elements can also be generated using the *`label_tag()`*. For example:

```
<div class="fieldWrapper">
    {{ form.subject.errors }}
    {{ form.subject.label_tag }}
    {{ form.subject }}
</div>
```

Rendering form error messages

Of course, the price of this flexibility is more work. Until now we haven't had to worry about how to display form errors, because that's taken care of for us. In this example we have had to make sure we take care of any errors for each field and any errors for the form as a whole. Note `{{ form.non_field_errors }}` at the top of the form and the template lookup for errors on each field.

Using `{{ form.name_of_field.errors }}` displays a list of form errors, rendered as an unordered list. This might look like:

```
<ul class="errorlist">
    <li>Sender is required.</li>
</ul>
```

The list has a CSS class of `errorlist` to allow you to style its appearance. If you wish to further customize the display of errors you can do so by looping over them:

```
{% if form.subject.errors %}
    <ol>
    {% for error in form.subject.errors %}
        <li><strong>{{ error|escape }}</strong></li>
    {% endfor %}
```

```
    </ol>
{% endif %}
```

Non-field errors (and/or hidden field errors that are rendered at the top of the form when using helpers like `form.as_p()`) will be rendered with an additional class of `nonfield` to help distinguish them from field-specific errors. For example, `{{ form.non_field_errors }}` would look like:

```
<ul class="errorlist nonfield">
    <li>Generic validation error</li>
</ul>
```

See The Forms API for more on errors, styling, and working with form attributes in templates.

Looping over the form's fields

If you're using the same HTML for each of your form fields, you can reduce duplicate code by looping through each field in turn using a `{% for %}` loop:

```
{% for field in form %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
        {% if field.help_text %}
        <p class="help">{{ field.help_text|safe }}</p>
        {% endif %}
    </div>
{% endfor %}
```

Useful attributes on `{{ field }}` include:

`{{ field.label }}` The label of the field, e.g. `Email address`.

`{{ field.label_tag }}` The field's label wrapped in the appropriate HTML `<label>` tag. This includes the form's *`label_suffix`*. For example, the default `label_suffix` is a colon:

```
<label for="id_email">Email address:</label>
```

`{{ field.id_for_label }}` The ID that will be used for this field (`id_email` in the example above). If you are constructing the label manually, you may want to use this in lieu of `label_tag`. It's also useful, for example, if you have some inline JavaScript and want to avoid hardcoding the field's ID.

`{{ field.value }}` The value of the field. e.g `someone@example.com`.

`{{ field.html_name }}` The name of the field that will be used in the input element's name field. This takes the form prefix into account, if it has been set.

`{{ field.help_text }}` Any help text that has been associated with the field.

`{{ field.errors }}` Outputs a `<ul class="errorlist">` containing any validation errors corresponding to this field. You can customize the presentation of the errors with a `{% for error in field.errors %}` loop. In this case, each object in the loop is a simple string containing the error message.

`{{ field.is_hidden }}` This attribute is `True` if the form field is a hidden field and `False` otherwise. It's not particularly useful as a template variable, but could be useful in conditional tests such as:

```
{% if field.is_hidden %}
   {# Do something special #}
{% endif %}
```

{{ field.field }} The *Field* instance from the form class that this *BoundField* wraps. You can use it to access *Field* attributes, e.g. {{ char_field.field.max_length }}.

:

For a complete list of attributes and methods, see *BoundField*.

Looping over hidden and visible fields

If you're manually laying out a form in a template, as opposed to relying on Django's default form layout, you might want to treat <input type="hidden"> fields differently from non-hidden fields. For example, because hidden fields don't display anything, putting error messages "next to" the field could cause confusion for your users – so errors for those fields should be handled differently.

Django provides two methods on a form that allow you to loop over the hidden and visible fields independently: hidden_fields() and visible_fields(). Here's a modification of an earlier example that uses these two methods:

```
{# Include the hidden fields #}
{% for hidden in form.hidden_fields %}
{{ hidden }}
{% endfor %}
{# Include the visible fields #}
{% for field in form.visible_fields %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
    </div>
{% endfor %}
```

This example does not handle any errors in the hidden fields. Usually, an error in a hidden field is a sign of form tampering, since normal form interaction won't alter them. However, you could easily insert some error displays for those form errors, as well.

Reusable form templates

If your site uses the same rendering logic for forms in multiple places, you can reduce duplication by saving the form's loop in a standalone template and using the *include* tag to reuse it in other templates:

```
# In your form template:
{% include "form_snippet.html" %}

# In form_snippet.html:
{% for field in form %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
    </div>
{% endfor %}
```

If the form object passed to a template has a different name within the context, you can alias it using the with argument of the *include* tag:

```
{% include "form_snippet.html" with form=comment_form %}
```

If you find yourself doing this often, you might consider creating a custom *inclusion tag*.

3.4.7 Further topics

This covers the basics, but forms can do a whole lot more:

Formsets

class BaseFormSet

A formset is a layer of abstraction to work with multiple forms on the same page. It can be best compared to a data grid. Let's say you have the following form:

```
>>> from django import forms
>>> class ArticleForm(forms.Form):
...     title = forms.CharField()
...     pub_date = forms.DateField()
```

You might want to allow the user to create several articles at once. To create a formset out of an `ArticleForm` you would do:

```
>>> from django.forms import formset_factory
>>> ArticleFormSet = formset_factory(ArticleForm)
```

You now have created a formset named `ArticleFormSet`. The formset gives you the ability to iterate over the forms in the formset and display them as you would with a regular form:

```
>>> formset = ArticleFormSet()
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text" name="form-0-title" id
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text" name="form-0-pub
```

As you can see it only displayed one empty form. The number of empty forms that is displayed is controlled by the `extra` parameter. By default, *`formset_factory()`* defines one extra form; the following example will display two blank forms:

```
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2)
```

Iterating over the `formset` will render the forms in the order they were created. You can change this order by providing an alternate implementation for the `__iter__()` method.

Formsets can also be indexed into, which returns the corresponding form. If you override `__iter__`, you will need to also override `__getitem__` to have matching behavior.

Using initial data with a formset

Initial data is what drives the main usability of a formset. As shown above you can define the number of extra forms. What this means is that you are telling the formset how many additional forms to show in addition to the number of forms it generates from the initial data. Let's take a look at an example:

```
>>> import datetime
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2)
>>> formset = ArticleFormSet(initial=[
...     {'title': 'Django is now open source',
...      'pub_date': datetime.date.today(),}
... ])
```

```
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text" name="form-0-title" va
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text" name="form-0-pub
<tr><th><label for="id_form-1-title">Title:</label></th><td><input type="text" name="form-1-title" id
<tr><th><label for="id_form-1-pub_date">Pub date:</label></th><td><input type="text" name="form-1-pub
<tr><th><label for="id_form-2-title">Title:</label></th><td><input type="text" name="form-2-title" id
<tr><th><label for="id_form-2-pub_date">Pub date:</label></th><td><input type="text" name="form-2-pub
```

There are now a total of three forms showing above. One for the initial data that was passed in and two extra forms. Also note that we are passing in a list of dictionaries as the initial data.

If you use an `initial` for displaying a formset, you should pass the same `initial` when processing that formset's submission so that the formset can detect which forms were changed by the user. For example, you might have something like: `ArticleFormSet(request.POST, initial=[...])`.

:

Creating formsets from models with model formsets.

Limiting the maximum number of forms

The `max_num` parameter to *`formset_factory()`* gives you the ability to limit the number of forms the formset will display:

```
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2, max_num=1)
>>> formset = ArticleFormSet()
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text" name="form-0-title" id
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text" name="form-0-pub
```

If the value of `max_num` is greater than the number of existing items in the initial data, up to `extra` additional blank forms will be added to the formset, so long as the total number of forms does not exceed `max_num`. For example, if `extra=2` and `max_num=2` and the formset is initialized with one `initial` item, a form for the initial item and one blank form will be displayed.

If the number of items in the initial data exceeds `max_num`, all initial data forms will be displayed regardless of the value of `max_num` and no extra forms will be displayed. For example, if `extra=3` and `max_num=1` and the formset is initialized with two initial items, two forms with the initial data will be displayed.

A `max_num` value of `None` (the default) puts a high limit on the number of forms displayed (1000). In practice this is equivalent to no limit.

By default, `max_num` only affects how many forms are displayed and does not affect validation. If `validate_max=True` is passed to the *`formset_factory()`*, then `max_num` will affect validation. See *validate_max*.

Formset validation

Validation with a formset is almost identical to a regular `Form`. There is an `is_valid` method on the formset to provide a convenient way to validate all forms in the formset:

```
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm)
>>> data = {
...     'form-TOTAL_FORMS': '1',
...     'form-INITIAL_FORMS': '0',
...     'form-MAX_NUM_FORMS': '',
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
True
```

We passed in no data to the formset which is resulting in a valid form. The formset is smart enough to ignore extra forms that were not changed. If we provide an invalid article:

```
>>> data = {
...     'form-TOTAL_FORMS': '2',
...     'form-INITIAL_FORMS': '0',
...     'form-MAX_NUM_FORMS': '',
...     'form-0-title': 'Test',
...     'form-0-pub_date': '1904-06-16',
...     'form-1-title': 'Test',
...     'form-1-pub_date': '', # <-- this date is missing but required
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
False
>>> formset.errors
[{}, {'pub_date': ['This field is required.']}]
```

As we can see, `formset.errors` is a list whose entries correspond to the forms in the formset. Validation was performed for each of the two forms, and the expected error message appears for the second item.

Just like when using a normal `Form`, each field in a formset's forms may include HTML attributes such as `maxlength` for browser validation. However, form fields of formsets won't include the `required` attribute as that validation may be incorrect when adding and deleting forms.

`BaseFormSet.`**`total_error_count`**`()`

To check how many errors there are in the formset, we can use the `total_error_count` method:

```
>>> # Using the previous example
>>> formset.errors
[{}, {'pub_date': ['This field is required.']}]
>>> len(formset.errors)
2
>>> formset.total_error_count()
1
```

We can also check if form data differs from the initial data (i.e. the form was sent without any data):

```
>>> data = {
...     'form-TOTAL_FORMS': '1',
...     'form-INITIAL_FORMS': '0',
...     'form-MAX_NUM_FORMS': '',
...     'form-0-title': '',
...     'form-0-pub_date': '',
... }
>>> formset = ArticleFormSet(data)
```

```
>>> formset.has_changed()
False
```

Understanding the ManagementForm You may have noticed the additional data (`form-TOTAL_FORMS`, `form-INITIAL_FORMS` and `form-MAX_NUM_FORMS`) that was required in the formset's data above. This data is required for the `ManagementForm`. This form is used by the formset to manage the collection of forms contained in the formset. If you don't provide this management data, an exception will be raised:

```
>>> data = {
...     'form-0-title': 'Test',
...     'form-0-pub_date': '',
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
Traceback (most recent call last):
...
django.forms.utils.ValidationError: ['ManagementForm data is missing or has been tampered with']
```

It is used to keep track of how many form instances are being displayed. If you are adding new forms via JavaScript, you should increment the count fields in this form as well. On the other hand, if you are using JavaScript to allow deletion of existing objects, then you need to ensure the ones being removed are properly marked for deletion by including `form-#-DELETE` in the `POST` data. It is expected that all forms are present in the `POST` data regardless.

The management form is available as an attribute of the formset itself. When rendering a formset in a template, you can include all the management data by rendering `{{ my_formset.management_form }}` (substituting the name of your formset as appropriate).

total_form_count and initial_form_count `BaseFormSet` has a couple of methods that are closely related to the `ManagementForm`, `total_form_count` and `initial_form_count`.

`total_form_count` returns the total number of forms in this formset. `initial_form_count` returns the number of forms in the formset that were pre-filled, and is also used to determine how many forms are required. You will probably never need to override either of these methods, so please be sure you understand what they do before doing so.

empty_form `BaseFormSet` provides an additional attribute `empty_form` which returns a form instance with a prefix of `__prefix__` for easier use in dynamic forms with JavaScript.

Custom formset validation A formset has a `clean` method similar to the one on a `Form` class. This is where you define your own validation that works at the formset level:

```
>>> from django.forms import BaseFormSet
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm

>>> class BaseArticleFormSet(BaseFormSet):
...     def clean(self):
...         """Checks that no two articles have the same title."""
...         if any(self.errors):
...             # Don't bother validating the formset unless each form is valid on its own
...             return
...         titles = []
...         for form in self.forms:
...             title = form.cleaned_data['title']
```

```
...             if title in titles:
...                 raise forms.ValidationError("Articles in a set must have distinct titles.")
...             titles.append(title)

>>> ArticleFormSet = formset_factory(ArticleForm, formset=BaseArticleFormSet)
>>> data = {
...     'form-TOTAL_FORMS': '2',
...     'form-INITIAL_FORMS': '0',
...     'form-MAX_NUM_FORMS': '',
...     'form-0-title': 'Test',
...     'form-0-pub_date': '1904-06-16',
...     'form-1-title': 'Test',
...     'form-1-pub_date': '1912-06-23',
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
False
>>> formset.errors
[{}, {}]
>>> formset.non_form_errors()
['Articles in a set must have distinct titles.']
```

The formset `clean` method is called after all the `Form.clean` methods have been called. The errors will be found using the `non_form_errors()` method on the formset.

Validating the number of forms in a formset

Django provides a couple ways to validate the minimum or maximum number of submitted forms. Applications which need more customizable validation of the number of forms should use custom formset validation.

`validate_max` If `validate_max=True` is passed to *`formset_factory()`*, validation will also check that the number of forms in the data set, minus those marked for deletion, is less than or equal to `max_num`.

```
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, max_num=1, validate_max=True)
>>> data = {
...     'form-TOTAL_FORMS': '2',
...     'form-INITIAL_FORMS': '0',
...     'form-MIN_NUM_FORMS': '',
...     'form-MAX_NUM_FORMS': '',
...     'form-0-title': 'Test',
...     'form-0-pub_date': '1904-06-16',
...     'form-1-title': 'Test 2',
...     'form-1-pub_date': '1912-06-23',
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
False
>>> formset.errors
[{}, {}]
>>> formset.non_form_errors()
['Please submit 1 or fewer forms.']
```

`validate_max=True` validates against `max_num` strictly even if `max_num` was exceeded because the amount of initial data supplied was excessive.

: Regardless of `validate_max`, if the number of forms in a data set exceeds `max_num` by more than 1000, then the form will fail to validate as if `validate_max` were set, and additionally only the first 1000 forms above `max_num` will be validated. The remainder will be truncated entirely. This is to protect against memory exhaustion attacks using forged POST requests.

validate_min If `validate_min=True` is passed to *formset_factory()*, validation will also check that the number of forms in the data set, minus those marked for deletion, is greater than or equal to `min_num`.

```
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, min_num=3, validate_min=True)
>>> data = {
...     'form-TOTAL_FORMS': '2',
...     'form-INITIAL_FORMS': '0',
...     'form-MIN_NUM_FORMS': '',
...     'form-MAX_NUM_FORMS': '',
...     'form-0-title': 'Test',
...     'form-0-pub_date': '1904-06-16',
...     'form-1-title': 'Test 2',
...     'form-1-pub_date': '1912-06-23',
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
False
>>> formset.errors
[{}, {}]
>>> formset.non_form_errors()
['Please submit 3 or more forms.']
```

Dealing with ordering and deletion of forms

The *formset_factory()* provides two optional parameters `can_order` and `can_delete` to help with ordering of forms in formsets and deletion of forms from a formset.

can_order
`BaseFormSet.`**can_order**
Default: `False`

Lets you create a formset with the ability to order:

```
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, can_order=True)
>>> formset = ArticleFormSet(initial=[
...     {'title': 'Article #1', 'pub_date': datetime.date(2008, 5, 10)},
...     {'title': 'Article #2', 'pub_date': datetime.date(2008, 5, 11)},
... ])
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text" name="form-0-title" va
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text" name="form-0-pub
<tr><th><label for="id_form-0-ORDER">Order:</label></th><td><input type="number" name="form-0-ORDER"
<tr><th><label for="id_form-1-title">Title:</label></th><td><input type="text" name="form-1-title" va
```

```
<tr><th><label for="id_form-1-pub_date">Pub date:</label></th><td><input type="text" name="form-1-pub
<tr><th><label for="id_form-1-ORDER">Order:</label></th><td><input type="number" name="form-1-ORDER"
<tr><th><label for="id_form-2-title">Title:</label></th><td><input type="text" name="form-2-title" id
<tr><th><label for="id_form-2-pub_date">Pub date:</label></th><td><input type="text" name="form-2-pub
<tr><th><label for="id_form-2-ORDER">Order:</label></th><td><input type="number" name="form-2-ORDER"
```

This adds an additional field to each form. This new field is named `ORDER` and is an `forms.IntegerField`. For the forms that came from the initial data it automatically assigned them a numeric value. Let's look at what will happen when the user changes these values:

```
>>> data = {
...     'form-TOTAL_FORMS': '3',
...     'form-INITIAL_FORMS': '2',
...     'form-MAX_NUM_FORMS': '',
...     'form-0-title': 'Article #1',
...     'form-0-pub_date': '2008-05-10',
...     'form-0-ORDER': '2',
...     'form-1-title': 'Article #2',
...     'form-1-pub_date': '2008-05-11',
...     'form-1-ORDER': '1',
...     'form-2-title': 'Article #3',
...     'form-2-pub_date': '2008-05-01',
...     'form-2-ORDER': '0',
... }

>>> formset = ArticleFormSet(data, initial=[
...     {'title': 'Article #1', 'pub_date': datetime.date(2008, 5, 10)},
...     {'title': 'Article #2', 'pub_date': datetime.date(2008, 5, 11)},
... ])
>>> formset.is_valid()
True
>>> for form in formset.ordered_forms:
...     print(form.cleaned_data)
{'pub_date': datetime.date(2008, 5, 1), 'ORDER': 0, 'title': 'Article #3'}
{'pub_date': datetime.date(2008, 5, 11), 'ORDER': 1, 'title': 'Article #2'}
{'pub_date': datetime.date(2008, 5, 10), 'ORDER': 2, 'title': 'Article #1'}
```

can_delete

`BaseFormSet.`**can_delete**

Default: `False`

Lets you create a formset with the ability to select forms for deletion:

```
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
>>> ArticleFormSet = formset_factory(ArticleForm, can_delete=True)
>>> formset = ArticleFormSet(initial=[
...     {'title': 'Article #1', 'pub_date': datetime.date(2008, 5, 10)},
...     {'title': 'Article #2', 'pub_date': datetime.date(2008, 5, 11)},
... ])
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text" name="form-0-title" va
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text" name="form-0-pub
<tr><th><label for="id_form-0-DELETE">Delete:</label></th><td><input type="checkbox" name="form-0-DEL
<tr><th><label for="id_form-1-title">Title:</label></th><td><input type="text" name="form-1-title" va
<tr><th><label for="id_form-1-pub_date">Pub date:</label></th><td><input type="text" name="form-1-pub
<tr><th><label for="id_form-1-DELETE">Delete:</label></th><td><input type="checkbox" name="form-1-DEL
```

```
<tr><th><label for="id_form-2-title">Title:</label></th><td><input type="text" name="form-2-title" id
<tr><th><label for="id_form-2-pub_date">Pub date:</label></th><td><input type="text" name="form-2-pub
<tr><th><label for="id_form-2-DELETE">Delete:</label></th><td><input type="checkbox" name="form-2-DEL
```

Similar to `can_order` this adds a new field to each form named `DELETE` and is a `forms.BooleanField`. When data comes through marking any of the delete fields you can access them with `deleted_forms`:

```
>>> data = {
...     'form-TOTAL_FORMS': '3',
...     'form-INITIAL_FORMS': '2',
...     'form-MAX_NUM_FORMS': '',
...     'form-0-title': 'Article #1',
...     'form-0-pub_date': '2008-05-10',
...     'form-0-DELETE': 'on',
...     'form-1-title': 'Article #2',
...     'form-1-pub_date': '2008-05-11',
...     'form-1-DELETE': '',
...     'form-2-title': '',
...     'form-2-pub_date': '',
...     'form-2-DELETE': '',
... }

>>> formset = ArticleFormSet(data, initial=[
...     {'title': 'Article #1', 'pub_date': datetime.date(2008, 5, 10)},
...     {'title': 'Article #2', 'pub_date': datetime.date(2008, 5, 11)},
... ])
>>> [form.cleaned_data for form in formset.deleted_forms]
[{'DELETE': True, 'pub_date': datetime.date(2008, 5, 10), 'title': 'Article #1'}]
```

If you are using a *ModelFormSet*, model instances for deleted forms will be deleted when you call `formset.save()`.

If you call `formset.save(commit=False)`, objects will not be deleted automatically. You'll need to call `delete()` on each of the *formset.deleted_objects* to actually delete them:

```
>>> instances = formset.save(commit=False)
>>> for obj in formset.deleted_objects:
...     obj.delete()
```

On the other hand, if you are using a plain `FormSet`, it's up to you to handle `formset.deleted_forms`, perhaps in your formset's `save()` method, as there's no general notion of what it means to delete a form.

Adding additional fields to a formset

If you need to add additional fields to the formset this can be easily accomplished. The formset base class provides an `add_fields` method. You can simply override this method to add your own fields or even redefine the default fields/attributes of the order and deletion fields:

```
>>> from django.forms import BaseFormSet
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm
>>> class BaseArticleFormSet(BaseFormSet):
...     def add_fields(self, form, index):
...         super(BaseArticleFormSet, self).add_fields(form, index)
...         form.fields["my_field"] = forms.CharField()

>>> ArticleFormSet = formset_factory(ArticleForm, formset=BaseArticleFormSet)
>>> formset = ArticleFormSet()
```

```
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text" name="form-0-title" id
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text" name="form-0-pub
<tr><th><label for="id_form-0-my_field">My field:</label></th><td><input type="text" name="form-0-my_
```

Passing custom parameters to formset forms

Sometimes your form class takes custom parameters, like `MyArticleForm`. You can pass this parameter when instantiating the formset:

```
>>> from django.forms import BaseFormSet
>>> from django.forms import formset_factory
>>> from myapp.forms import ArticleForm

>>> class MyArticleForm(ArticleForm):
...     def __init__(self, *args, **kwargs):
...         self.user = kwargs.pop('user')
...         super(MyArticleForm, self).__init__(*args, **kwargs)

>>> ArticleFormSet = formset_factory(MyArticleForm)
>>> formset = ArticleFormSet(form_kwargs={'user': request.user})
```

The `form_kwargs` may also depend on the specific form instance. The formset base class provides a `get_form_kwargs` method. The method takes a single argument - the index of the form in the formset. The index is `None` for the *empty_form*:

```
>>> from django.forms import BaseFormSet
>>> from django.forms import formset_factory

>>> class BaseArticleFormSet(BaseFormSet):
...     def get_form_kwargs(self, index):
...         kwargs = super(BaseArticleFormSet, self).get_form_kwargs(index)
...         kwargs['custom_kwarg'] = index
...         return kwargs
```

The `form_kwargs` argument was added.

Using a formset in views and templates

Using a formset inside a view is as easy as using a regular `Form` class. The only thing you will want to be aware of is making sure to use the management form inside the template. Let's look at a sample view:

```
from django.forms import formset_factory
from django.shortcuts import render
from myapp.forms import ArticleForm

def manage_articles(request):
    ArticleFormSet = formset_factory(ArticleForm)
    if request.method == 'POST':
        formset = ArticleFormSet(request.POST, request.FILES)
        if formset.is_valid():
            # do something with the formset.cleaned_data
            pass
    else:
```

```
        formset = ArticleFormSet()
    return render(request, 'manage_articles.html', {'formset': formset})
```

The `manage_articles.html` template might look like this:

```
<form method="post" action="">
    {{ formset.management_form }}
    <table>
        {% for form in formset %}
        {{ form }}
        {% endfor %}
    </table>
</form>
```

However there's a slight shortcut for the above by letting the formset itself deal with the management form:

```
<form method="post" action="">
    <table>
        {{ formset }}
    </table>
</form>
```

The above ends up calling the `as_table` method on the formset class.

Manually rendered `can_delete` and `can_order` If you manually render fields in the template, you can render `can_delete` parameter with `{{ form.DELETE }}`:

```
<form method="post" action="">
    {{ formset.management_form }}
    {% for form in formset %}
        <ul>
            <li>{{ form.title }}</li>
            <li>{{ form.pub_date }}</li>
            {% if formset.can_delete %}
                <li>{{ form.DELETE }}</li>
            {% endif %}
        </ul>
    {% endfor %}
</form>
```

Similarly, if the formset has the ability to order (`can_order=True`), it is possible to render it with `{{ form.ORDER }}`.

Using more than one formset in a view You are able to use more than one formset in a view if you like. Formsets borrow much of its behavior from forms. With that said you are able to use `prefix` to prefix formset form field names with a given value to allow more than one formset to be sent to a view without name clashing. Lets take a look at how this might be accomplished:

```
from django.forms import formset_factory
from django.shortcuts import render
from myapp.forms import ArticleForm, BookForm

def manage_articles(request):
    ArticleFormSet = formset_factory(ArticleForm)
    BookFormSet = formset_factory(BookForm)
    if request.method == 'POST':
        article_formset = ArticleFormSet(request.POST, request.FILES, prefix='articles')
```

```
        book_formset = BookFormSet(request.POST, request.FILES, prefix='books')
        if article_formset.is_valid() and book_formset.is_valid():
            # do something with the cleaned_data on the formsets.
            pass
    else:
        article_formset = ArticleFormSet(prefix='articles')
        book_formset = BookFormSet(prefix='books')
    return render(request, 'manage_articles.html', {
        'article_formset': article_formset,
        'book_formset': book_formset,
    })
```

You would then render the formsets as normal. It is important to point out that you need to pass `prefix` on both the POST and non-POST cases so that it is rendered and processed correctly.

Creating forms from models

`ModelForm`

class `ModelForm`

If you're building a database-driven app, chances are you'll have forms that map closely to Django models. For instance, you might have a `BlogComment` model, and you want to create a form that lets people submit comments. In this case, it would be redundant to define the field types in your form, because you've already defined the fields in your model.

For this reason, Django provides a helper class that lets you create a `Form` class from a Django model.

For example:

```
>>> from django.forms import ModelForm
>>> from myapp.models import Article

# Create the form class.
>>> class ArticleForm(ModelForm):
...     class Meta:
...         model = Article
...         fields = ['pub_date', 'headline', 'content', 'reporter']

# Creating a form to add an article.
>>> form = ArticleForm()

# Creating a form to change an existing article.
>>> article = Article.objects.get(pk=1)
>>> form = ArticleForm(instance=article)
```

Field types The generated `Form` class will have a form field for every model field specified, in the order specified in the `fields` attribute.

Each model field has a corresponding default form field. For example, a `CharField` on a model is represented as a `CharField` on a form. A model `ManyToManyField` is represented as a `MultipleChoiceField`. Here is the full list of conversions:

Model field	Form field
AutoField	Not represented in the form
BigAutoField	Not represented in the form
BigIntegerField	*IntegerField* with `min_value` set to -9223372036854775808 and `max_value` set to 9223372036854775807.
BooleanField	*BooleanField*
CharField	*CharField* with `max_length` set to the model field's `max_length`
CommaSeparatedIntegerField	*CharField*
DateField	*DateField*
DateTimeField	*DateTimeField*
DecimalField	*DecimalField*
EmailField	*EmailField*
FileField	*FileField*
FilePathField	*FilePathField*
FloatField	*FloatField*
ForeignKey	*ModelChoiceField* (see below)
ImageField	*ImageField*
IntegerField	*IntegerField*
IPAddressField	IPAddressField
GenericIPAddressField	*GenericIPAddressField*
ManyToManyField	*ModelMultipleChoiceField* (see below)
NullBooleanField	*NullBooleanField*
PositiveIntegerField	*IntegerField*
PositiveSmallIntegerField	*IntegerField*
SlugField	*SlugField*
SmallIntegerField	*IntegerField*
TextField	*CharField* with `widget=forms.Textarea`
TimeField	*TimeField*
URLField	*URLField*

As you might expect, the `ForeignKey` and `ManyToManyField` model field types are special cases:

- `ForeignKey` is represented by `django.forms.ModelChoiceField`, which is a `ChoiceField` whose choices are a model `QuerySet`.
- `ManyToManyField` is represented by `django.forms.ModelMultipleChoiceField`, which is a `MultipleChoiceField` whose choices are a model `QuerySet`.

In addition, each generated form field has attributes set as follows:

- If the model field has `blank=True`, then `required` is set to `False` on the form field. Otherwise, `required=True`.
- The form field's `label` is set to the `verbose_name` of the model field, with the first character capitalized.
- The form field's `help_text` is set to the `help_text` of the model field.
- If the model field has `choices` set, then the form field's `widget` will be set to `Select`, with choices coming from the model field's `choices`. The choices will normally include the blank choice which is selected by default. If the field is required, this forces the user to make a selection. The blank choice will not be included if the model field has `blank=False` and an explicit `default` value (the `default` value will be initially selected instead).

Finally, note that you can override the form field used for a given model field. See *Overriding the default fields* below.

A full example Consider this set of models:

```
from django.db import models
from django.forms import ModelForm

TITLE_CHOICES = (
    ('MR', 'Mr.'),
    ('MRS', 'Mrs.'),
    ('MS', 'Ms.'),
)

class Author(models.Model):
    name = models.CharField(max_length=100)
    title = models.CharField(max_length=3, choices=TITLE_CHOICES)
    birth_date = models.DateField(blank=True, null=True)

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Book(models.Model):
    name = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)

class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ['name', 'title', 'birth_date']

class BookForm(ModelForm):
    class Meta:
        model = Book
        fields = ['name', 'authors']
```

With these models, the `ModelForm` subclasses above would be roughly equivalent to this (the only difference being the `save()` method, which we'll discuss in a moment.):

```
from django import forms

class AuthorForm(forms.Form):
    name = forms.CharField(max_length=100)
    title = forms.CharField(
        max_length=3,
        widget=forms.Select(choices=TITLE_CHOICES),
    )
    birth_date = forms.DateField(required=False)

class BookForm(forms.Form):
    name = forms.CharField(max_length=100)
    authors = forms.ModelMultipleChoiceField(queryset=Author.objects.all())
```

Validation on a `ModelForm` There are two main steps involved in validating a `ModelForm`:

1. Validating the form
2. *Validating the model instance*

Just like normal form validation, model form validation is triggered implicitly when calling *`is_valid()`* or accessing the *`errors`* attribute and explicitly when calling `full_clean()`, although you will typically not use the latter method in practice.

`Model` validation (*`Model.full_clean()`*) is triggered from within the form validation step, right after the form's `clean()` method is called.

: The cleaning process modifies the model instance passed to the `ModelForm` constructor in various ways. For instance, any date fields on the model are converted into actual date objects. Failed validation may leave the underlying model instance in an inconsistent state and therefore it's not recommended to reuse it.

Overriding the clean() method You can override the `clean()` method on a model form to provide additional validation in the same way you can on a normal form.

A model form instance attached to a model object will contain an `instance` attribute that gives its methods access to that specific model instance.

: The `ModelForm.clean()` method sets a flag that makes the *model validation* step validate the uniqueness of model fields that are marked as `unique`, `unique_together` or `unique_for_date|month|year`. If you would like to override the `clean()` method and maintain this validation, you must call the parent class's `clean()` method.

Interaction with model validation As part of the validation process, `ModelForm` will call the `clean()` method of each field on your model that has a corresponding field on your form. If you have excluded any model fields, validation will not be run on those fields. See the form validation documentation for more on how field cleaning and validation work.

The model's `clean()` method will be called before any uniqueness checks are made. See *Validating objects* for more information on the model's `clean()` hook.

Considerations regarding model's `error_messages` Error messages defined at the *`form field`* level or at the *form Meta* level always take precedence over the error messages defined at the *`model field`* level.

Error messages defined on *`model fields`* are only used when the `ValidationError` is raised during the *model validation* step and no corresponding error messages are defined at the form level.

You can override the error messages from `NON_FIELD_ERRORS` raised by model validation by adding the *`NON_FIELD_ERRORS`* key to the `error_messages` dictionary of the `ModelForm`'s inner `Meta` class:

```
from django.forms import ModelForm
from django.core.exceptions import NON_FIELD_ERRORS

class ArticleForm(ModelForm):
    class Meta:
        error_messages = {
            NON_FIELD_ERRORS: {
                'unique_together': "%(model_name)s's %(field_labels)s are not unique.",
            }
        }
```

The `save()` method Every `ModelForm` also has a `save()` method. This method creates and saves a database object from the data bound to the form. A subclass of `ModelForm` can accept an existing model instance as the keyword argument `instance`; if this is supplied, `save()` will update that instance. If it's not supplied, `save()` will create a new instance of the specified model:

```
>>> from myapp.models import Article
>>> from myapp.forms import ArticleForm

# Create a form instance from POST data.
>>> f = ArticleForm(request.POST)

# Save a new Article object from the form's data.
>>> new_article = f.save()

# Create a form to edit an existing Article, but use
# POST data to populate the form.
>>> a = Article.objects.get(pk=1)
>>> f = ArticleForm(request.POST, instance=a)
>>> f.save()
```

Note that if the form *hasn't been validated*, calling `save()` will do so by checking `form.errors`. A `ValueError` will be raised if the data in the form doesn't validate – i.e., if `form.errors` evaluates to `True`.

If an optional field doesn't appear in the form's data, the resulting model instance uses the model field *`default`*, if there is one, for that field. This behavior doesn't apply to fields that use *`CheckboxInput`* and *`CheckboxSelectMultiple`* (or any custom widget whose *`value_omitted_from_data()`* method always returns `False`) since an unchecked checkbox doesn't appear in the data of an HTML form submission. Use a custom form field or widget if you're designing an API and want the default fallback for a *`BooleanField`*.

Older versions don't have the exception for *`CheckboxInput`* which means that unchecked checkboxes receive a value of `True` if that's the model field default.

The *`value_omitted_from_data()`* method was added.

This `save()` method accepts an optional `commit` keyword argument, which accepts either `True` or `False`. If you call `save()` with `commit=False`, then it will return an object that hasn't yet been saved to the database. In this case, it's up to you to call `save()` on the resulting model instance. This is useful if you want to do custom processing on the object before saving it, or if you want to use one of the specialized *model saving options*. `commit` is `True` by default.

Another side effect of using `commit=False` is seen when your model has a many-to-many relation with another model. If your model has a many-to-many relation and you specify `commit=False` when you save a form, Django cannot immediately save the form data for the many-to-many relation. This is because it isn't possible to save many-to-many data for an instance until the instance exists in the database.

To work around this problem, every time you save a form using `commit=False`, Django adds a `save_m2m()` method to your `ModelForm` subclass. After you've manually saved the instance produced by the form, you can invoke `save_m2m()` to save the many-to-many form data. For example:

```
# Create a form instance with POST data.
>>> f = AuthorForm(request.POST)

# Create, but don't save the new author instance.
>>> new_author = f.save(commit=False)

# Modify the author in some way.
>>> new_author.some_field = 'some_value'

# Save the new instance.
>>> new_author.save()

# Now, save the many-to-many data for the form.
>>> f.save_m2m()
```

Calling `save_m2m()` is only required if you use `save(commit=False)`. When you use a simple `save()` on a form, all data – including many-to-many data – is saved without the need for any additional method calls. For example:

```
# Create a form instance with POST data.
>>> a = Author()
>>> f = AuthorForm(request.POST, instance=a)

# Create and save the new author instance. There's no need to do anything else.
>>> new_author = f.save()
```

Other than the `save()` and `save_m2m()` methods, a `ModelForm` works exactly the same way as any other `forms` form. For example, the `is_valid()` method is used to check for validity, the `is_multipart()` method is used to determine whether a form requires multipart file upload (and hence whether `request.FILES` must be passed to the form), etc. See *Binding uploaded files to a form* for more information.

Selecting the fields to use It is strongly recommended that you explicitly set all fields that should be edited in the form using the `fields` attribute. Failure to do so can easily lead to security problems when a form unexpectedly allows a user to set certain fields, especially when new fields are added to a model. Depending on how the form is rendered, the problem may not even be visible on the web page.

The alternative approach would be to include all fields automatically, or blacklist only some. This fundamental approach is known to be much less secure and has led to serious exploits on major websites (e.g. GitHub).

There are, however, two shortcuts available for cases where you can guarantee these security concerns do not apply to you:

1. Set the `fields` attribute to the special value `'__all__'` to indicate that all fields in the model should be used. For example:

```
from django.forms import ModelForm

class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = '__all__'
```

2. Set the `exclude` attribute of the `ModelForm`'s inner `Meta` class to a list of fields to be excluded from the form.

   For example:

```
class PartialAuthorForm(ModelForm):
    class Meta:
        model = Author
        exclude = ['title']
```

   Since the `Author` model has the 3 fields `name`, `title` and `birth_date`, this will result in the fields `name` and `birth_date` being present on the form.

If either of these are used, the order the fields appear in the form will be the order the fields are defined in the model, with `ManyToManyField` instances appearing last.

In addition, Django applies the following rule: if you set `editable=False` on the model field, *any* form created from the model via `ModelForm` will not include that field.

: Any fields not included in a form by the above logic will not be set by the form's `save()` method. Also, if you manually add the excluded fields back to the form, they will not be initialized from the model instance.

Django will prevent any attempt to save an incomplete model, so if the model does not allow the missing fields to be empty, and does not provide a default value for the missing fields, any attempt to `save()` a `ModelForm` with missing fields will fail. To avoid this failure, you must instantiate your model with initial values for the missing, but required fields:

```
author = Author(title='Mr')
form = PartialAuthorForm(request.POST, instance=author)
form.save()
```

Alternatively, you can use `save(commit=False)` and manually set any extra required fields:

```
form = PartialAuthorForm(request.POST)
author = form.save(commit=False)
author.title = 'Mr'
author.save()
```

See the *section on saving forms* for more details on using `save(commit=False)`.

Overriding the default fields The default field types, as described in the *Field types* table above, are sensible defaults. If you have a `DateField` in your model, chances are you'd want that to be represented as a `DateField` in your form. But `ModelForm` gives you the flexibility of changing the form field for a given model.

To specify a custom widget for a field, use the `widgets` attribute of the inner `Meta` class. This should be a dictionary mapping field names to widget classes or instances.

For example, if you want the `CharField` for the `name` attribute of `Author` to be represented by a `<textarea>` instead of its default `<input type="text">`, you can override the field's widget:

```
from django.forms import ModelForm, Textarea
from myapp.models import Author

class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ('name', 'title', 'birth_date')
        widgets = {
            'name': Textarea(attrs={'cols': 80, 'rows': 20}),
        }
```

The `widgets` dictionary accepts either widget instances (e.g., `Textarea(...)`) or classes (e.g., `Textarea`).

Similarly, you can specify the `labels`, `help_texts` and `error_messages` attributes of the inner `Meta` class if you want to further customize a field.

For example if you wanted to customize the wording of all user facing strings for the `name` field:

```
from django.utils.translation import ugettext_lazy as _

class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ('name', 'title', 'birth_date')
        labels = {
            'name': _('Writer'),
        }
        help_texts = {
            'name': _('Some useful help text.'),
        }
        error_messages = {
```

```
            'name': {
                'max_length': _("This writer's name is too long."),
            },
        }
```

You can also specify `field_classes` to customize the type of fields instantiated by the form.

For example, if you wanted to use `MySlugFormField` for the `slug` field, you could do the following:

```
from django.forms import ModelForm
from myapp.models import Article

class ArticleForm(ModelForm):
    class Meta:
        model = Article
        fields = ['pub_date', 'headline', 'content', 'reporter', 'slug']
        field_classes = {
            'slug': MySlugFormField,
        }
```

Finally, if you want complete control over of a field – including its type, validators, required, etc. – you can do this by declaratively specifying fields like you would in a regular `Form`.

If you want to specify a field's validators, you can do so by defining the field declaratively and setting its `validators` parameter:

```
from django.forms import ModelForm, CharField
from myapp.models import Article

class ArticleForm(ModelForm):
    slug = CharField(validators=[validate_slug])

    class Meta:
        model = Article
        fields = ['pub_date', 'headline', 'content', 'reporter', 'slug']
```

The `Meta.field_classes` attribute was added.

: When you explicitly instantiate a form field like this, it is important to understand how `ModelForm` and regular `Form` are related.

`ModelForm` is a regular `Form` which can automatically generate certain fields. The fields that are automatically generated depend on the content of the `Meta` class and on which fields have already been defined declaratively. Basically, `ModelForm` will **only** generate fields that are **missing** from the form, or in other words, fields that weren't defined declaratively.

Fields defined declaratively are left as-is, therefore any customizations made to `Meta` attributes such as `widgets`, `labels`, `help_texts`, or `error_messages` are ignored; these only apply to fields that are generated automatically.

Similarly, fields defined declaratively do not draw their attributes like `max_length` or `required` from the corresponding model. If you want to maintain the behavior specified in the model, you must set the relevant arguments explicitly when declaring the form field.

For example, if the `Article` model looks like this:

```
class Article(models.Model):
    headline = models.CharField(
        max_length=200,
        null=True,
```

```
        blank=True,
        help_text='Use puns liberally',
    )
    content = models.TextField()
```

and you want to do some custom validation for `headline`, while keeping the `blank` and `help_text` values as specified, you might define `ArticleForm` like this:

```
class ArticleForm(ModelForm):
    headline = MyFormField(
        max_length=200,
        required=False,
        help_text='Use puns liberally',
    )

    class Meta:
        model = Article
        fields = ['headline', 'content']
```

You must ensure that the type of the form field can be used to set the contents of the corresponding model field. When they are not compatible, you will get a `ValueError` as no implicit conversion takes place.

See the form field documentation for more information on fields and their arguments.

Enabling localization of fields By default, the fields in a `ModelForm` will not localize their data. To enable localization for fields, you can use the `localized_fields` attribute on the `Meta` class.

```
>>> from django.forms import ModelForm
>>> from myapp.models import Author
>>> class AuthorForm(ModelForm):
...     class Meta:
...         model = Author
...         localized_fields = ('birth_date',)
```

If `localized_fields` is set to the special value `'__all__'`, all fields will be localized.

Form inheritance As with basic forms, you can extend and reuse `ModelForms` by inheriting them. This is useful if you need to declare extra fields or extra methods on a parent class for use in a number of forms derived from models. For example, using the previous `ArticleForm` class:

```
>>> class EnhancedArticleForm(ArticleForm):
...     def clean_pub_date(self):
...         ...
```

This creates a form that behaves identically to `ArticleForm`, except there's some extra validation and cleaning for the `pub_date` field.

You can also subclass the parent's `Meta` inner class if you want to change the `Meta.fields` or `Meta.exclude` lists:

```
>>> class RestrictedArticleForm(EnhancedArticleForm):
...     class Meta(ArticleForm.Meta):
...         exclude = ('body',)
```

This adds the extra method from the `EnhancedArticleForm` and modifies the original `ArticleForm.Meta` to remove one field.

There are a couple of things to note, however.

- Normal Python name resolution rules apply. If you have multiple base classes that declare a `Meta` inner class, only the first one will be used. This means the child's `Meta`, if it exists, otherwise the `Meta` of the first parent, etc.
- It's possible to inherit from both `Form` and `ModelForm` simultaneously, however, you must ensure that `ModelForm` appears first in the MRO. This is because these classes rely on different metaclasses and a class can only have one metaclass.
- It's possible to declaratively remove a `Field` inherited from a parent class by setting the name to be `None` on the subclass.

  You can only use this technique to opt out from a field defined declaratively by a parent class; it won't prevent the `ModelForm` metaclass from generating a default field. To opt-out from default fields, see *Selecting the fields to use*.

Providing initial values As with regular forms, it's possible to specify initial data for forms by specifying an `initial` parameter when instantiating the form. Initial values provided this way will override both initial values from the form field and values from an attached model instance. For example:

```
>>> article = Article.objects.get(pk=1)
>>> article.headline
'My headline'
>>> form = ArticleForm(initial={'headline': 'Initial headline'}, instance=article)
>>> form['headline'].value()
'Initial headline'
```

ModelForm factory function You can create forms from a given model using the standalone function *`modelform_factory()`*, instead of using a class definition. This may be more convenient if you do not have many customizations to make:

```
>>> from django.forms import modelform_factory
>>> from myapp.models import Book
>>> BookForm = modelform_factory(Book, fields=("author", "title"))
```

This can also be used to make simple modifications to existing forms, for example by specifying the widgets to be used for a given field:

```
>>> from django.forms import Textarea
>>> Form = modelform_factory(Book, form=BookForm,
...                          widgets={"title": Textarea()})
```

The fields to include can be specified using the `fields` and `exclude` keyword arguments, or the corresponding attributes on the `ModelForm` inner `Meta` class. Please see the `ModelForm` *Selecting the fields to use* documentation.

... or enable localization for specific fields:

```
>>> Form = modelform_factory(Author, form=AuthorForm, localized_fields=("birth_date",))
```

Model formsets

class `models.`**`BaseModelFormSet`**

Like regular formsets, Django provides a couple of enhanced formset classes that make it easy to work with Django models. Let's reuse the `Author` model from above:

```
>>> from django.forms import modelformset_factory
>>> from myapp.models import Author
>>> AuthorFormSet = modelformset_factory(Author, fields=('name', 'title'))
```

Using `fields` restricts the formset to use only the given fields. Alternatively, you can take an "opt-out" approach, specifying which fields to exclude:

```
>>> AuthorFormSet = modelformset_factory(Author, exclude=('birth_date',))
```

This will create a formset that is capable of working with the data associated with the `Author` model. It works just like a regular formset:

```
>>> formset = AuthorFormSet()
>>> print(formset)
<input type="hidden" name="form-TOTAL_FORMS" value="1" id="id_form-TOTAL_FORMS" /><input type="hidden
<tr><th><label for="id_form-0-name">Name:</label></th><td><input id="id_form-0-name" type="text" name
<tr><th><label for="id_form-0-title">Title:</label></th><td><select name="form-0-title" id="id_form-0
<option value="" selected="selected">---------</option>
<option value="MR">Mr.</option>
<option value="MRS">Mrs.</option>
<option value="MS">Ms.</option>
</select><input type="hidden" name="form-0-id" id="id_form-0-id" /></td></tr>
```

: *modelformset_factory()* uses *formset_factory()* to generate formsets. This means that a model formset is just an extension of a basic formset that knows how to interact with a particular model.

Changing the queryset By default, when you create a formset from a model, the formset will use a queryset that includes all objects in the model (e.g., `Author.objects.all()`). You can override this behavior by using the `queryset` argument:

```
>>> formset = AuthorFormSet(queryset=Author.objects.filter(name__startswith='O'))
```

Alternatively, you can create a subclass that sets `self.queryset` in `__init__`:

```
from django.forms import BaseModelFormSet
from myapp.models import Author

class BaseAuthorFormSet(BaseModelFormSet):
    def __init__(self, *args, **kwargs):
        super(BaseAuthorFormSet, self).__init__(*args, **kwargs)
        self.queryset = Author.objects.filter(name__startswith='O')
```

Then, pass your `BaseAuthorFormSet` class to the factory function:

```
>>> AuthorFormSet = modelformset_factory(
...     Author, fields=('name', 'title'), formset=BaseAuthorFormSet)
```

If you want to return a formset that doesn't include *any* pre-existing instances of the model, you can specify an empty QuerySet:

```
>>> AuthorFormSet(queryset=Author.objects.none())
```

Changing the form By default, when you use `modelformset_factory`, a model form will be created using *modelform_factory()*. Often, it can be useful to specify a custom model form. For example, you can create a custom model form that has custom validation:

```
class AuthorForm(forms.ModelForm):
    class Meta:
        model = Author
        fields = ('name', 'title')

    def clean_name(self):
        # custom validation for the name field
        ...
```

Then, pass your model form to the factory function:

```
AuthorFormSet = modelformset_factory(Author, form=AuthorForm)
```

It is not always necessary to define a custom model form. The `modelformset_factory` function has several arguments which are passed through to `modelform_factory`, which are described below.

Specifying widgets to use in the form with `widgets` Using the `widgets` parameter, you can specify a dictionary of values to customize the `ModelForm`'s widget class for a particular field. This works the same way as the `widgets` dictionary on the inner `Meta` class of a `ModelForm` works:

```
>>> AuthorFormSet = modelformset_factory(
...     Author, fields=('name', 'title'),
...     widgets={'name': Textarea(attrs={'cols': 80, 'rows': 20})})
```

Enabling localization for fields with `localized_fields` Using the `localized_fields` parameter, you can enable localization for fields in the form.

```
>>> AuthorFormSet = modelformset_factory(
...     Author, fields=('name', 'title', 'birth_date'),
...     localized_fields=('birth_date',))
```

If `localized_fields` is set to the special value `'__all__'`, all fields will be localized.

Providing initial values As with regular formsets, it's possible to *specify initial data* for forms in the formset by specifying an `initial` parameter when instantiating the model formset class returned by *`modelformset_factory()`*. However, with model formsets, the initial values only apply to extra forms, those that aren't attached to an existing model instance. If the extra forms with initial data aren't changed by the user, they won't be validated or saved.

Saving objects in the formset As with a `ModelForm`, you can save the data as a model object. This is done with the formset's `save()` method:

```
# Create a formset instance with POST data.
>>> formset = AuthorFormSet(request.POST)

# Assuming all is valid, save the data.
>>> instances = formset.save()
```

The `save()` method returns the instances that have been saved to the database. If a given instance's data didn't change in the bound data, the instance won't be saved to the database and won't be included in the return value (`instances`, in the above example).

When fields are missing from the form (for example because they have been excluded), these fields will not be set by the `save()` method. You can find more information about this restriction, which also holds for regular `ModelForms`, in *Selecting the fields to use*.

Pass `commit=False` to return the unsaved model instances:

```
# don't save to the database
>>> instances = formset.save(commit=False)
>>> for instance in instances:
...     # do something with instance
...     instance.save()
```

This gives you the ability to attach data to the instances before saving them to the database. If your formset contains a `ManyToManyField`, you'll also need to call `formset.save_m2m()` to ensure the many-to-many relationships are saved properly.

After calling `save()`, your model formset will have three new attributes containing the formset's changes:

`models.BaseModelFormSet.`**`changed_objects`**

`models.BaseModelFormSet.`**`deleted_objects`**

`models.BaseModelFormSet.`**`new_objects`**

Limiting the number of editable objects As with regular formsets, you can use the `max_num` and `extra` parameters to *`modelformset_factory()`* to limit the number of extra forms displayed.

`max_num` does not prevent existing objects from being displayed:

```
>>> Author.objects.order_by('name')
<QuerySet [<Author: Charles Baudelaire>, <Author: Paul Verlaine>, <Author: Walt Whitman>]>

>>> AuthorFormSet = modelformset_factory(Author, fields=('name',), max_num=1)
>>> formset = AuthorFormSet(queryset=Author.objects.order_by('name'))
>>> [x.name for x in formset.get_queryset()]
['Charles Baudelaire', 'Paul Verlaine', 'Walt Whitman']
```

Also, `extra=0` doesn't prevent creation of new model instances as you can *add additional forms with JavaScript* or just send additional POST data. Formsets don't yet provide functionality for an "edit only" view that prevents creation of new instances.

If the value of `max_num` is greater than the number of existing related objects, up to `extra` additional blank forms will be added to the formset, so long as the total number of forms does not exceed `max_num`:

```
>>> AuthorFormSet = modelformset_factory(Author, fields=('name',), max_num=4, extra=2)
>>> formset = AuthorFormSet(queryset=Author.objects.order_by('name'))
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-name">Name:</label></th><td><input id="id_form-0-name" type="text" name
<tr><th><label for="id_form-1-name">Name:</label></th><td><input id="id_form-1-name" type="text" name
<tr><th><label for="id_form-2-name">Name:</label></th><td><input id="id_form-2-name" type="text" name
<tr><th><label for="id_form-3-name">Name:</label></th><td><input id="id_form-3-name" type="text" name
```

A `max_num` value of `None` (the default) puts a high limit on the number of forms displayed (1000). In practice this is equivalent to no limit.

Using a model formset in a view Model formsets are very similar to formsets. Let's say we want to present a formset to edit `Author` model instances:

```
from django.forms import modelformset_factory
from django.shortcuts import render
from myapp.models import Author
```

```
def manage_authors(request):
    AuthorFormSet = modelformset_factory(Author, fields=('name', 'title'))
    if request.method == 'POST':
        formset = AuthorFormSet(request.POST, request.FILES)
        if formset.is_valid():
            formset.save()
            # do something.
    else:
        formset = AuthorFormSet()
    return render(request, 'manage_authors.html', {'formset': formset})
```

As you can see, the view logic of a model formset isn't drastically different than that of a "normal" formset. The only difference is that we call `formset.save()` to save the data into the database. (This was described above, in *Saving objects in the formset*.)

Overriding `clean()` on a `ModelFormSet` Just like with `ModelForms`, by default the `clean()` method of a `ModelFormSet` will validate that none of the items in the formset violate the unique constraints on your model (either `unique`, `unique_together` or `unique_for_date|month|year`). If you want to override the `clean()` method on a `ModelFormSet` and maintain this validation, you must call the parent class's `clean` method:

```
from django.forms import BaseModelFormSet

class MyModelFormSet(BaseModelFormSet):
    def clean(self):
        super(MyModelFormSet, self).clean()
        # example custom validation across forms in the formset
        for form in self.forms:
            # your custom formset validation
            ...
```

Also note that by the time you reach this step, individual model instances have already been created for each `Form`. Modifying a value in `form.cleaned_data` is not sufficient to affect the saved value. If you wish to modify a value in `ModelFormSet.clean()` you must modify `form.instance`:

```
from django.forms import BaseModelFormSet

class MyModelFormSet(BaseModelFormSet):
    def clean(self):
        super(MyModelFormSet, self).clean()

        for form in self.forms:
            name = form.cleaned_data['name'].upper()
            form.cleaned_data['name'] = name
            # update the instance value.
            form.instance.name = name
```

Using a custom queryset As stated earlier, you can override the default queryset used by the model formset:

```
from django.forms import modelformset_factory
from django.shortcuts import render
from myapp.models import Author

def manage_authors(request):
    AuthorFormSet = modelformset_factory(Author, fields=('name', 'title'))
    if request.method == "POST":
```

```
        formset = AuthorFormSet(
            request.POST, request.FILES,
            queryset=Author.objects.filter(name__startswith='O'),
        )
        if formset.is_valid():
            formset.save()
            # Do something.
    else:
        formset = AuthorFormSet(queryset=Author.objects.filter(name__startswith='O'))
    return render(request, 'manage_authors.html', {'formset': formset})
```

Note that we pass the `queryset` argument in both the `POST` and `GET` cases in this example.

Using the formset in the template There are three ways to render a formset in a Django template.

First, you can let the formset do most of the work:

```
<form method="post" action="">
    {{ formset }}
</form>
```

Second, you can manually render the formset, but let the form deal with itself:

```
<form method="post" action="">
    {{ formset.management_form }}
    {% for form in formset %}
        {{ form }}
    {% endfor %}
</form>
```

When you manually render the forms yourself, be sure to render the management form as shown above. See the *management form documentation*.

Third, you can manually render each field:

```
<form method="post" action="">
    {{ formset.management_form }}
    {% for form in formset %}
        {% for field in form %}
            {{ field.label_tag }} {{ field }}
        {% endfor %}
    {% endfor %}
</form>
```

If you opt to use this third method and you don't iterate over the fields with a `{% for %}` loop, you'll need to render the primary key field. For example, if you were rendering the `name` and `age` fields of a model:

```
<form method="post" action="">
    {{ formset.management_form }}
    {% for form in formset %}
        {{ form.id }}
        <ul>
            <li>{{ form.name }}</li>
            <li>{{ form.age }}</li>
        </ul>
    {% endfor %}
</form>
```

Notice how we need to explicitly render `{{ form.id }}`. This ensures that the model formset, in the `POST` case, will work correctly. (This example assumes a primary key named `id`. If you've explicitly defined your own primary key that isn't called `id`, make sure it gets rendered.)

Inline formsets

class `models.`**BaseInlineFormSet**

Inline formsets is a small abstraction layer on top of model formsets. These simplify the case of working with related objects via a foreign key. Suppose you have these two models:

```
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    title = models.CharField(max_length=100)
```

If you want to create a formset that allows you to edit books belonging to a particular author, you could do this:

```
>>> from django.forms import inlineformset_factory
>>> BookFormSet = inlineformset_factory(Author, Book, fields=('title',))
>>> author = Author.objects.get(name='Mike Royko')
>>> formset = BookFormSet(instance=author)
```

: *inlineformset_factory()* uses *modelformset_factory()* and marks `can_delete=True`.

:

Manually rendered can_delete and can_order.

Overriding methods on an `InlineFormSet` When overriding methods on `InlineFormSet`, you should subclass *BaseInlineFormSet* rather than *BaseModelFormSet*.

For example, if you want to override `clean()`:

```
from django.forms import BaseInlineFormSet

class CustomInlineFormSet(BaseInlineFormSet):
    def clean(self):
        super(CustomInlineFormSet, self).clean()
        # example custom validation across forms in the formset
        for form in self.forms:
            # your custom formset validation
            ...
```

See also *Overriding clean() on a ModelFormSet*.

Then when you create your inline formset, pass in the optional argument `formset`:

```
>>> from django.forms import inlineformset_factory
>>> BookFormSet = inlineformset_factory(Author, Book, fields=('title',),
...     formset=CustomInlineFormSet)
>>> author = Author.objects.get(name='Mike Royko')
>>> formset = BookFormSet(instance=author)
```

More than one foreign key to the same model If your model contains more than one foreign key to the same model, you'll need to resolve the ambiguity manually using `fk_name`. For example, consider the following model:

```
class Friendship(models.Model):
    from_friend = models.ForeignKey(
        Friend,
        on_delete=models.CASCADE,
        related_name='from_friends',
    )
    to_friend = models.ForeignKey(
        Friend,
        on_delete=models.CASCADE,
        related_name='friends',
    )
    length_in_months = models.IntegerField()
```

To resolve this, you can use `fk_name` to *`inlineformset_factory()`*:

```
>>> FriendshipFormSet = inlineformset_factory(Friend, Friendship, fk_name='from_friend',
...     fields=('to_friend', 'length_in_months'))
```

Using an inline formset in a view You may want to provide a view that allows a user to edit the related objects of a model. Here's how you can do that:

```
def manage_books(request, author_id):
    author = Author.objects.get(pk=author_id)
    BookInlineFormSet = inlineformset_factory(Author, Book, fields=('title',))
    if request.method == "POST":
        formset = BookInlineFormSet(request.POST, request.FILES, instance=author)
        if formset.is_valid():
            formset.save()
            # Do something. Should generally end with a redirect. For example:
            return HttpResponseRedirect(author.get_absolute_url())
    else:
        formset = BookInlineFormSet(instance=author)
    return render(request, 'manage_books.html', {'formset': formset})
```

Notice how we pass `instance` in both the `POST` and `GET` cases.

Specifying widgets to use in the inline form `inlineformset_factory` uses `modelformset_factory` and passes most of its arguments to `modelformset_factory`. This means you can use the `widgets` parameter in much the same way as passing it to `modelformset_factory`. See *Specifying widgets to use in the form with widgets* above.

Form Assets (the `Media` class)

Rendering an attractive and easy-to-use Web form requires more than just HTML - it also requires CSS stylesheets, and if you want to use fancy "Web2.0" widgets, you may also need to include some JavaScript on each page. The exact combination of CSS and JavaScript that is required for any given page will depend upon the widgets that are in use on that page.

This is where asset definitions come in. Django allows you to associate different files – like stylesheets and scripts – with the forms and widgets that require those assets. For example, if you want to use a calendar to render DateFields, you can define a custom Calendar widget. This widget can then be associated with the CSS and JavaScript that is required to render the calendar. When the Calendar widget is used on a form, Django is able to identify the CSS and

JavaScript files that are required, and provide the list of file names in a form suitable for easy inclusion on your Web page.

Assets and Django Admin

The Django Admin application defines a number of customized widgets for calendars, filtered selections, and so on. These widgets define asset requirements, and the Django Admin uses the custom widgets in place of the Django defaults. The Admin templates will only include those files that are required to render the widgets on any given page.

If you like the widgets that the Django Admin application uses, feel free to use them in your own application! They're all stored in `django.contrib.admin.widgets`.

Which JavaScript toolkit?

Many JavaScript toolkits exist, and many of them include widgets (such as calendar widgets) that can be used to enhance your application. Django has deliberately avoided blessing any one JavaScript toolkit. Each toolkit has its own relative strengths and weaknesses - use whichever toolkit suits your requirements. Django is able to integrate with any JavaScript toolkit.

Assets as a static definition

The easiest way to define assets is as a static definition. Using this method, the declaration is an inner `Media` class. The properties of the inner class define the requirements.

Here's a simple example:

```
from django import forms

class CalendarWidget(forms.TextInput):
    class Media:
        css = {
            'all': ('pretty.css',)
        }
        js = ('animations.js', 'actions.js')
```

This code defines a `CalendarWidget`, which will be based on `TextInput`. Every time the CalendarWidget is used on a form, that form will be directed to include the CSS file `pretty.css`, and the JavaScript files `animations.js` and `actions.js`.

This static definition is converted at runtime into a widget property named `media`. The list of assets for a `CalendarWidget` instance can be retrieved through this property:

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="http://static.example.com/pretty.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://static.example.com/actions.js"></script>
```

Here's a list of all possible `Media` options. There are no required options.

`css` A dictionary describing the CSS files required for various forms of output media.

The values in the dictionary should be a tuple/list of file names. See *the section on paths* for details of how to specify paths to these files.

The keys in the dictionary are the output media types. These are the same types accepted by CSS files in media declarations: 'all', 'aural', 'braille', 'embossed', 'handheld', 'print', 'projection', 'screen', 'tty' and 'tv'. If you need to have different stylesheets for different media types, provide a list of CSS files for each output medium. The following example would provide two CSS options – one for the screen, and one for print:

```
class Media:
    css = {
        'screen': ('pretty.css',),
        'print': ('newspaper.css',)
    }
```

If a group of CSS files are appropriate for multiple output media types, the dictionary key can be a comma separated list of output media types. In the following example, TV's and projectors will have the same media requirements:

```
class Media:
    css = {
        'screen': ('pretty.css',),
        'tv,projector': ('lo_res.css',),
        'print': ('newspaper.css',)
    }
```

If this last CSS definition were to be rendered, it would become the following HTML:

```
<link href="http://static.example.com/pretty.css" type="text/css" media="screen" rel="stylesheet" />
<link href="http://static.example.com/lo_res.css" type="text/css" media="tv,projector" rel="styleshee
<link href="http://static.example.com/newspaper.css" type="text/css" media="print" rel="stylesheet" /
```

js A tuple describing the required JavaScript files. See *the section on paths* for details of how to specify paths to these files.

extend A boolean defining inheritance behavior for `Media` declarations.

By default, any object using a static `Media` definition will inherit all the assets associated with the parent widget. This occurs regardless of how the parent defines its own requirements. For example, if we were to extend our basic Calendar widget from the example above:

```
>>> class FancyCalendarWidget(CalendarWidget):
...     class Media:
...         css = {
...             'all': ('fancy.css',)
...         }
...         js = ('whizbang.js',)

>>> w = FancyCalendarWidget()
>>> print(w.media)
<link href="http://static.example.com/pretty.css" type="text/css" media="all" rel="stylesheet" />
<link href="http://static.example.com/fancy.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://static.example.com/actions.js"></script>
<script type="text/javascript" src="http://static.example.com/whizbang.js"></script>
```

The FancyCalendar widget inherits all the assets from its parent widget. If you don't want `Media` to be inherited in this way, add an `extend=False` declaration to the `Media` declaration:

```
>>> class FancyCalendarWidget(CalendarWidget):
...     class Media:
...         extend = False
```

```
...         css = {
...             'all': ('fancy.css',)
...         }
...         js = ('whizbang.js',)

>>> w = FancyCalendarWidget()
>>> print(w.media)
<link href="http://static.example.com/fancy.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://static.example.com/whizbang.js"></script>
```

If you require even more control over inheritance, define your assets using a *dynamic property*. Dynamic properties give you complete control over which files are inherited, and which are not.

`Media` as a dynamic property

If you need to perform some more sophisticated manipulation of asset requirements, you can define the `media` property directly. This is done by defining a widget property that returns an instance of `forms.Media`. The constructor for `forms.Media` accepts `css` and `js` keyword arguments in the same format as that used in a static media definition.

For example, the static definition for our Calendar Widget could also be defined in a dynamic fashion:

```
class CalendarWidget(forms.TextInput):
    def _media(self):
        return forms.Media(css={'all': ('pretty.css',)},
                           js=('animations.js', 'actions.js'))
    media = property(_media)
```

See the section on *Media objects* for more details on how to construct return values for dynamic `media` properties.

Paths in asset definitions

Paths used to specify assets can be either relative or absolute. If a path starts with `/`, `http://` or `https://`, it will be interpreted as an absolute path, and left as-is. All other paths will be prepended with the value of the appropriate prefix. If the *`django.contrib.staticfiles`* app is installed, it will be used to serve assets.

Whether or not you use *`django.contrib.staticfiles`*, the *`STATIC_URL`* and *`STATIC_ROOT`* settings are required to render a complete web page.

To find the appropriate prefix to use, Django will check if the *`STATIC_URL`* setting is not `None` and automatically fall back to using *`MEDIA_URL`*. For example, if the *`MEDIA_URL`* for your site was `'http://uploads.example.com/'` and *`STATIC_URL`* was `None`:

```
>>> from django import forms
>>> class CalendarWidget(forms.TextInput):
...     class Media:
...         css = {
...             'all': ('/css/pretty.css',),
...         }
...         js = ('animations.js', 'http://othersite.com/actions.js')

>>> w = CalendarWidget()
>>> print(w.media)
<link href="/css/pretty.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://uploads.example.com/animations.js"></script>
<script type="text/javascript" src="http://othersite.com/actions.js"></script>
```

But if *STATIC_URL* is 'http://static.example.com/':

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="/css/pretty.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://othersite.com/actions.js"></script>
```

Or if *staticfiles* is configured using the *~django.contib.staticfiles.ManifestStaticFilesStorage*:

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="/css/pretty.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="https://static.example.com/animations.27e20196a850.js"></script>
<script type="text/javascript" src="http://othersite.com/actions.js"></script>
```

Older versions didn't serve assets using *django.contrib.staticfiles*.

Media objects

When you interrogate the media attribute of a widget or form, the value that is returned is a forms.Media object. As we have already seen, the string representation of a Media object is the HTML required to include the relevant files in the <head> block of your HTML page.

However, Media objects have some other interesting properties.

Subsets of assets If you only want files of a particular type, you can use the subscript operator to filter out a medium of interest. For example:

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="http://static.example.com/pretty.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://static.example.com/actions.js"></script>

>>> print(w.media['css'])
<link href="http://static.example.com/pretty.css" type="text/css" media="all" rel="stylesheet" />
```

When you use the subscript operator, the value that is returned is a new Media object – but one that only contains the media of interest.

Combining Media objects Media objects can also be added together. When two Media objects are added, the resulting Media object contains the union of the assets specified by both:

```
>>> from django import forms
>>> class CalendarWidget(forms.TextInput):
...     class Media:
...         css = {
...             'all': ('pretty.css',)
...         }
...         js = ('animations.js', 'actions.js')

>>> class OtherWidget(forms.TextInput):
...     class Media:
...         js = ('whizbang.js',)
```

```
>>> w1 = CalendarWidget()
>>> w2 = OtherWidget()
>>> print(w1.media + w2.media)
<link href="http://static.example.com/pretty.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://static.example.com/actions.js"></script>
<script type="text/javascript" src="http://static.example.com/whizbang.js"></script>
```

`Media` on Forms

Widgets aren't the only objects that can have `media` definitions – forms can also define `media`. The rules for `media` definitions on forms are the same as the rules for widgets: declarations can be static or dynamic; path and inheritance rules for those declarations are exactly the same.

Regardless of whether you define a `media` declaration, *all* Form objects have a `media` property. The default value for this property is the result of adding the `media` definitions for all widgets that are part of the form:

```
>>> from django import forms
>>> class ContactForm(forms.Form):
...     date = DateField(widget=CalendarWidget)
...     name = CharField(max_length=40, widget=OtherWidget)

>>> f = ContactForm()
>>> f.media
<link href="http://static.example.com/pretty.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://static.example.com/actions.js"></script>
<script type="text/javascript" src="http://static.example.com/whizbang.js"></script>
```

If you want to associate additional assets with a form – for example, CSS for form layout – simply add a `Media` declaration to the form:

```
>>> class ContactForm(forms.Form):
...     date = DateField(widget=CalendarWidget)
...     name = CharField(max_length=40, widget=OtherWidget)
...
...     class Media:
...         css = {
...             'all': ('layout.css',)
...         }

>>> f = ContactForm()
>>> f.media
<link href="http://static.example.com/pretty.css" type="text/css" media="all" rel="stylesheet" />
<link href="http://static.example.com/layout.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://static.example.com/actions.js"></script>
<script type="text/javascript" src="http://static.example.com/whizbang.js"></script>
```

:

The Forms Reference Covers the full API reference, including form fields, form widgets, and form and field validation.

3.5 Templates

Being a web framework, Django needs a convenient way to generate HTML dynamically. The most common approach relies on templates. A template contains the static parts of the desired HTML output as well as some special syntax describing how dynamic content will be inserted. For a hands-on example of creating HTML pages with templates, see Tutorial 3.

A Django project can be configured with one or several template engines (or even zero if you don't use templates). Django ships built-in backends for its own template system, creatively called the Django template language (DTL), and for the popular alternative Jinja2. Backends for other template languages may be available from third-parties.

Django defines a standard API for loading and rendering templates regardless of the backend. Loading consists of finding the template for a given identifier and preprocessing it, usually compiling it to an in-memory representation. Rendering means interpolating the template with context data and returning the resulting string.

The Django template language is Django's own template system. Until Django 1.8 it was the only built-in option available. It's a good template library even though it's fairly opinionated and sports a few idiosyncrasies. If you don't have a pressing reason to choose another backend, you should use the DTL, especially if you're writing a pluggable application and you intend to distribute templates. Django's contrib apps that include templates, like django.contrib.admin, use the DTL.

For historical reasons, both the generic support for template engines and the implementation of the Django template language live in the `django.template` namespace.

3.5.1 Support for template engines

Configuration

Templates engines are configured with the *TEMPLATES* setting. It's a list of configurations, one for each engine. The default value is empty. The `settings.py` generated by the *startproject* command defines a more useful value:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            # ... some options here ...
        },
    },
]
```

BACKEND is a dotted Python path to a template engine class implementing Django's template backend API. The built-in backends are *django.template.backends.django.DjangoTemplates* and *django.template.backends.jinja2.Jinja2*.

Since most engines load templates from files, the top-level configuration for each engine contains two common settings:

- *DIRS* defines a list of directories where the engine should look for template source files, in search order.
- *APP_DIRS* tells whether the engine should look for templates inside installed applications. Each backend defines a conventional name for the subdirectory inside applications where its templates should be stored.

While uncommon, it's possible to configure several instances of the same backend with different options. In that case you should define a unique *NAME* for each engine.

OPTIONS contains backend-specific settings.

Usage

The `django.template.loader` module defines two functions to load templates.

get_template(*template_name*, *using=None*)
This function loads the template with the given name and returns a `Template` object.

The exact type of the return value depends on the backend that loaded the template. Each backend has its own `Template` class.

`get_template()` tries each template engine in order until one succeeds. If the template cannot be found, it raises *TemplateDoesNotExist*. If the template is found but contains invalid syntax, it raises *TemplateSyntaxError*.

How templates are searched and loaded depends on each engine's backend and configuration.

If you want to restrict the search to a particular template engine, pass the engine's *NAME* in the `using` argument.

select_template(*template_name_list*, *using=None*)
`select_template()` is just like `get_template()`, except it takes a list of template names. It tries each name in order and returns the first template that exists.

If loading a template fails, the following two exceptions, defined in `django.template`, may be raised:

exception TemplateDoesNotExist(*msg*, *tried=None*, *backend=None*, *chain=None*)
This exception is raised when a template cannot be found. It accepts the following optional arguments for populating the *template postmortem* on the debug page:

backend The template backend instance from which the exception originated.

tried A list of sources that were tried when finding the template. This is formatted as a list of tuples containing `(origin, status)`, where `origin` is an *origin-like* object and `status` is a string with the reason the template wasn't found.

chain A list of intermediate *TemplateDoesNotExist* exceptions raised when trying to load a template. This is used by functions, such as *get_template()*, that try to load a given template from multiple engines.

The `backend`, `tried`, and `chain` arguments were added.

exception TemplateSyntaxError(*msg*)
This exception is raised when a template was found but contains errors.

`Template` objects returned by `get_template()` and `select_template()` must provide a `render()` method with the following signature:

`Template.`**render**(*context=None*, *request=None*)
Renders this template with a given context.

If `context` is provided, it must be a `dict`. If it isn't provided, the engine will render the template with an empty context.

If `request` is provided, it must be an *HttpRequest*. Then the engine must make it, as well as the CSRF token, available in the template. How this is achieved is up to each backend.

Here's an example of the search algorithm. For this example the *TEMPLATES* setting is:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
```

```
            '/home/html/example.com',
            '/home/html/default',
        ],
    },
    {
        'BACKEND': 'django.template.backends.jinja2.Jinja2',
        'DIRS': [
            '/home/html/jinja2',
        ],
    },
]
```

If you call `get_template('story_detail.html')`, here are the files Django will look for, in order:

- `/home/html/example.com/story_detail.html` (`'django'` engine)
- `/home/html/default/story_detail.html` (`'django'` engine)
- `/home/html/jinja2/story_detail.html` (`'jinja2'` engine)

If you call `select_template(['story_253_detail.html', 'story_detail.html'])`, here's what Django will look for:

- `/home/html/example.com/story_253_detail.html` (`'django'` engine)
- `/home/html/default/story_253_detail.html` (`'django'` engine)
- `/home/html/jinja2/story_253_detail.html` (`'jinja2'` engine)
- `/home/html/example.com/story_detail.html` (`'django'` engine)
- `/home/html/default/story_detail.html` (`'django'` engine)
- `/home/html/jinja2/story_detail.html` (`'jinja2'` engine)

When Django finds a template that exists, it stops looking.

Tip

You can use *`select_template()`* for flexible template loading. For example, if you've written a news story and want some stories to have custom templates, use something like `select_template(['story_%s_detail.html' % story.id, 'story_detail.html'])`. That'll allow you to use a custom template for an individual story, with a fallback template for stories that don't have custom templates.

It's possible – and preferable – to organize templates in subdirectories inside each directory containing templates. The convention is to make a subdirectory for each Django app, with subdirectories within those subdirectories as needed.

Do this for your own sanity. Storing all templates in the root level of a single directory gets messy.

To load a template that's within a subdirectory, just use a slash, like so:

```
get_template('news/story_detail.html')
```

Using the same *`TEMPLATES`* option as above, this will attempt to load the following templates:

- `/home/html/example.com/news/story_detail.html` (`'django'` engine)
- `/home/html/default/news/story_detail.html` (`'django'` engine)
- `/home/html/jinja2/news/story_detail.html` (`'jinja2'` engine)

In addition, to cut down on the repetitive nature of loading and rendering templates, Django provides a shortcut function which automates the process.

render_to_string(*template_name*, *context=None*, *request=None*, *using=None*)

`render_to_string()` loads a template like *get_template()* and calls its `render()` method immediately. It takes the following arguments.

template_name The name of the template to load and render. If it's a list of template names, Django uses *select_template()* instead of *get_template()* to find the template.

context A `dict` to be used as the template's context for rendering.

request An optional *HttpRequest* that will be available during the template's rendering process.

using An optional template engine *NAME*. The search for the template will be restricted to that engine.

Usage example:

```
from django.template.loader import render_to_string
rendered = render_to_string('my_template.html', {'foo': 'bar'})
```

See also the *render()* shortcut which calls *render_to_string()* and feeds the result into an *HttpResponse* suitable for returning from a view.

Finally, you can use configured engines directly:

engines

Template engines are available in `django.template.engines`:

```
from django.template import engines

django_engine = engines['django']
template = django_engine.from_string("Hello {{ name }}!")
```

The lookup key — `'django'` in this example — is the engine's *NAME*.

Built-in backends

class DjangoTemplates

Set *BACKEND* to `'django.template.backends.django.DjangoTemplates'` to configure a Django template engine.

When *APP_DIRS* is `True`, `DjangoTemplates` engines look for templates in the `templates` subdirectory of installed applications. This generic name was kept for backwards-compatibility.

`DjangoTemplates` engines accept the following *OPTIONS*:

- `'autoescape'`: a boolean that controls whether HTML autoescaping is enabled.

  It defaults to `True`.

  : Only set it to `False` if you're rendering non-HTML templates!

  The `autoescape` option was added.

- `'context_processors'`: a list of dotted Python paths to callables that are used to populate the context when a template is rendered with a request. These callables take a request object as their argument and return a `dict` of items to be merged into the context.

  It defaults to an empty list.

  See *RequestContext* for more information.

- `'debug'`: a boolean that turns on/off template debug mode. If it is `True`, the fancy error page will display a detailed report for any exception raised during template rendering. This report contains the relevant snippet of the template with the appropriate line highlighted.

  It defaults to the value of the *DEBUG* setting.

- `'loaders'`: a list of dotted Python paths to template loader classes. Each `Loader` class knows how to import templates from a particular source. Optionally, a tuple can be used instead of a string. The first item in the tuple should be the `Loader` class name, and subsequent items are passed to the `Loader` during initialization.

  The default depends on the values of *DIRS* and *APP_DIRS*.

  See *Loader types* for details.

- `'string_if_invalid'`: the output, as a string, that the template system should use for invalid (e.g. misspelled) variables.

  It defaults to an empty string.

  See *How invalid variables are handled* for details.

- `'file_charset'`: the charset used to read template files on disk.

  It defaults to the value of *FILE_CHARSET*.

- `'libraries'`: A dictionary of labels and dotted Python paths of template tag modules to register with the template engine. This can be used to add new libraries or provide alternate labels for existing ones. For example:

```
OPTIONS={
    'libraries': {
        'myapp_tags': 'path.to.myapp.tags',
        'admin.urls': 'django.contrib.admin.templatetags.admin_urls',
    },
}
```

  Libraries can be loaded by passing the corresponding dictionary key to the *{% load %}* tag.

- `'builtins'`: A list of dotted Python paths of template tag modules to add to built-ins. For example:

```
OPTIONS={
    'builtins': ['myapp.builtins'],
}
```

  Tags and filters from built-in libraries can be used without first calling the *{% load %}* tag.

The `libraries` and `builtins` arguments were added.

class `Jinja2`

Requires Jinja2 to be installed:

```
$ pip install Jinja2
```

Set *BACKEND* to `'django.template.backends.jinja2.Jinja2'` to configure a Jinja2 engine.

When *APP_DIRS* is `True`, `Jinja2` engines look for templates in the `jinja2` subdirectory of installed applications.

The most important entry in *OPTIONS* is `'environment'`. It's a dotted Python path to a callable returning a Jinja2 environment. It defaults to `'jinja2.Environment'`. Django invokes that callable and passes other options as keyword arguments. Furthermore, Django adds defaults that differ from Jinja2's for a few options:

- `'autoescape'`: `True`
- `'loader'`: a loader configured for *DIRS* and *APP_DIRS*
- `'auto_reload'`: `settings.DEBUG`

- `'undefined': DebugUndefined if settings.DEBUG else Undefined`

The default configuration is purposefully kept to a minimum. If a template is rendered with a request (e.g. when using *render()*), the `Jinja2` backend adds the globals `request`, `csrf_input`, and `csrf_token` to the context. Apart from that, this backend doesn't create a Django-flavored environment. It doesn't know about Django context processors, filters, and tags. In order to use Django-specific APIs, you must configure them into the environment.

For example, you can create `myproject/jinja2.py` with this content:

```
from __future__ import absolute_import  # Python 2 only

from django.contrib.staticfiles.storage import staticfiles_storage
from django.urls import reverse

from jinja2 import Environment


def environment(**options):
    env = Environment(**options)
    env.globals.update({
        'static': staticfiles_storage.url,
        'url': reverse,
    })
    return env
```

and set the `'environment'` option to `'myproject.jinja2.environment'`.

Then you could use the following constructs in Jinja2 templates:

```
<img src="{{ static('path/to/company-logo.png') }}" alt="Company Logo">

<a href="{{ url('admin:index') }}">Administration</a>
```

The concepts of tags and filters exist both in the Django template language and in Jinja2 but they're used differently. Since Jinja2 supports passing arguments to callables in templates, many features that require a template tag or filter in Django templates can be achieved simply by calling a function in Jinja2 templates, as shown in the example above. Jinja2's global namespace removes the need for template context processors. The Django template language doesn't have an equivalent of Jinja2 tests.

Custom backends

Here's how to implement a custom template backend in order to use another template system. A template backend is a class that inherits `django.template.backends.base.BaseEngine`. It must implement `get_template()` and optionally `from_string()`. Here's an example for a fictional `foobar` template library:

```
from django.template import TemplateDoesNotExist, TemplateSyntaxError
from django.template.backends.base import BaseEngine
from django.template.backends.utils import csrf_input_lazy, csrf_token_lazy

import foobar


class FooBar(BaseEngine):

    # Name of the subdirectory containing the templates for this engine
    # inside an installed application.
    app_dirname = 'foobar'

    def __init__(self, params):
```

```
        params = params.copy()
        options = params.pop('OPTIONS').copy()
        super(FooBar, self).__init__(params)

        self.engine = foobar.Engine(**options)

    def from_string(self, template_code):
        try:
          return Template(self.engine.from_string(template_code))
        except foobar.TemplateCompilationFailed as exc:
            raise TemplateSyntaxError(exc.args)

    def get_template(self, template_name):
        try:
            return Template(self.engine.get_template(template_name))
        except foobar.TemplateNotFound as exc:
            raise TemplateDoesNotExist(exc.args, backend=self)
        except foobar.TemplateCompilationFailed as exc:
            raise TemplateSyntaxError(exc.args)


class Template(object):

    def __init__(self, template):
        self.template = template

    def render(self, context=None, request=None):
        if context is None:
            context = {}
        if request is not None:
            context['request'] = request
            context['csrf_input'] = csrf_input_lazy(request)
            context['csrf_token'] = csrf_token_lazy(request)
        return self.template.render(context)
```

See DEP 182 for more information.

Debug integration for custom engines

Debug page integration for non-Django template engines was added.

The Django debug page has hooks to provide detailed information when a template error arises. Custom template engines can use these hooks to enhance the traceback information that appears to users. The following hooks are available:

Template postmortem

The postmortem appears when *TemplateDoesNotExist* is raised. It lists the template engines and loaders that were used when trying to find a given template. For example, if two Django engines are configured, the postmortem will appear like:

Custom engines can populate the postmortem by passing the `backend` and `tried` arguments when raising *`TemplateDoesNotExist`*. Backends that use the postmortem *should specify an origin* on the template object.

Contextual line information

If an error happens during template parsing or rendering, Django can display the line the error happened on. For example:

Custom engines can populate this information by setting a `template_debug` attribute on exceptions raised during parsing and rendering. This attribute is a `dict` with the following values:

- `'name'`: The name of the template in which the exception occurred.
- `'message'`: The exception message.
- `'source_lines'`: The lines before, after, and including the line the exception occurred on. This is for context, so it shouldn't contain more than 20 lines or so.
- `'line'`: The line number on which the exception occurred.
- `'before'`: The content on the error line before the token that raised the error.
- `'during'`: The token that raised the error.
- `'after'`: The content on the error line after the token that raised the error.
- `'total'`: The number of lines in `source_lines`.
- `'top'`: The line number where `source_lines` starts.
- `'bottom'`: The line number where `source_lines` ends.

Given the above template error, `template_debug` would look like:

```
{
    'name': '/path/to/template.html',
    'message': "Invalid block tag: 'syntax'",
    'source_lines': [
        (1, 'some\n'),
        (2, 'lines\n'),
        (3, 'before\n'),
        (4, 'Hello {% syntax error %} {{ world }}\n'),
        (5, 'some\n'),
        (6, 'lines\n'),
        (7, 'after\n'),
        (8, ''),
    ],
    'line': 4,
    'before': 'Hello ',
    'during': '{% syntax error %}',
    'after': ' {{ world }}\n',
    'total': 9,
    'bottom': 9,
    'top': 1,
}
```

Origin API and 3rd-party integration

Django templates have an *Origin* object available through the `template.origin` attribute. This enables debug information to be displayed in the *template postmortem*, as well as in 3rd-party libraries, like the Django Debug Toolbar.

Custom engines can provide their own `template.origin` information by creating an object that specifies the following attributes:

- `'name'`: The full path to the template.
- `'template_name'`: The relative path to the template as passed into the the template loading methods.
- `'loader_name'`: An optional string identifying the function or class used to load the template, e.g. `django.template.loaders.filesystem.Loader`.

3.5.2 The Django template language

Syntax

About this section

This is an overview of the Django template language's syntax. For details see the language syntax reference.

A Django template is simply a text document or a Python string marked-up using the Django template language. Some constructs are recognized and interpreted by the template engine. The main ones are variables and tags.

A template is rendered with a context. Rendering replaces variables with their values, which are looked up in the context, and executes tags. Everything else is output as is.

The syntax of the Django template language involves four constructs.

Variables

A variable outputs a value from the context, which is a dict-like object mapping keys to values.

Variables are surrounded by {{ and }} like this:

```
My first name is {{ first_name }}. My last name is {{ last_name }}.
```

With a context of {'first_name': 'John', 'last_name': 'Doe'}, this template renders to:

```
My first name is John. My last name is Doe.
```

Dictionary lookup, attribute lookup and list-index lookups are implemented with a dot notation:

```
{{ my_dict.key }}
{{ my_object.attribute }}
{{ my_list.0 }}
```

If a variable resolves to a callable, the template system will call it with no arguments and use its result instead of the callable.

Tags

Tags provide arbitrary logic in the rendering process.

This definition is deliberately vague. For example, a tag can output content, serve as a control structure e.g. an "if" statement or a "for" loop, grab content from a database, or even enable access to other template tags.

Tags are surrounded by {% and %} like this:

```
{% csrf_token %}
```

Most tags accept arguments:

```
{% cycle 'odd' 'even' %}
```

Some tags require beginning and ending tags:

```
{% if user.is_authenticated %}Hello, {{ user.username }}.{% endif %}
```

A *reference of built-in tags* is available as well as *instructions for writing custom tags*.

Filters

Filters transform the values of variables and tag arguments.

They look like this:

```
{{ django|title }}
```

With a context of {'django': 'the web framework for perfectionists with deadlines'}, this template renders to:

```
The Web Framework For Perfectionists With Deadlines
```

Some filters take an argument:

```
{{ my_date|date:"Y-m-d" }}
```

A *reference of built-in filters* is available as well as *instructions for writing custom filters*.

Comments

Comments look like this:

```
{# this won't be rendered #}
```

A `{% comment %}` tag provides multi-line comments.

Components

About this section

This is an overview of the Django template language's APIs. For details see the API reference.

Engine

`django.template.Engine` encapsulates an instance of the Django template system. The main reason for instantiating an `Engine` directly is to use the Django template language outside of a Django project.

`django.template.backends.django.DjangoTemplates` is a thin wrapper adapting `django.template.Engine` to Django's template backend API.

Template

`django.template.Template` represents a compiled template. Templates are obtained with `Engine.get_template()` or `Engine.from_string()`

Likewise `django.template.backends.django.Template` is a thin wrapper adapting `django.template.Template` to the common template API.

Context

`django.template.Context` holds some metadata in addition to the context data. It is passed to `Template.render()` for rendering a template.

`django.template.RequestContext` is a subclass of `Context` that stores the current `HttpRequest` and runs template context processors.

The common API doesn't have an equivalent concept. Context data is passed in a plain `dict` and the current `HttpRequest` is passed separately if needed.

Loaders

Template loaders are responsible for locating templates, loading them, and returning `Template` objects.

Django provides several *built-in template loaders* and supports *custom template loaders*.

Context processors

Context processors are functions that receive the current *HttpRequest* as an argument and return a dict of data to be added to the rendering context.

Their main use is to add common data shared by all templates to the context without repeating code in every view.

Django provides many *built-in context processors*. Implementing a custom context processor is as simple as defining a function.

3.6 Class-based views

A view is a callable which takes a request and returns a response. This can be more than just a function, and Django provides an example of some classes which can be used as views. These allow you to structure your views and reuse code by harnessing inheritance and mixins. There are also some generic views for simple tasks which we'll get to later, but you may want to design your own structure of reusable views which suits your use case. For full details, see the class-based views reference documentation.

3.6.1 Introduction to class-based views

Class-based views provide an alternative way to implement views as Python objects instead of functions. They do not replace function-based views, but have certain differences and advantages when compared to function-based views:

- Organization of code related to specific HTTP methods (GET, POST, etc.) can be addressed by separate methods instead of conditional branching.
- Object oriented techniques such as mixins (multiple inheritance) can be used to factor code into reusable components.

The relationship and history of generic views, class-based views, and class-based generic views

In the beginning there was only the view function contract, Django passed your function an *HttpRequest* and expected back an *HttpResponse*. This was the extent of what Django provided.

Early on it was recognized that there were common idioms and patterns found in view development. Function-based generic views were introduced to abstract these patterns and ease view development for the common cases.

The problem with function-based generic views is that while they covered the simple cases well, there was no way to extend or customize them beyond some simple configuration options, limiting their usefulness in many real-world applications.

Class-based generic views were created with the same objective as function-based generic views, to make view development easier. However, the way the solution is implemented, through the use of mixins, provides a toolkit that results in class-based generic views being more extensible and flexible than their function-based counterparts.

If you have tried function based generic views in the past and found them lacking, you should not think of class-based generic views as simply a class-based equivalent, but rather as a fresh approach to solving the original problems that generic views were meant to solve.

The toolkit of base classes and mixins that Django uses to build class-based generic views are built for maximum flexibility, and as such have many hooks in the form of default method implementations and attributes that you are unlikely to be concerned with in the simplest use cases. For example, instead of limiting you to a class-based attribute for form_class, the implementation uses a get_form method, which calls a get_form_class method, which in its default implementation just returns the form_class attribute of the class. This gives you several options for

specifying what form to use, from a simple attribute, to a fully dynamic, callable hook. These options seem to add hollow complexity for simple situations, but without them, more advanced designs would be limited.

Using class-based views

At its core, a class-based view allows you to respond to different HTTP request methods with different class instance methods, instead of with conditionally branching code inside a single view function.

So where the code to handle HTTP `GET` in a view function would look something like:

```
from django.http import HttpResponse

def my_view(request):
    if request.method == 'GET':
        # <view logic>
        return HttpResponse('result')
```

In a class-based view, this would become:

```
from django.http import HttpResponse
from django.views import View

class MyView(View):
    def get(self, request):
        # <view logic>
        return HttpResponse('result')
```

Because Django's URL resolver expects to send the request and associated arguments to a callable function, not a class, class-based views have an *`as_view()`* class method which returns a function that can be called when a request arrives for a URL matching the associated pattern. The function creates an instance of the class and calls its *`dispatch()`* method. `dispatch` looks at the request to determine whether it is a `GET`, `POST`, etc, and relays the request to a matching method if one is defined, or raises *`HttpResponseNotAllowed`* if not:

```
# urls.py
from django.conf.urls import url
from myapp.views import MyView

urlpatterns = [
    url(r'^about/$', MyView.as_view()),
]
```

It is worth noting that what your method returns is identical to what you return from a function-based view, namely some form of *`HttpResponse`*. This means that http shortcuts or *`TemplateResponse`* objects are valid to use inside a class-based view.

While a minimal class-based view does not require any class attributes to perform its job, class attributes are useful in many class-based designs, and there are two ways to configure or set class attributes.

The first is the standard Python way of subclassing and overriding attributes and methods in the subclass. So that if your parent class had an attribute `greeting` like this:

```
from django.http import HttpResponse
from django.views import View

class GreetingView(View):
    greeting = "Good Day"

    def get(self, request):
        return HttpResponse(self.greeting)
```

You can override that in a subclass:

```
class MorningGreetingView(GreetingView):
    greeting = "Morning to ya"
```

Another option is to configure class attributes as keyword arguments to the *as_view()* call in the URLconf:

```
urlpatterns = [
    url(r'^about/$', GreetingView.as_view(greeting="G'day")),
]
```

: While your class is instantiated for each request dispatched to it, class attributes set through the *as_view()* entry point are configured only once at the time your URLs are imported.

Using mixins

Mixins are a form of multiple inheritance where behaviors and attributes of multiple parent classes can be combined.

For example, in the generic class-based views there is a mixin called *TemplateResponseMixin* whose primary purpose is to define the method *render_to_response()*. When combined with the behavior of the *View* base class, the result is a *TemplateView* class that will dispatch requests to the appropriate matching methods (a behavior defined in the `View` base class), and that has a *render_to_response()* method that uses a *template_name* attribute to return a *TemplateResponse* object (a behavior defined in the `TemplateResponseMixin`).

Mixins are an excellent way of reusing code across multiple classes, but they come with some cost. The more your code is scattered among mixins, the harder it will be to read a child class and know what exactly it is doing, and the harder it will be to know which methods from which mixins to override if you are subclassing something that has a deep inheritance tree.

Note also that you can only inherit from one generic view - that is, only one parent class may inherit from *View* and the rest (if any) should be mixins. Trying to inherit from more than one class that inherits from `View` - for example, trying to use a form at the top of a list and combining *ProcessFormView* and *ListView* - won't work as expected.

Handling forms with class-based views

A basic function-based view that handles forms may look something like this:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render

from .forms import MyForm

def myview(request):
    if request.method == "POST":
        form = MyForm(request.POST)
        if form.is_valid():
            # <process form cleaned data>
            return HttpResponseRedirect('/success/')
    else:
        form = MyForm(initial={'key': 'value'})

    return render(request, 'form_template.html', {'form': form})
```

A similar class-based view might look like:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from django.views import View

from .forms import MyForm

class MyFormView(View):
    form_class = MyForm
    initial = {'key': 'value'}
    template_name = 'form_template.html'

    def get(self, request, *args, **kwargs):
        form = self.form_class(initial=self.initial)
        return render(request, self.template_name, {'form': form})

    def post(self, request, *args, **kwargs):
        form = self.form_class(request.POST)
        if form.is_valid():
            # <process form cleaned data>
            return HttpResponseRedirect('/success/')

        return render(request, self.template_name, {'form': form})
```

This is a very simple case, but you can see that you would then have the option of customizing this view by overriding any of the class attributes, e.g. `form_class`, via URLconf configuration, or subclassing and overriding one or more of the methods (or both!).

Decorating class-based views

The extension of class-based views isn't limited to using mixins. You can also use decorators. Since class-based views aren't functions, decorating them works differently depending on if you're using `as_view()` or creating a subclass.

Decorating in URLconf

The simplest way of decorating class-based views is to decorate the result of the *`as_view()`* method. The easiest place to do this is in the URLconf where you deploy your view:

```
from django.contrib.auth.decorators import login_required, permission_required
from django.views.generic import TemplateView

from .views import VoteView

urlpatterns = [
    url(r'^about/$', login_required(TemplateView.as_view(template_name="secret.html"))),
    url(r'^vote/$', permission_required('polls.can_vote')(VoteView.as_view())),
]
```

This approach applies the decorator on a per-instance basis. If you want every instance of a view to be decorated, you need to take a different approach.

Decorating the class

To decorate every instance of a class-based view, you need to decorate the class definition itself. To do this you apply the decorator to the *`dispatch()`* method of the class.

A method on a class isn't quite the same as a standalone function, so you can't just apply a function decorator to the method – you need to transform it into a method decorator first. The `method_decorator` decorator transforms a function decorator into a method decorator so that it can be used on an instance method. For example:

```
from django.contrib.auth.decorators import login_required
from django.utils.decorators import method_decorator
from django.views.generic import TemplateView

class ProtectedView(TemplateView):
    template_name = 'secret.html'

    @method_decorator(login_required)
    def dispatch(self, *args, **kwargs):
        return super(ProtectedView, self).dispatch(*args, **kwargs)
```

Or, more succinctly, you can decorate the class instead and pass the name of the method to be decorated as the keyword argument `name`:

```
@method_decorator(login_required, name='dispatch')
class ProtectedView(TemplateView):
    template_name = 'secret.html'
```

If you have a set of common decorators used in several places, you can define a list or tuple of decorators and use this instead of invoking `method_decorator()` multiple times. These two classes are equivalent:

```
decorators = [never_cache, login_required]

@method_decorator(decorators, name='dispatch')
class ProtectedView(TemplateView):
    template_name = 'secret.html'

@method_decorator(never_cache, name='dispatch')
@method_decorator(login_required, name='dispatch')
class ProtectedView(TemplateView):
    template_name = 'secret.html'
```

The decorators will process a request in the order they are passed to the decorator. In the example, `never_cache()` will process the request before `login_required()`.

The ability to use `method_decorator()` on a class and the ability for it to accept a list or tuple of decorators were added.

In this example, every instance of `ProtectedView` will have login protection.

: `method_decorator` passes `*args` and `**kwargs` as parameters to the decorated method on the class. If your method does not accept a compatible set of parameters it will raise a `TypeError` exception.

3.6.2 Built-in class-based generic views

Writing Web applications can be monotonous, because we repeat certain patterns again and again. Django tries to take away some of that monotony at the model and template layers, but Web developers also experience this boredom at the view level.

Django's *generic views* were developed to ease that pain. They take certain common idioms and patterns found in view development and abstract them so that you can quickly write common views of data without having to write too much code.

We can recognize certain common tasks, like displaying a list of objects, and write code that displays a list of *any* object. Then the model in question can be passed as an extra argument to the URLconf.

Django ships with generic views to do the following:

- Display list and detail pages for a single object. If we were creating an application to manage conferences then a `TalkListView` and a `RegisteredUserListView` would be examples of list views. A single talk page is an example of what we call a "detail" view.
- Present date-based objects in year/month/day archive pages, associated detail, and "latest" pages.
- Allow users to create, update, and delete objects – with or without authorization.

Taken together, these views provide easy interfaces to perform the most common tasks developers encounter.

Extending generic views

There's no question that using generic views can speed up development substantially. In most projects, however, there comes a moment when the generic views no longer suffice. Indeed, the most common question asked by new Django developers is how to make generic views handle a wider array of situations.

This is one of the reasons generic views were redesigned for the 1.3 release - previously, they were just view functions with a bewildering array of options; now, rather than passing in a large amount of configuration in the URLconf, the recommended way to extend generic views is to subclass them, and override their attributes or methods.

That said, generic views will have a limit. If you find you're struggling to implement your view as a subclass of a generic view, then you may find it more effective to write just the code you need, using your own class-based or functional views.

More examples of generic views are available in some third party applications, or you could write your own as needed.

Generic views of objects

TemplateView certainly is useful, but Django's generic views really shine when it comes to presenting views of your database content. Because it's such a common task, Django comes with a handful of built-in generic views that make generating list and detail views of objects incredibly easy.

Let's start by looking at some examples of showing a list of objects or an individual object.

We'll be using these models:

```
# models.py
from django.db import models

class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    class Meta:
        ordering = ["-name"]

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Author(models.Model):
```

```
    salutation = models.CharField(max_length=10)
    name = models.CharField(max_length=200)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='author_headshots')

    def __str__(self):              # __unicode__ on Python 2
        return self.name

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField('Author')
    publisher = models.ForeignKey(Publisher, on_delete=models.CASCADE)
    publication_date = models.DateField()
```

Now we need to define a view:

```
# views.py
from django.views.generic import ListView
from books.models import Publisher

class PublisherList(ListView):
    model = Publisher
```

Finally hook that view into your urls:

```
# urls.py
from django.conf.urls import url
from books.views import PublisherList

urlpatterns = [
    url(r'^publishers/$', PublisherList.as_view()),
]
```

That's all the Python code we need to write. We still need to write a template, however. We could explicitly tell the view which template to use by adding a `template_name` attribute to the view, but in the absence of an explicit template Django will infer one from the object's name. In this case, the inferred template will be `"books/publisher_list.html"` – the "books" part comes from the name of the app that defines the model, while the "publisher" bit is just the lowercased version of the model's name.

: Thus, when (for example) the `APP_DIRS` option of a `DjangoTemplates` backend is set to True in *TEMPLATES*, a template location could be: /path/to/project/books/templates/books/publisher_list.html

---

This template will be rendered against a context containing a variable called `object_list` that contains all the publisher objects. A very simple template might look like the following:

```
{% extends "base.html" %}

{% block content %}
    <h2>Publishers</h2>
    <ul>
        {% for publisher in object_list %}
            <li>{{ publisher.name }}</li>
        {% endfor %}
    </ul>
{% endblock %}
```

That's really all there is to it. All the cool features of generic views come from changing the attributes set on the

generic view. The generic views reference documents all the generic views and their options in detail; the rest of this document will consider some of the common ways you might customize and extend generic views.

Making "friendly" template contexts

You might have noticed that our sample publisher list template stores all the publishers in a variable named `object_list`. While this works just fine, it isn't all that "friendly" to template authors: they have to "just know" that they're dealing with publishers here.

Well, if you're dealing with a model object, this is already done for you. When you are dealing with an object or queryset, Django is able to populate the context using the lower cased version of the model class' name. This is provided in addition to the default `object_list` entry, but contains exactly the same data, i.e. `publisher_list`.

If this still isn't a good match, you can manually set the name of the context variable. The `context_object_name` attribute on a generic view specifies the context variable to use:

```
# views.py
from django.views.generic import ListView
from books.models import Publisher

class PublisherList(ListView):
    model = Publisher
    context_object_name = 'my_favorite_publishers'
```

Providing a useful `context_object_name` is always a good idea. Your coworkers who design templates will thank you.

Adding extra context

Often you simply need to present some extra information beyond that provided by the generic view. For example, think of showing a list of all the books on each publisher detail page. The *DetailView* generic view provides the publisher to the context, but how do we get additional information in that template?

The answer is to subclass *DetailView* and provide your own implementation of the `get_context_data` method. The default implementation simply adds the object being displayed to the template, but you can override it to send more:

```
from django.views.generic import DetailView
from books.models import Publisher, Book

class PublisherDetail(DetailView):

    model = Publisher

    def get_context_data(self, **kwargs):
        # Call the base implementation first to get a context
        context = super(PublisherDetail, self).get_context_data(**kwargs)
        # Add in a QuerySet of all the books
        context['book_list'] = Book.objects.all()
        return context
```

: Generally, `get_context_data` will merge the context data of all parent classes with those of the current class. To preserve this behavior in your own classes where you want to alter the context, you should be sure to call `get_context_data` on the super class. When no two classes try to define the same key, this will give the expected results. However if any class attempts to override a key after parent classes have set it (after the call to super), any

children of that class will also need to explicitly set it after super if they want to be sure to override all parents. If you're having trouble, review the method resolution order of your view.

Another consideration is that the context data from class-based generic views will override data provided by context processors; see *get_context_data()* for an example.

Viewing subsets of objects

Now let's take a closer look at the `model` argument we've been using all along. The `model` argument, which specifies the database model that the view will operate upon, is available on all the generic views that operate on a single object or a collection of objects. However, the `model` argument is not the only way to specify the objects that the view will operate upon – you can also specify the list of objects using the `queryset` argument:

```
from django.views.generic import DetailView
from books.models import Publisher

class PublisherDetail(DetailView):

    context_object_name = 'publisher'
    queryset = Publisher.objects.all()
```

Specifying `model = Publisher` is really just shorthand for saying `queryset = Publisher.objects.all()`. However, by using `queryset` to define a filtered list of objects you can be more specific about the objects that will be visible in the view (see Making queries for more information about *QuerySet* objects, and see the class-based views reference for the complete details).

To pick a simple example, we might want to order a list of books by publication date, with the most recent first:

```
from django.views.generic import ListView
from books.models import Book

class BookList(ListView):
    queryset = Book.objects.order_by('-publication_date')
    context_object_name = 'book_list'
```

That's a pretty simple example, but it illustrates the idea nicely. Of course, you'll usually want to do more than just reorder objects. If you want to present a list of books by a particular publisher, you can use the same technique:

```
from django.views.generic import ListView
from books.models import Book

class AcmeBookList(ListView):

    context_object_name = 'book_list'
    queryset = Book.objects.filter(publisher__name='ACME Publishing')
    template_name = 'books/acme_list.html'
```

Notice that along with a filtered `queryset`, we're also using a custom template name. If we didn't, the generic view would use the same template as the "vanilla" object list, which might not be what we want.

Also notice that this isn't a very elegant way of doing publisher-specific books. If we want to add another publisher page, we'd need another handful of lines in the URLconf, and more than a few publishers would get unreasonable. We'll deal with this problem in the next section.

: If you get a 404 when requesting `/books/acme/`, check to ensure you actually have a Publisher with the name 'ACME Publishing'. Generic views have an `allow_empty` parameter for this case. See the class-based-views

reference for more details.

Dynamic filtering

Another common need is to filter down the objects given in a list page by some key in the URL. Earlier we hard-coded the publisher's name in the URLconf, but what if we wanted to write a view that displayed all the books by some arbitrary publisher?

Handily, the `ListView` has a *`get_queryset()`* method we can override. Previously, it has just been returning the value of the `queryset` attribute, but now we can add more logic.

The key part to making this work is that when class-based views are called, various useful things are stored on `self`; as well as the request (`self.request`) this includes the positional (`self.args`) and name-based (`self.kwargs`) arguments captured according to the URLconf.

Here, we have a URLconf with a single captured group:

```
# urls.py
from django.conf.urls import url
from books.views import PublisherBookList

urlpatterns = [
    url(r'^books/([\w-]+)/$', PublisherBookList.as_view()),
]
```

Next, we'll write the `PublisherBookList` view itself:

```
# views.py
from django.shortcuts import get_object_or_404
from django.views.generic import ListView
from books.models import Book, Publisher

class PublisherBookList(ListView):

    template_name = 'books/books_by_publisher.html'

    def get_queryset(self):
        self.publisher = get_object_or_404(Publisher, name=self.args[0])
        return Book.objects.filter(publisher=self.publisher)
```

As you can see, it's quite easy to add more logic to the queryset selection; if we wanted, we could use `self.request.user` to filter using the current user, or other more complex logic.

We can also add the publisher into the context at the same time, so we can use it in the template:

```
# ...

def get_context_data(self, **kwargs):
    # Call the base implementation first to get a context
    context = super(PublisherBookList, self).get_context_data(**kwargs)
    # Add in the publisher
    context['publisher'] = self.publisher
    return context
```

Performing extra work

The last common pattern we'll look at involves doing some extra work before or after calling the generic view.

Imagine we had a `last_accessed` field on our `Author` model that we were using to keep track of the last time anybody looked at that author:

```
# models.py
from django.db import models

class Author(models.Model):
    salutation = models.CharField(max_length=10)
    name = models.CharField(max_length=200)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='author_headshots')
    last_accessed = models.DateTimeField()
```

The generic `DetailView` class, of course, wouldn't know anything about this field, but once again we could easily write a custom view to keep that field updated.

First, we'd need to add an author detail bit in the URLconf to point to a custom view:

```
from django.conf.urls import url
from books.views import AuthorDetailView

urlpatterns = [
    #...
    url(r'^authors/(?P<pk>[0-9]+)/$', AuthorDetailView.as_view(), name='author-detail'),
]
```

Then we'd write our new view – `get_object` is the method that retrieves the object – so we simply override it and wrap the call:

```
from django.views.generic import DetailView
from django.utils import timezone
from books.models import Author

class AuthorDetailView(DetailView):

    queryset = Author.objects.all()

    def get_object(self):
        # Call the superclass
        object = super(AuthorDetailView, self).get_object()
        # Record the last accessed date
        object.last_accessed = timezone.now()
        object.save()
        # Return the object
        return object
```

: The URLconf here uses the named group `pk` - this name is the default name that `DetailView` uses to find the value of the primary key used to filter the queryset.

If you want to call the group something else, you can set `pk_url_kwarg` on the view. More details can be found in the reference for *DetailView*

3.6.3 Form handling with class-based views

Form processing generally has 3 paths:

- Initial GET (blank or prepopulated form)

- POST with invalid data (typically redisplay form with errors)
- POST with valid data (process the data and typically redirect)

Implementing this yourself often results in a lot of repeated boilerplate code (see *Using a form in a view*). To help avoid this, Django provides a collection of generic class-based views for form processing.

Basic forms

Given a simple contact form:

forms.py

```
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField()
    message = forms.CharField(widget=forms.Textarea)

    def send_email(self):
        # send email using the self.cleaned_data dictionary
        pass
```

The view can be constructed using a `FormView`:

views.py

```
from myapp.forms import ContactForm
from django.views.generic.edit import FormView

class ContactView(FormView):
    template_name = 'contact.html'
    form_class = ContactForm
    success_url = '/thanks/'

    def form_valid(self, form):
        # This method is called when valid form data has been POSTed.
        # It should return an HttpResponse.
        form.send_email()
        return super(ContactView, self).form_valid(form)
```

Notes:

- FormView inherits *`TemplateResponseMixin`* so *`template_name`* can be used here.
- The default implementation for *`form_valid()`* simply redirects to the *`success_url`*.

Model forms

Generic views really shine when working with models. These generic views will automatically create a *`ModelForm`*, so long as they can work out which model class to use:

- If the *`model`* attribute is given, that model class will be used.
- If *`get_object()`* returns an object, the class of that object will be used.
- If a *`queryset`* is given, the model for that queryset will be used.

Model form views provide a *`form_valid()`* implementation that saves the model automatically. You can override this if you have any special requirements; see below for examples.

You don't even need to provide a `success_url` for *`CreateView`* or *`UpdateView`* - they will use *`get_absolute_url()`* on the model object if available.

If you want to use a custom *`ModelForm`* (for instance to add extra validation) simply set *`form_class`* on your view.

: When specifying a custom form class, you must still specify the model, even though the *`form_class`* may be a *`ModelForm`*.

First we need to add *`get_absolute_url()`* to our `Author` class:

models.py

```
from django.urls import reverse
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=200)

    def get_absolute_url(self):
        return reverse('author-detail', kwargs={'pk': self.pk})
```

Then we can use *`CreateView`* and friends to do the actual work. Notice how we're just configuring the generic class-based views here; we don't have to write any logic ourselves:

views.py

```
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from django.urls import reverse_lazy
from myapp.models import Author

class AuthorCreate(CreateView):
    model = Author
    fields = ['name']

class AuthorUpdate(UpdateView):
    model = Author
    fields = ['name']

class AuthorDelete(DeleteView):
    model = Author
    success_url = reverse_lazy('author-list')
```

: We have to use *`reverse_lazy()`* here, not just `reverse()` as the urls are not loaded when the file is imported.

The `fields` attribute works the same way as the `fields` attribute on the inner `Meta` class on *`ModelForm`*. Unless you define the form class in another way, the attribute is required and the view will raise an *`ImproperlyConfigured`* exception if it's not.

If you specify both the *`fields`* and *`form_class`* attributes, an *`ImproperlyConfigured`* exception will be raised.

Finally, we hook these new views into the URLconf:

urls.py

```
from django.conf.urls import url
from myapp.views import AuthorCreate, AuthorUpdate, AuthorDelete

urlpatterns = [
    # ...
    url(r'author/add/$', AuthorCreate.as_view(), name='author-add'),
    url(r'author/(?P<pk>[0-9]+)/$', AuthorUpdate.as_view(), name='author-update'),
    url(r'author/(?P<pk>[0-9]+)/delete/$', AuthorDelete.as_view(), name='author-delete'),
]
```

: These views inherit *SingleObjectTemplateResponseMixin* which uses *template_name_suffix* to construct the *template_name* based on the model.

In this example:

- *CreateView* and *UpdateView* use `myapp/author_form.html`
- *DeleteView* uses `myapp/author_confirm_delete.html`

If you wish to have separate templates for *CreateView* and *UpdateView*, you can set either *template_name* or *template_name_suffix* on your view class.

Models and `request.user`

To track the user that created an object using a *CreateView*, you can use a custom *ModelForm* to do this. First, add the foreign key relation to the model:

`models.py`

```
from django.contrib.auth.models import User
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=200)
    created_by = models.ForeignKey(User, on_delete=models.CASCADE)

    # ...
```

In the view, ensure that you don't include `created_by` in the list of fields to edit, and override *form_valid()* to add the user:

`views.py`

```
from django.views.generic.edit import CreateView
from myapp.models import Author

class AuthorCreate(CreateView):
    model = Author
    fields = ['name']

    def form_valid(self, form):
        form.instance.created_by = self.request.user
        return super(AuthorCreate, self).form_valid(form)
```

Note that you'll need to *decorate this view* using *login_required()*, or alternatively handle unauthorized users in the *form_valid()*.

AJAX example

Here is a simple example showing how you might go about implementing a form that works for AJAX requests as well as 'normal' form POSTs:

```
from django.http import JsonResponse
from django.views.generic.edit import CreateView
from myapp.models import Author

class AjaxableResponseMixin(object):
    """
    Mixin to add AJAX support to a form.
    Must be used with an object-based FormView (e.g. CreateView)
    """
    def form_invalid(self, form):
        response = super(AjaxableResponseMixin, self).form_invalid(form)
        if self.request.is_ajax():
            return JsonResponse(form.errors, status=400)
        else:
            return response

    def form_valid(self, form):
        # We make sure to call the parent's form_valid() method because
        # it might do some processing (in the case of CreateView, it will
        # call form.save() for example).
        response = super(AjaxableResponseMixin, self).form_valid(form)
        if self.request.is_ajax():
            data = {
                'pk': self.object.pk,
            }
            return JsonResponse(data)
        else:
            return response

class AuthorCreate(AjaxableResponseMixin, CreateView):
    model = Author
    fields = ['name']
```

3.6.4 Using mixins with class-based views

: This is an advanced topic. A working knowledge of Django's class-based views is advised before exploring these techniques.

Django's built-in class-based views provide a lot of functionality, but some of it you may want to use separately. For instance, you may want to write a view that renders a template to make the HTTP response, but you can't use *TemplateView*; perhaps you need to render a template only on `POST`, with `GET` doing something else entirely. While you could use *TemplateResponse* directly, this will likely result in duplicate code.

For this reason, Django also provides a number of mixins that provide more discrete functionality. Template rendering, for instance, is encapsulated in the *TemplateResponseMixin*. The Django reference documentation contains full documentation of all the mixins.

Context and template responses

Two central mixins are provided that help in providing a consistent interface to working with templates in class-based views.

TemplateResponseMixin Every built in view which returns a *TemplateResponse* will call the *render_to_response()* method that TemplateResponseMixin provides. Most of the time this will be called for you (for instance, it is called by the get() method implemented by both *TemplateView* and *DetailView*); similarly, it's unlikely that you'll need to override it, although if you want your response to return something not rendered via a Django template then you'll want to do it. For an example of this, see the *JSONResponseMixin example*.

render_to_response() itself calls *get_template_names()*, which by default will just look up *template_name* on the class-based view; two other mixins (*SingleObjectTemplateResponseMixin* and *MultipleObjectTemplateResponseMixin*) override this to provide more flexible defaults when dealing with actual objects.

ContextMixin Every built in view which needs context data, such as for rendering a template (including TemplateResponseMixin above), should call *get_context_data()* passing any data they want to ensure is in there as keyword arguments. get_context_data() returns a dictionary; in ContextMixin it simply returns its keyword arguments, but it is common to override this to add more members to the dictionary.

Building up Django's generic class-based views

Let's look at how two of Django's generic class-based views are built out of mixins providing discrete functionality. We'll consider *DetailView*, which renders a "detail" view of an object, and *ListView*, which will render a list of objects, typically from a queryset, and optionally paginate them. This will introduce us to four mixins which between them provide useful functionality when working with either a single Django object, or multiple objects.

There are also mixins involved in the generic edit views (*FormView*, and the model-specific views *CreateView*, *UpdateView* and *DeleteView*), and in the date-based generic views. These are covered in the mixin reference documentation.

DetailView: working with a single Django object

To show the detail of an object, we basically need to do two things: we need to look up the object and then we need to make a *TemplateResponse* with a suitable template, and that object as context.

To get the object, *DetailView* relies on *SingleObjectMixin*, which provides a *get_object()* method that figures out the object based on the URL of the request (it looks for pk and slug keyword arguments as declared in the URLConf, and looks the object up either from the *model* attribute on the view, or the *queryset* attribute if that's provided). SingleObjectMixin also overrides *get_context_data()*, which is used across all Django's built in class-based views to supply context data for template renders.

To then make a *TemplateResponse*, *DetailView* uses *SingleObjectTemplateResponseMixin*, which extends *TemplateResponseMixin*, overriding *get_template_names()* as discussed above. It actually provides a fairly sophisticated set of options, but the main one that most people are going to use is <app_label>/<model_name>_detail.html. The _detail part can be changed by setting *template_name_suffix* on a subclass to something else. (For instance, the generic edit views use _form for create and update views, and _confirm_delete for delete views.)

ListView: working with many Django objects

Lists of objects follow roughly the same pattern: we need a (possibly paginated) list of objects, typically a *QuerySet*, and then we need to make a *TemplateResponse* with a suitable template using that list of objects.

To get the objects, *ListView* uses *MultipleObjectMixin*, which provides both *get_queryset()* and *paginate_queryset()*. Unlike with *SingleObjectMixin*, there's no need to key off parts of the URL to figure out the queryset to work with, so the default just uses the *queryset* or *model* attribute on the view class. A common reason to override *get_queryset()* here would be to dynamically vary the objects, such as depending on the current user or to exclude posts in the future for a blog.

MultipleObjectMixin also overrides *get_context_data()* to include appropriate context variables for pagination (providing dummies if pagination is disabled). It relies on `object_list` being passed in as a keyword argument, which *ListView* arranges for it.

To make a *TemplateResponse*, *ListView* then uses *MultipleObjectTemplateResponseMixin*; as with *SingleObjectTemplateResponseMixin* above, this overrides `get_template_names()` to provide *a range of options*, with the most commonly-used being `<app_label>/<model_name>_list.html`, with the `_list` part again being taken from the *template_name_suffix* attribute. (The date based generic views use suffixes such as `_archive`, `_archive_year` and so on to use different templates for the various specialized date-based list views.)

Using Django's class-based view mixins

Now we've seen how Django's generic class-based views use the provided mixins, let's look at other ways we can combine them. Of course we're still going to be combining them with either built-in class-based views, or other generic class-based views, but there are a range of rarer problems you can solve than are provided for by Django out of the box.

: Not all mixins can be used together, and not all generic class based views can be used with all other mixins. Here we present a few examples that do work; if you want to bring together other functionality then you'll have to consider interactions between attributes and methods that overlap between the different classes you're using, and how method resolution order will affect which versions of the methods will be called in what order.
The reference documentation for Django's class-based views and class-based view mixins will help you in understanding which attributes and methods are likely to cause conflict between different classes and mixins.
If in doubt, it's often better to back off and base your work on *View* or *TemplateView*, perhaps with *SingleObjectMixin* and *MultipleObjectMixin*. Although you will probably end up writing more code, it is more likely to be clearly understandable to someone else coming to it later, and with fewer interactions to worry about you will save yourself some thinking. (Of course, you can always dip into Django's implementation of the generic class-based views for inspiration on how to tackle problems.)

Using `SingleObjectMixin` with View

If we want to write a simple class-based view that responds only to `POST`, we'll subclass *View* and write a `post()` method in the subclass. However if we want our processing to work on a particular object, identified from the URL, we'll want the functionality provided by *SingleObjectMixin*.

We'll demonstrate this with the `Author` model we used in the generic class-based views introduction.

views.py

```
from django.http import HttpResponseForbidden, HttpResponseRedirect
from django.urls import reverse
from django.views import View
from django.views.generic.detail import SingleObjectMixin
from books.models import Author

class RecordInterest(SingleObjectMixin, View):
    """Records the current user's interest in an author."""
    model = Author
```

```
    def post(self, request, *args, **kwargs):
        if not request.user.is_authenticated:
            return HttpResponseForbidden()

        # Look up the author we're interested in.
        self.object = self.get_object()
        # Actually record interest somehow here!

        return HttpResponseRedirect(reverse('author-detail', kwargs={'pk': self.object.pk}))
```

In practice you'd probably want to record the interest in a key-value store rather than in a relational database, so we've left that bit out. The only bit of the view that needs to worry about using *SingleObjectMixin* is where we want to look up the author we're interested in, which it just does with a simple call to `self.get_object()`. Everything else is taken care of for us by the mixin.

We can hook this into our URLs easily enough:

urls.py

```
from django.conf.urls import url
from books.views import RecordInterest

urlpatterns = [
    #...
    url(r'^author/(?P<pk>[0-9]+)/interest/$', RecordInterest.as_view(), name='author-interest'),
]
```

Note the `pk` named group, which *`get_object()`* uses to look up the `Author` instance. You could also use a slug, or any of the other features of *SingleObjectMixin*.

Using `SingleObjectMixin` with `ListView`

ListView provides built-in pagination, but you might want to paginate a list of objects that are all linked (by a foreign key) to another object. In our publishing example, you might want to paginate through all the books by a particular publisher.

One way to do this is to combine *ListView* with *SingleObjectMixin*, so that the queryset for the paginated list of books can hang off the publisher found as the single object. In order to do this, we need to have two different querysets:

`Book` queryset for use by *`ListView`* Since we have access to the `Publisher` whose books we want to list, we simply override `get_queryset()` and use the `Publisher`'s *reverse foreign key manager*.

`Publisher` queryset for use in *`get_object()`* We'll rely on the default implementation of `get_object()` to fetch the correct `Publisher` object. However, we need to explicitly pass a `queryset` argument because otherwise the default implementation of `get_object()` would call `get_queryset()` which we have overridden to return `Book` objects instead of `Publisher` ones.

: We have to think carefully about `get_context_data()`. Since both *SingleObjectMixin* and *ListView* will put things in the context data under the value of `context_object_name` if it's set, we'll instead explicitly ensure the `Publisher` is in the context data. *ListView* will add in the suitable `page_obj` and `paginator` for us providing we remember to call `super()`.

Now we can write a new `PublisherDetail`:

```
from django.views.generic import ListView
from django.views.generic.detail import SingleObjectMixin
from books.models import Publisher

class PublisherDetail(SingleObjectMixin, ListView):
    paginate_by = 2
    template_name = "books/publisher_detail.html"

    def get(self, request, *args, **kwargs):
        self.object = self.get_object(queryset=Publisher.objects.all())
        return super(PublisherDetail, self).get(request, *args, **kwargs)

    def get_context_data(self, **kwargs):
        context = super(PublisherDetail, self).get_context_data(**kwargs)
        context['publisher'] = self.object
        return context

    def get_queryset(self):
        return self.object.book_set.all()
```

Notice how we set `self.object` within `get()` so we can use it again later in `get_context_data()` and `get_queryset()`. If you don't set `template_name`, the template will default to the normal *`ListView`* choice, which in this case would be `"books/book_list.html"` because it's a list of books; *`ListView`* knows nothing about *`SingleObjectMixin`*, so it doesn't have any clue this view is anything to do with a `Publisher`.

The `paginate_by` is deliberately small in the example so you don't have to create lots of books to see the pagination working! Here's the template you'd want to use:

```
{% extends "base.html" %}

{% block content %}
    <h2>Publisher {{ publisher.name }}</h2>

    <ol>
      {% for book in page_obj %}
        <li>{{ book.title }}</li>
      {% endfor %}
    </ol>

    <div class="pagination">
        <span class="step-links">
            {% if page_obj.has_previous %}
                <a href="?page={{ page_obj.previous_page_number }}">previous</a>
            {% endif %}

            <span class="current">
                Page {{ page_obj.number }} of {{ paginator.num_pages }}.
            </span>

            {% if page_obj.has_next %}
                <a href="?page={{ page_obj.next_page_number }}">next</a>
            {% endif %}
        </span>
    </div>
{% endblock %}
```

Avoid anything more complex

Generally you can use *TemplateResponseMixin* and *SingleObjectMixin* when you need their functionality. As shown above, with a bit of care you can even combine `SingleObjectMixin` with *ListView*. However things get increasingly complex as you try to do so, and a good rule of thumb is:

: Each of your views should use only mixins or views from one of the groups of generic class-based views: detail, list, editing and date. For example it's fine to combine *TemplateView* (built in view) with *MultipleObjectMixin* (generic list), but you're likely to have problems combining `SingleObjectMixin` (generic detail) with `MultipleObjectMixin` (generic list).

To show what happens when you try to get more sophisticated, we show an example that sacrifices readability and maintainability when there is a simpler solution. First, let's look at a naive attempt to combine *DetailView* with *FormMixin* to enable us to `POST` a Django *Form* to the same URL as we're displaying an object using *DetailView*.

Using `FormMixin` with `DetailView`

Think back to our earlier example of using *View* and *SingleObjectMixin* together. We were recording a user's interest in a particular author; say now that we want to let them leave a message saying why they like them. Again, let's assume we're not going to store this in a relational database but instead in something more esoteric that we won't worry about here.

At this point it's natural to reach for a *Form* to encapsulate the information sent from the user's browser to Django. Say also that we're heavily invested in REST, so we want to use the same URL for displaying the author as for capturing the message from the user. Let's rewrite our `AuthorDetailView` to do that.

We'll keep the `GET` handling from *DetailView*, although we'll have to add a *Form* into the context data so we can render it in the template. We'll also want to pull in form processing from *FormMixin*, and write a bit of code so that on `POST` the form gets called appropriately.

: We use *FormMixin* and implement `post()` ourselves rather than try to mix *DetailView* with *FormView* (which provides a suitable `post()` already) because both of the views implement `get()`, and things would get much more confusing.

Our new `AuthorDetail` looks like this:

```
# CAUTION: you almost certainly do not want to do this.
# It is provided as part of a discussion of problems you can
# run into when combining different generic class-based view
# functionality that is not designed to be used together.

from django import forms
from django.http import HttpResponseForbidden
from django.urls import reverse
from django.views.generic import DetailView
from django.views.generic.edit import FormMixin
from books.models import Author

class AuthorInterestForm(forms.Form):
    message = forms.CharField()

class AuthorDetail(FormMixin, DetailView):
    model = Author
```

```
    form_class = AuthorInterestForm

    def get_success_url(self):
        return reverse('author-detail', kwargs={'pk': self.object.pk})

    def get_context_data(self, **kwargs):
        context = super(AuthorDetail, self).get_context_data(**kwargs)
        context['form'] = self.get_form()
        return context

    def post(self, request, *args, **kwargs):
        if not request.user.is_authenticated:
            return HttpResponseForbidden()
        self.object = self.get_object()
        form = self.get_form()
        if form.is_valid():
            return self.form_valid(form)
        else:
            return self.form_invalid(form)

    def form_valid(self, form):
        # Here, we would record the user's interest using the message
        # passed in form.cleaned_data['message']
        return super(AuthorDetail, self).form_valid(form)
```

`get_success_url()` is just providing somewhere to redirect to, which gets used in the default implementation of `form_valid()`. We have to provide our own `post()` as noted earlier, and override `get_context_data()` to make the *`Form`* available in the context data.

A better solution

It should be obvious that the number of subtle interactions between *`FormMixin`* and *`DetailView`* is already testing our ability to manage things. It's unlikely you'd want to write this kind of class yourself.

In this case, it would be fairly easy to just write the `post()` method yourself, keeping *`DetailView`* as the only generic functionality, although writing *`Form`* handling code involves a lot of duplication.

Alternatively, it would still be easier than the above approach to have a separate view for processing the form, which could use *`FormView`* distinct from *`DetailView`* without concerns.

An alternative better solution

What we're really trying to do here is to use two different class based views from the same URL. So why not do just that? We have a very clear division here: `GET` requests should get the *`DetailView`* (with the *`Form`* added to the context data), and `POST` requests should get the *`FormView`*. Let's set up those views first.

The `AuthorDisplay` view is almost the same as *when we first introduced AuthorDetail*; we have to write our own `get_context_data()` to make the `AuthorInterestForm` available to the template. We'll skip the `get_object()` override from before for clarity:

```
from django.views.generic import DetailView
from django import forms
from books.models import Author

class AuthorInterestForm(forms.Form):
    message = forms.CharField()
```

```
class AuthorDisplay(DetailView):
    model = Author

    def get_context_data(self, **kwargs):
        context = super(AuthorDisplay, self).get_context_data(**kwargs)
        context['form'] = AuthorInterestForm()
        return context
```

Then the `AuthorInterest` is a simple *`FormView`*, but we have to bring in *`SingleObjectMixin`* so we can find the author we're talking about, and we have to remember to set `template_name` to ensure that form errors will render the same template as `AuthorDisplay` is using on `GET`:

```
from django.urls import reverse
from django.http import HttpResponseForbidden
from django.views.generic import FormView
from django.views.generic.detail import SingleObjectMixin

class AuthorInterest(SingleObjectMixin, FormView):
    template_name = 'books/author_detail.html'
    form_class = AuthorInterestForm
    model = Author

    def post(self, request, *args, **kwargs):
        if not request.user.is_authenticated:
            return HttpResponseForbidden()
        self.object = self.get_object()
        return super(AuthorInterest, self).post(request, *args, **kwargs)

    def get_success_url(self):
        return reverse('author-detail', kwargs={'pk': self.object.pk})
```

Finally we bring this together in a new `AuthorDetail` view. We already know that calling *`as_view()`* on a class-based view gives us something that behaves exactly like a function based view, so we can do that at the point we choose between the two subviews.

You can of course pass through keyword arguments to *`as_view()`* in the same way you would in your URLconf, such as if you wanted the `AuthorInterest` behavior to also appear at another URL but using a different template:

```
from django.views import View

class AuthorDetail(View):

    def get(self, request, *args, **kwargs):
        view = AuthorDisplay.as_view()
        return view(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        view = AuthorInterest.as_view()
        return view(request, *args, **kwargs)
```

This approach can also be used with any other generic class-based views or your own class-based views inheriting directly from *`View`* or *`TemplateView`*, as it keeps the different views as separate as possible.

More than just HTML

Where class-based views shine is when you want to do the same thing many times. Suppose you're writing an API, and every view should return JSON instead of rendered HTML.

We can create a mixin class to use in all of our views, handling the conversion to JSON once.

For example, a simple JSON mixin might look something like this:

```
from django.http import JsonResponse

class JSONResponseMixin(object):
    """
    A mixin that can be used to render a JSON response.
    """
    def render_to_json_response(self, context, **response_kwargs):
        """
        Returns a JSON response, transforming 'context' to make the payload.
        """
        return JsonResponse(
            self.get_data(context),
            **response_kwargs
        )

    def get_data(self, context):
        """
        Returns an object that will be serialized as JSON by json.dumps().
        """
        # Note: This is *EXTREMELY* naive; in reality, you'll need
        # to do much more complex handling to ensure that arbitrary
        # objects -- such as Django model instances or querysets
        # -- can be serialized as JSON.
        return context
```

: Check out the Serializing Django objects documentation for more information on how to correctly transform Django models and querysets into JSON.

This mixin provides a `render_to_json_response()` method with the same signature as *`render_to_response()`*. To use it, we simply need to mix it into a `TemplateView` for example, and override `render_to_response()` to call `render_to_json_response()` instead:

```
from django.views.generic import TemplateView

class JSONView(JSONResponseMixin, TemplateView):
    def render_to_response(self, context, **response_kwargs):
        return self.render_to_json_response(context, **response_kwargs)
```

Equally we could use our mixin with one of the generic views. We can make our own version of *`DetailView`* by mixing `JSONResponseMixin` with the `django.views.generic.detail.BaseDetailView` – (the *`DetailView`* before template rendering behavior has been mixed in):

```
from django.views.generic.detail import BaseDetailView

class JSONDetailView(JSONResponseMixin, BaseDetailView):
    def render_to_response(self, context, **response_kwargs):
        return self.render_to_json_response(context, **response_kwargs)
```

This view can then be deployed in the same way as any other *`DetailView`*, with exactly the same behavior – except for the format of the response.

If you want to be really adventurous, you could even mix a *`DetailView`* subclass that is able to return *both* HTML and JSON content, depending on some property of the HTTP request, such as a query argument or a HTTP header. Just mix in both the `JSONResponseMixin` and a *`SingleObjectTemplateResponseMixin`*, and override

the implementation of *render_to_response()* to defer to the appropriate rendering method depending on the type of response that the user requested:

```
from django.views.generic.detail import SingleObjectTemplateResponseMixin

class HybridDetailView(JSONResponseMixin, SingleObjectTemplateResponseMixin, BaseDetailView):
    def render_to_response(self, context):
        # Look for a 'format=json' GET argument
        if self.request.GET.get('format') == 'json':
            return self.render_to_json_response(context)
        else:
            return super(HybridDetailView, self).render_to_response(context)
```

Because of the way that Python resolves method overloading, the call to `super(HybridDetailView, self).render_to_response(context)` ends up calling the *render_to_response()* implementation of *TemplateResponseMixin*.

3.6.5 Basic examples

Django provides base view classes which will suit a wide range of applications. All views inherit from the *View* class, which handles linking the view in to the URLs, HTTP method dispatching and other simple features. *RedirectView* is for a simple HTTP redirect, and *TemplateView* extends the base class to make it also render a template.

3.6.6 Simple usage in your URLconf

The simplest way to use generic views is to create them directly in your URLconf. If you're only changing a few simple attributes on a class-based view, you can simply pass them into the *as_view()* method call itself:

```
from django.conf.urls import url
from django.views.generic import TemplateView

urlpatterns = [
    url(r'^about/$', TemplateView.as_view(template_name="about.html")),
]
```

Any arguments passed to *as_view()* will override attributes set on the class. In this example, we set `template_name` on the `TemplateView`. A similar overriding pattern can be used for the `url` attribute on *RedirectView*.

3.6.7 Subclassing generic views

The second, more powerful way to use generic views is to inherit from an existing view and override attributes (such as the `template_name`) or methods (such as `get_context_data`) in your subclass to provide new values or methods. Consider, for example, a view that just displays one template, `about.html`. Django has a generic view to do this - *TemplateView* - so we can just subclass it, and override the template name:

```
# some_app/views.py
from django.views.generic import TemplateView

class AboutView(TemplateView):
    template_name = "about.html"
```

Then we just need to add this new view into our URLconf. *TemplateView* is a class, not a function, so we point the URL to the *as_view()* class method instead, which provides a function-like entry to class-based views:

```
# urls.py
from django.conf.urls import url
from some_app.views import AboutView

urlpatterns = [
    url(r'^about/$', AboutView.as_view()),
]
```

For more information on how to use the built in generic views, consult the next topic on generic class-based views.

Supporting other HTTP methods

Suppose somebody wants to access our book library over HTTP using the views as an API. The API client would connect every now and then and download book data for the books published since last visit. But if no new books appeared since then, it is a waste of CPU time and bandwidth to fetch the books from the database, render a full response and send it to the client. It might be preferable to ask the API when the most recent book was published.

We map the URL to book list view in the URLconf:

```
from django.conf.urls import url
from books.views import BookListView

urlpatterns = [
    url(r'^books/$', BookListView.as_view()),
]
```

And the view:

```
from django.http import HttpResponse
from django.views.generic import ListView
from books.models import Book

class BookListView(ListView):
    model = Book

    def head(self, *args, **kwargs):
        last_book = self.get_queryset().latest('publication_date')
        response = HttpResponse('')
        # RFC 1123 date format
        response['Last-Modified'] = last_book.publication_date.strftime('%a, %d %b %Y %H:%M:%S GMT')
        return response
```

If the view is accessed from a `GET` request, a plain-and-simple object list is returned in the response (using `book_list.html` template). But if the client issues a `HEAD` request, the response has an empty body and the `Last-Modified` header indicates when the most recent book was published. Based on this information, the client may or may not download the full object list.

3.7 Migrations

Migrations are Django's way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your database schema. They're designed to be mostly automatic, but you'll need to know when to make migrations, when to run them, and the common problems you might run into.

3.7.1 The Commands

There are several commands which you will use to interact with migrations and Django's handling of database schema:

- *migrate*, which is responsible for applying migrations, as well as unapplying and listing their status.
- *makemigrations*, which is responsible for creating new migrations based on the changes you have made to your models.
- *sqlmigrate*, which displays the SQL statements for a migration.
- *showmigrations*, which lists a project's migrations.

You should think of migrations as a version control system for your database schema. `makemigrations` is responsible for packaging up your model changes into individual migration files - analogous to commits - and `migrate` is responsible for applying those to your database.

The migration files for each app live in a "migrations" directory inside of that app, and are designed to be committed to, and distributed as part of, its codebase. You should be making them once on your development machine and then running the same migrations on your colleagues' machines, your staging machines, and eventually your production machines.

: It is possible to override the name of the package which contains the migrations on a per-app basis by modifying the *MIGRATION_MODULES* setting.

Migrations will run the same way on the same dataset and produce consistent results, meaning that what you see in development and staging is, under the same circumstances, exactly what will happen in production.

Django will make migrations for any change to your models or fields - even options that don't affect the database - as the only way it can reconstruct a field correctly is to have all the changes in the history, and you might need those options in some data migrations later on (for example, if you've set custom validators).

3.7.2 Backend Support

Migrations are supported on all backends that Django ships with, as well as any third-party backends if they have programmed in support for schema alteration (done via the SchemaEditor class).

However, some databases are more capable than others when it comes to schema migrations; some of the caveats are covered below.

PostgreSQL

PostgreSQL is the most capable of all the databases here in terms of schema support; the only caveat is that adding columns with default values will cause a full rewrite of the table, for a time proportional to its size.

For this reason, it's recommended you always create new columns with `null=True`, as this way they will be added immediately.

MySQL

MySQL lacks support for transactions around schema alteration operations, meaning that if a migration fails to apply you will have to manually unpick the changes in order to try again (it's impossible to roll back to an earlier point).

In addition, MySQL will fully rewrite tables for almost every schema operation and generally takes a time proportional to the number of rows in the table to add or remove columns. On slower hardware this can be worse than a minute

per million rows - adding a few columns to a table with just a few million rows could lock your site up for over ten minutes.

Finally, MySQL has reasonably small limits on name lengths for columns, tables and indexes, as well as a limit on the combined size of all columns an index covers. This means that indexes that are possible on other backends will fail to be created under MySQL.

SQLite

SQLite has very little built-in schema alteration support, and so Django attempts to emulate it by:

- Creating a new table with the new schema
- Copying the data across
- Dropping the old table
- Renaming the new table to match the original name

This process generally works well, but it can be slow and occasionally buggy. It is not recommended that you run and migrate SQLite in a production environment unless you are very aware of the risks and its limitations; the support Django ships with is designed to allow developers to use SQLite on their local machines to develop less complex Django projects without the need for a full database.

3.7.3 Workflow

Working with migrations is simple. Make changes to your models - say, add a field and remove a model - and then run *makemigrations*:

```
$ python manage.py makemigrations
Migrations for 'books':
  books/migrations/0003_auto.py:
    - Alter field author on book
```

Your models will be scanned and compared to the versions currently contained in your migration files, and then a new set of migrations will be written out. Make sure to read the output to see what `makemigrations` thinks you have changed - it's not perfect, and for complex changes it might not be detecting what you expect.

Once you have your new migration files, you should apply them to your database to make sure they work as expected:

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: books
Running migrations:
  Rendering model states... DONE
  Applying books.0003_auto... OK
```

Once the migration is applied, commit the migration and the models change to your version control system as a single commit - that way, when other developers (or your production servers) check out the code, they'll get both the changes to your models and the accompanying migration at the same time.

If you want to give the migration(s) a meaningful name instead of a generated one, you can use the *makemigrations --name* option:

```
$ python manage.py makemigrations --name changed_my_model your_app_label
```

Version control

Because migrations are stored in version control, you'll occasionally come across situations where you and another developer have both committed a migration to the same app at the same time, resulting in two migrations with the same number.

Don't worry - the numbers are just there for developers' reference, Django just cares that each migration has a different name. Migrations specify which other migrations they depend on - including earlier migrations in the same app - in the file, so it's possible to detect when there's two new migrations for the same app that aren't ordered.

When this happens, Django will prompt you and give you some options. If it thinks it's safe enough, it will offer to automatically linearize the two migrations for you. If not, you'll have to go in and modify the migrations yourself - don't worry, this isn't difficult, and is explained more in *Migration files* below.

3.7.4 Dependencies

While migrations are per-app, the tables and relationships implied by your models are too complex to be created for just one app at a time. When you make a migration that requires something else to run - for example, you add a `ForeignKey` in your `books` app to your `authors` app - the resulting migration will contain a dependency on a migration in `authors`.

This means that when you run the migrations, the `authors` migration runs first and creates the table the `ForeignKey` references, and then the migration that makes the `ForeignKey` column runs afterwards and creates the constraint. If this didn't happen, the migration would try to create the `ForeignKey` column without the table it's referencing existing and your database would throw an error.

This dependency behavior affects most migration operations where you restrict to a single app. Restricting to a single app (either in `makemigrations` or `migrate`) is a best-efforts promise, and not a guarantee; any other apps that need to be used to get dependencies correct will be.

3.7.5 Migration files

Migrations are stored as an on-disk format, referred to here as "migration files". These files are actually just normal Python files with an agreed-upon object layout, written in a declarative style.

A basic migration file looks like this:

```
from django.db import migrations, models

class Migration(migrations.Migration):

    dependencies = [("migrations", "0001_initial")]

    operations = [
        migrations.DeleteModel("Tribble"),
        migrations.AddField("Author", "rating", models.IntegerField(default=0)),
    ]
```

What Django looks for when it loads a migration file (as a Python module) is a subclass of `django.db.migrations.Migration` called `Migration`. It then inspects this object for four attributes, only two of which are used most of the time:

- `dependencies`, a list of migrations this one depends on.
- `operations`, a list of `Operation` classes that define what this migration does.

The operations are the key; they are a set of declarative instructions which tell Django what schema changes need to be made. Django scans them and builds an in-memory representation of all of the schema changes to all apps, and uses this to generate the SQL which makes the schema changes.

That in-memory structure is also used to work out what the differences are between your models and the current state of your migrations; Django runs through all the changes, in order, on an in-memory set of models to come up with the state of your models last time you ran `makemigrations`. It then uses these models to compare against the ones in your `models.py` files to work out what you have changed.

You should rarely, if ever, need to edit migration files by hand, but it's entirely possible to write them manually if you need to. Some of the more complex operations are not autodetectable and are only available via a hand-written migration, so don't be scared about editing them if you have to.

Custom fields

You can't modify the number of positional arguments in an already migrated custom field without raising a `TypeError`. The old migration will call the modified `__init__` method with the old signature. So if you need a new argument, please create a keyword argument and add something like `assert 'argument_name' in kwargs` in the constructor.

Model managers

You can optionally serialize managers into migrations and have them available in *`RunPython`* operations. This is done by defining a `use_in_migrations` attribute on the manager class:

```
class MyManager(models.Manager):
    use_in_migrations = True

class MyModel(models.Model):
    objects = MyManager()
```

If you are using the *`from_queryset()`* function to dynamically generate a manager class, you need to inherit from the generated class to make it importable:

```
class MyManager(MyBaseManager.from_queryset(CustomQuerySet)):
    use_in_migrations = True

class MyModel(models.Model):
    objects = MyManager()
```

Please refer to the notes about *Historical models* in migrations to see the implications that come along.

Initial migrations

`Migration.`**`initial`**

The "initial migrations" for an app are the migrations that create the first version of that app's tables. Usually an app will have just one initial migration, but in some cases of complex model interdependencies it may have two or more.

Initial migrations are marked with an `initial = True` class attribute on the migration class. If an `initial` class attribute isn't found, a migration will be considered "initial" if it is the first migration in the app (i.e. if it has no dependencies on any other migration in the same app).

When the *`migrate --fake-initial`* option is used, these initial migrations are treated specially. For an initial migration that creates one or more tables (`CreateModel` operation), Django checks that all of those tables already exist in the database and fake-applies the migration if so. Similarly, for an initial migration that adds one or more

fields (`AddField` operation), Django checks that all of the respective columns already exist in the database and fake-applies the migration if so. Without `--fake-initial`, initial migrations are treated no differently from any other migration.

History consistency

As previously discussed, you may need to linearize migrations manually when two development branches are joined. While editing migration dependencies, you can inadvertently create an inconsistent history state where a migration has been applied but some of its dependencies haven't. This is a strong indication that the dependencies are incorrect, so Django will refuse to run migrations or make new migrations until it's fixed. When using multiple databases, you can use the *`allow_migrate()`* method of *database routers* to control which databases *`makemigrations`* checks for consistent history.

Migration consistency checks were added. Checks based on database routers were added in 1.10.1.

3.7.6 Adding migrations to apps

Adding migrations to new apps is straightforward - they come preconfigured to accept migrations, and so just run *`makemigrations`* once you've made some changes.

If your app already has models and database tables, and doesn't have migrations yet (for example, you created it against a previous Django version), you'll need to convert it to use migrations; this is a simple process:

```
$ python manage.py makemigrations your_app_label
```

This will make a new initial migration for your app. Now, run `python manage.py migrate --fake-initial`, and Django will detect that you have an initial migration *and* that the tables it wants to create already exist, and will mark the migration as already applied. (Without the *`migrate --fake-initial`* flag, the command would error out because the tables it wants to create already exist.)

Note that this only works given two things:

- You have not changed your models since you made their tables. For migrations to work, you must make the initial migration *first* and then make changes, as Django compares changes against migration files, not the database.
- You have not manually edited your database - Django won't be able to detect that your database doesn't match your models, you'll just get errors when migrations try to modify those tables.

3.7.7 Historical models

When you run migrations, Django is working from historical versions of your models stored in the migration files. If you write Python code using the *`RunPython`* operation, or if you have `allow_migrate` methods on your database routers, you will be exposed to these versions of your models.

Because it's impossible to serialize arbitrary Python code, these historical models will not have any custom methods that you have defined. They will, however, have the same fields, relationships, managers (limited to those with `use_in_migrations = True`) and `Meta` options (also versioned, so they may be different from your current ones).

: This means that you will NOT have custom `save()` methods called on objects when you access them in migrations, and you will NOT have any custom constructors or instance methods. Plan appropriately!

References to functions in field options such as `upload_to` and `limit_choices_to` and model manager declarations with managers having `use_in_migrations = True` are serialized in migrations, so the functions and

classes will need to be kept around for as long as there is a migration referencing them. Any custom model fields will also need to be kept, since these are imported directly by migrations.

In addition, the base classes of the model are just stored as pointers, so you must always keep base classes around for as long as there is a migration that contains a reference to them. On the plus side, methods and managers from these base classes inherit normally, so if you absolutely need access to these you can opt to move them into a superclass.

To remove old references, you can *squash migrations* or, if there aren't many references, copy them into the migration files.

3.7.8 Considerations when removing model fields

Similar to the "references to historical functions" considerations described in the previous section, removing custom model fields from your project or third-party app will cause a problem if they are referenced in old migrations.

To help with this situation, Django provides some model field attributes to assist with model field deprecation using the system checks framework.

Add the `system_check_deprecated_details` attribute to your model field similar to the following:

```
class IPAddressField(Field):
    system_check_deprecated_details = {
        'msg': (
            'IPAddressField has been deprecated. Support for it (except '
            'in historical migrations) will be removed in Django 1.9.'
        ),
        'hint': 'Use GenericIPAddressField instead.',  # optional
        'id': 'fields.W900',  # pick a unique ID for your field.
    }
```

After a deprecation period of your choosing (two or three feature releases for fields in Django itself), change the `system_check_deprecated_details` attribute to `system_check_removed_details` and update the dictionary similar to:

```
class IPAddressField(Field):
    system_check_removed_details = {
        'msg': (
            'IPAddressField has been removed except for support in '
            'historical migrations.'
        ),
        'hint': 'Use GenericIPAddressField instead.',
        'id': 'fields.E900',  # pick a unique ID for your field.
    }
```

You should keep the field's methods that are required for it to operate in database migrations such as `__init__()`, `deconstruct()`, and `get_internal_type()`. Keep this stub field for as long as any migrations which reference the field exist. For example, after squashing migrations and removing the old ones, you should be able to remove the field completely.

3.7.9 Data Migrations

As well as changing the database schema, you can also use migrations to change the data in the database itself, in conjunction with the schema if you want.

Migrations that alter data are usually called "data migrations"; they're best written as separate migrations, sitting alongside your schema migrations.

Django can't automatically generate data migrations for you, as it does with schema migrations, but it's not very hard to write them. Migration files in Django are made up of Operations, and the main operation you use for data migrations is *RunPython*.

To start, make an empty migration file you can work from (Django will put the file in the right place, suggest a name, and add dependencies for you):

```
python manage.py makemigrations --empty yourappname
```

Then, open up the file; it should look something like this:

```
# -*- coding: utf-8 -*-
# Generated by Django A.B on YYYY-MM-DD HH:MM
from __future__ import unicode_literals

from django.db import migrations, models

class Migration(migrations.Migration):
    initial = True

    dependencies = [
        ('yourappname', '0001_initial'),
    ]

    operations = [
    ]
```

Now, all you need to do is create a new function and have *RunPython* use it. *RunPython* expects a callable as its argument which takes two arguments - the first is an app registry that has the historical versions of all your models loaded into it to match where in your history the migration sits, and the second is a SchemaEditor, which you can use to manually effect database schema changes (but beware, doing this can confuse the migration autodetector!)

Let's write a simple migration that populates our new `name` field with the combined values of `first_name` and `last_name` (we've come to our senses and realized that not everyone has first and last names). All we need to do is use the historical model and iterate over the rows:

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.db import migrations, models

def combine_names(apps, schema_editor):
    # We can't import the Person model directly as it may be a newer
    # version than this migration expects. We use the historical version.
    Person = apps.get_model("yourappname", "Person")
    for person in Person.objects.all():
        person.name = "%s %s" % (person.first_name, person.last_name)
        person.save()

class Migration(migrations.Migration):
    initial = True

    dependencies = [
        ('yourappname', '0001_initial'),
    ]

    operations = [
        migrations.RunPython(combine_names),
    ]
```

Once that's done, we can just run `python manage.py migrate` as normal and the data migration will run in place alongside other migrations.

You can pass a second callable to *RunPython* to run whatever logic you want executed when migrating backwards. If this callable is omitted, migrating backwards will raise an exception.

Accessing models from other apps

When writing a `RunPython` function that uses models from apps other than the one in which the migration is located, the migration's `dependencies` attribute should include the latest migration of each app that is involved, otherwise you may get an error similar to: `LookupError: No installed app with label 'myappname'` when you try to retrieve the model in the `RunPython` function using `apps.get_model()`.

In the following example, we have a migration in `app1` which needs to use models in `app2`. We aren't concerned with the details of `move_m1` other than the fact it will need to access models from both apps. Therefore we've added a dependency that specifies the last migration of `app2`:

```
class Migration(migrations.Migration):

    dependencies = [
        ('app1', '0001_initial'),
        # added dependency to enable using models from app2 in move_m1
        ('app2', '0004_foobar'),
    ]

    operations = [
        migrations.RunPython(move_m1),
    ]
```

More advanced migrations

If you're interested in the more advanced migration operations, or want to be able to write your own, see the migration operations reference and the "how-to" on writing migrations.

3.7.10 Squashing migrations

You are encouraged to make migrations freely and not worry about how many you have; the migration code is optimized to deal with hundreds at a time without much slowdown. However, eventually you will want to move back from having several hundred migrations to just a few, and that's where squashing comes in.

Squashing is the act of reducing an existing set of many migrations down to one (or sometimes a few) migrations which still represent the same changes.

Django does this by taking all of your existing migrations, extracting their `Operations` and putting them all in sequence, and then running an optimizer over them to try and reduce the length of the list - for example, it knows that *CreateModel* and *DeleteModel* cancel each other out, and it knows that *AddField* can be rolled into *CreateModel*.

Once the operation sequence has been reduced as much as possible - the amount possible depends on how closely intertwined your models are and if you have any *RunSQL* or *RunPython* operations (which can't be optimized through) - Django will then write it back out into a new set of migration files.

These files are marked to say they replace the previously-squashed migrations, so they can coexist with the old migration files, and Django will intelligently switch between them depending where you are in the history. If you're still part-way through the set of migrations that you squashed, it will keep using them until it hits the end and then switch to the squashed history, while new installs will just use the new squashed migration and skip all the old ones.

This enables you to squash and not mess up systems currently in production that aren't fully up-to-date yet. The recommended process is to squash, keeping the old files, commit and release, wait until all systems are upgraded with the new release (or if you're a third-party project, just ensure your users upgrade releases in order without skipping any), and then remove the old files, commit and do a second release.

The command that backs all this is *squashmigrations* - just pass it the app label and migration name you want to squash up to, and it'll get to work:

```
$ ./manage.py squashmigrations myapp 0004
Will squash the following migrations:
 - 0001_initial
 - 0002_some_change
 - 0003_another_change
 - 0004_undo_something
Do you wish to proceed? [yN] y
Optimizing...
  Optimized from 12 operations to 7 operations.
Created new squashed migration /home/andrew/Programs/DjangoTest/test/migrations/0001_squashed_0004_u
  You should commit this migration but leave the old ones in place;
  the new migration will be used for new installs. Once you are sure
  all instances of the codebase have applied the migrations you squashed,
  you can delete them.
```

Note that model interdependencies in Django can get very complex, and squashing may result in migrations that do not run; either mis-optimized (in which case you can try again with `--no-optimize`, though you should also report an issue), or with a `CircularDependencyError`, in which case you can manually resolve it.

To manually resolve a `CircularDependencyError`, break out one of the ForeignKeys in the circular dependency loop into a separate migration, and move the dependency on the other app with it. If you're unsure, see how makemigrations deals with the problem when asked to create brand new migrations from your models. In a future release of Django, squashmigrations will be updated to attempt to resolve these errors itself.

Once you've squashed your migration, you should then commit it alongside the migrations it replaces and distribute this change to all running instances of your application, making sure that they run `migrate` to store the change in their database.

You must then transition the squashed migration to a normal migration by:

- Deleting all the migration files it replaces.
- Updating all migrations that depend on the deleted migrations to depend on the squashed migration instead.
- Removing the `replaces` attribute in the `Migration` class of the squashed migration (this is how Django tells that it is a squashed migration).

: Once you've squashed a migration, you should not then re-squash that squashed migration until you have fully transitioned it to a normal migration.

3.7.11 Serializing values

Migrations are just Python files containing the old definitions of your models - thus, to write them, Django must take the current state of your models and serialize them out into a file.

While Django can serialize most things, there are some things that we just can't serialize out into a valid Python representation - there's no Python standard for how a value can be turned back into code (`repr()` only works for basic values, and doesn't specify import paths).

Django can serialize the following:

- `int`, `long`, `float`, `bool`, `str`, `unicode`, `bytes`, `None`
- `list`, `set`, `tuple`, `dict`
- `datetime.date`, `datetime.time`, and `datetime.datetime` instances (include those that are timezone-aware)
- `decimal.Decimal` instances
- `enum.Enum` instances
- `functools.partial` instances which have serializable `func`, `args`, and `keywords` values.
- `LazyObject` instances which wrap a serializable value.
- Any Django field
- Any function or method reference (e.g. `datetime.datetime.today`) (must be in module's top-level scope)
- Any class reference (must be in module's top-level scope)
- Anything with a custom `deconstruct()` method (*see below*)

Serialization support for `functools.partial` and `LazyObject` instances was added.

Serialization support for `enum.Enum` was added.

Django can serialize the following on Python 3 only:

- Unbound methods used from within the class body (see below)

Django cannot serialize:

- Nested classes
- Arbitrary class instances (e.g. `MyClass(4.3, 5.7)`)
- Lambdas

Due to the fact `__qualname__` was only introduced in Python 3, Django can only serialize the following pattern (an unbound method used within the class body) on Python 3, and will fail to serialize a reference to it on Python 2:

```
class MyModel(models.Model):

    def upload_to(self):
        return "something dynamic"

    my_file = models.FileField(upload_to=upload_to)
```

If you are using Python 2, we recommend you move your methods for upload_to and similar arguments that accept callables (e.g. `default`) to live in the main module body, rather than the class body.

Adding a `deconstruct()` method

You can let Django serialize your own custom class instances by giving the class a `deconstruct()` method. It takes no arguments, and should return a tuple of three things `(path, args, kwargs)`:

- `path` should be the Python path to the class, with the class name included as the last part (for example, `myapp.custom_things.MyClass`). If your class is not available at the top level of a module it is not serializable.
- `args` should be a list of positional arguments to pass to your class' `__init__` method. Everything in this list should itself be serializable.

- `kwargs` should be a dict of keyword arguments to pass to your class' `__init__` method. Every value should itself be serializable.

: This return value is different from the `deconstruct()` method *for custom fields* which returns a tuple of four items.

Django will write out the value as an instantiation of your class with the given arguments, similar to the way it writes out references to Django fields.

To prevent a new migration from being created each time *makemigrations* is run, you should also add a `__eq__()` method to the decorated class. This function will be called by Django's migration framework to detect changes between states.

As long as all of the arguments to your class' constructor are themselves serializable, you can use the `@deconstructible` class decorator from `django.utils.deconstruct` to add the `deconstruct()` method:

```
from django.utils.deconstruct import deconstructible

@deconstructible
class MyCustomClass(object):

    def __init__(self, foo=1):
        self.foo = foo
        ...

    def __eq__(self, other):
        return self.foo == other.foo
```

The decorator adds logic to capture and preserve the arguments on their way into your constructor, and then returns those arguments exactly when deconstruct() is called.

3.7.12 Supporting Python 2 and 3

In order to generate migrations that support both Python 2 and 3, all string literals used in your models and fields (e.g. `verbose_name`, `related_name`, etc.), must be consistently either bytestrings or text (unicode) strings in both Python 2 and 3 (rather than bytes in Python 2 and text in Python 3, the default situation for unmarked string literals.) Otherwise running *makemigrations* under Python 3 will generate spurious new migrations to convert all these string attributes to text.

The easiest way to achieve this is to follow the advice in Django's Python 3 porting guide and make sure that all your modules begin with `from __future__ import unicode_literals`, so that all unmarked string literals are always unicode, regardless of Python version. When you add this to an app with existing migrations generated on Python 2, your next run of *makemigrations* on Python 3 will likely generate many changes as it converts all the bytestring attributes to text strings; this is normal and should only happen once.

3.7.13 Supporting multiple Django versions

If you are the maintainer of a third-party app with models, you may need to ship migrations that support multiple Django versions. In this case, you should always run *makemigrations* **with the lowest Django version you wish to support**.

The migrations system will maintain backwards-compatibility according to the same policy as the rest of Django, so migration files generated on Django X.Y should run unchanged on Django X.Y+1. The migrations system does

not promise forwards-compatibility, however. New features may be added, and migration files generated with newer versions of Django may not work on older versions.

:

The Migrations Operations Reference Covers the schema operations API, special operations, and writing your own operations.

The Writing Migrations "how-to" Explains how to structure and write database migrations for different scenarios you might encounter.

3.8 Managing files

This document describes Django's file access APIs for files such as those uploaded by a user. The lower level APIs are general enough that you could use them for other purposes. If you want to handle "static files" (JS, CSS, etc.), see Managing static files (e.g. images, JavaScript, CSS).

By default, Django stores files locally, using the *MEDIA_ROOT* and *MEDIA_URL* settings. The examples below assume that you're using these defaults.

However, Django provides ways to write custom *file storage systems* that allow you to completely customize where and how Django stores files. The second half of this document describes how these storage systems work.

3.8.1 Using files in models

When you use a *FileField* or *ImageField*, Django provides a set of APIs you can use to deal with that file.

Consider the following model, using an *ImageField* to store a photo:

```
from django.db import models

class Car(models.Model):
    name = models.CharField(max_length=255)
    price = models.DecimalField(max_digits=5, decimal_places=2)
    photo = models.ImageField(upload_to='cars')
```

Any `Car` instance will have a `photo` attribute that you can use to get at the details of the attached photo:

```
>>> car = Car.objects.get(name="57 Chevy")
>>> car.photo
<ImageFieldFile: chevy.jpg>
>>> car.photo.name
'cars/chevy.jpg'
>>> car.photo.path
'/media/cars/chevy.jpg'
>>> car.photo.url
'http://media.example.com/cars/chevy.jpg'
```

This object – `car.photo` in the example – is a `File` object, which means it has all the methods and attributes described below.

: The file is saved as part of saving the model in the database, so the actual file name used on disk cannot be relied on until after the model has been saved.

For example, you can change the file name by setting the file's *name* to a path relative to the file storage's location (*MEDIA_ROOT* if you are using the default *FileSystemStorage*):

```
>>> import os
>>> from django.conf import settings
>>> initial_path = car.photo.path
>>> car.photo.name = 'cars/chevy_ii.jpg'
>>> new_path = settings.MEDIA_ROOT + car.photo.name
>>> # Move the file on the filesystem
>>> os.rename(initial_path, new_path)
>>> car.save()
>>> car.photo.path
'/media/cars/chevy_ii.jpg'
>>> car.photo.path == new_path
True
```

3.8.2 The `File` object

Internally, Django uses a *`django.core.files.File`* instance any time it needs to represent a file.

Most of the time you'll simply use a `File` that Django's given you (i.e. a file attached to a model as above, or perhaps an uploaded file).

If you need to construct a `File` yourself, the easiest way is to create one using a Python built-in `file` object:

```
>>> from django.core.files import File

# Create a Python file object using open()
>>> f = open('/path/to/hello.world', 'w')
>>> myfile = File(f)
```

Now you can use any of the documented attributes and methods of the *`File`* class.

Be aware that files created in this way are not automatically closed. The following approach may be used to close files automatically:

```
>>> from django.core.files import File

# Create a Python file object using open() and the with statement
>>> with open('/path/to/hello.world', 'w') as f:
...     myfile = File(f)
...     myfile.write('Hello World')
...
>>> myfile.closed
True
>>> f.closed
True
```

Closing files is especially important when accessing file fields in a loop over a large number of objects. If files are not manually closed after accessing them, the risk of running out of file descriptors may arise. This may lead to the following error:

```
IOError: [Errno 24] Too many open files
```

3.8.3 File storage

Behind the scenes, Django delegates decisions about how and where to store files to a file storage system. This is the object that actually understands things like file systems, opening and reading files, etc.

Django's default file storage is given by the *DEFAULT_FILE_STORAGE* setting; if you don't explicitly provide a storage system, this is the one that will be used.

See below for details of the built-in default file storage system, and see Writing a custom storage system for information on writing your own file storage system.

Storage objects

Though most of the time you'll want to use a `File` object (which delegates to the proper storage for that file), you can use file storage systems directly. You can create an instance of some custom file storage class, or – often more useful – you can use the global default storage system:

```
>>> from django.core.files.storage import default_storage
>>> from django.core.files.base import ContentFile

>>> path = default_storage.save('/path/to/file', ContentFile('new content'))
>>> path
'/path/to/file'

>>> default_storage.size(path)
11
>>> default_storage.open(path).read()
'new content'

>>> default_storage.delete(path)
>>> default_storage.exists(path)
False
```

See File storage API for the file storage API.

The built-in filesystem storage class

Django ships with a *django.core.files.storage.FileSystemStorage* class which implements basic local filesystem file storage.

For example, the following code will store uploaded files under `/media/photos` regardless of what your *MEDIA_ROOT* setting is:

```
from django.db import models
from django.core.files.storage import FileSystemStorage

fs = FileSystemStorage(location='/media/photos')

class Car(models.Model):
    ...
    photo = models.ImageField(storage=fs)
```

Custom storage systems work the same way: you can pass them in as the `storage` argument to a *FileField*.

3.9 Testing in Django

Automated testing is an extremely useful bug-killing tool for the modern Web developer. You can use a collection of tests – a **test suite** – to solve, or avoid, a number of problems:

- When you're writing new code, you can use tests to validate your code works as expected.

- When you're refactoring or modifying old code, you can use tests to ensure your changes haven't affected your application's behavior unexpectedly.

Testing a Web application is a complex task, because a Web application is made of several layers of logic – from HTTP-level request handling, to form validation and processing, to template rendering. With Django's test-execution framework and assorted utilities, you can simulate requests, insert test data, inspect your application's output and generally verify your code is doing what it should be doing.

The best part is, it's really easy.

The preferred way to write tests in Django is using the `unittest` module built in to the Python standard library. This is covered in detail in the Writing and running tests document.

You can also use any *other* Python test framework; Django provides an API and tools for that kind of integration. They are described in the *Using different testing frameworks* section of Advanced testing topics.

3.9.1 Writing and running tests

:

The testing tutorial, the testing tools reference, and the advanced testing topics.

This document is split into two primary sections. First, we explain how to write tests with Django. Then, we explain how to run them.

Writing tests

Django's unit tests use a Python standard library module: `unittest`. This module defines tests using a class-based approach.

Here is an example which subclasses from *`django.test.TestCase`*, which is a subclass of `unittest.TestCase` that runs each test inside a transaction to provide isolation:

```
from django.test import TestCase
from myapp.models import Animal

class AnimalTestCase(TestCase):
    def setUp(self):
        Animal.objects.create(name="lion", sound="roar")
        Animal.objects.create(name="cat", sound="meow")

    def test_animals_can_speak(self):
        """Animals that can speak are correctly identified"""
        lion = Animal.objects.get(name="lion")
        cat = Animal.objects.get(name="cat")
        self.assertEqual(lion.speak(), 'The lion says "roar"')
        self.assertEqual(cat.speak(), 'The cat says "meow"')
```

When you *run your tests*, the default behavior of the test utility is to find all the test cases (that is, subclasses of `unittest.TestCase`) in any file whose name begins with `test`, automatically build a test suite out of those test cases, and run that suite.

For more details about `unittest`, see the Python documentation.

Where should the tests live?

The default *`startapp`* template creates a `tests.py` file in the new application. This might be fine if you only have a few tests, but as your test suite grows you'll likely want to restructure it into a tests package so you can split your

tests into different submodules such as `test_models.py`, `test_views.py`, `test_forms.py`, etc. Feel free to pick whatever organizational scheme you like.

See also *Using the Django test runner to test reusable applications*.

: If your tests rely on database access such as creating or querying models, be sure to create your test classes as subclasses of *`django.test.TestCase`* rather than `unittest.TestCase`.
Using `unittest.TestCase` avoids the cost of running each test in a transaction and flushing the database, but if your tests interact with the database their behavior will vary based on the order that the test runner executes them. This can lead to unit tests that pass when run in isolation but fail when run in a suite.

Running tests

Once you've written tests, run them using the *`test`* command of your project's `manage.py` utility:

```
$ ./manage.py test
```

Test discovery is based on the unittest module's built-in test discovery. By default, this will discover tests in any file named "test*.py" under the current working directory.

You can specify particular tests to run by supplying any number of "test labels" to `./manage.py test`. Each test label can be a full Python dotted path to a package, module, `TestCase` subclass, or test method. For instance:

```
# Run all the tests in the animals.tests module
$ ./manage.py test animals.tests

# Run all the tests found within the 'animals' package
$ ./manage.py test animals

# Run just one test case
$ ./manage.py test animals.tests.AnimalTestCase

# Run just one test method
$ ./manage.py test animals.tests.AnimalTestCase.test_animals_can_speak
```

You can also provide a path to a directory to discover tests below that directory:

```
$ ./manage.py test animals/
```

You can specify a custom filename pattern match using the `-p` (or `--pattern`) option, if your test files are named differently from the `test*.py` pattern:

```
$ ./manage.py test --pattern="tests_*.py"
```

If you press `Ctrl-C` while the tests are running, the test runner will wait for the currently running test to complete and then exit gracefully. During a graceful exit the test runner will output details of any test failures, report on how many tests were run and how many errors and failures were encountered, and destroy any test databases as usual. Thus pressing `Ctrl-C` can be very useful if you forget to pass the *`--failfast`* option, notice that some tests are unexpectedly failing and want to get details on the failures without waiting for the full test run to complete.

If you do not want to wait for the currently running test to finish, you can press `Ctrl-C` a second time and the test run will halt immediately, but not gracefully. No details of the tests run before the interruption will be reported, and any test databases created by the run will not be destroyed.

Test with warnings enabled

It's a good idea to run your tests with Python warnings enabled: `python -Wall manage.py test`. The `-Wall` flag tells Python to display deprecation warnings. Django, like many other Python libraries, uses these warnings to flag when features are going away. It also might flag areas in your code that aren't strictly wrong but could benefit from a better implementation.

The test database

Tests that require a database (namely, model tests) will not use your "real" (production) database. Separate, blank databases are created for the tests.

Regardless of whether the tests pass or fail, the test databases are destroyed when all the tests have been executed.

You can prevent the test databases from being destroyed by using the *test --keepdb* option. This will preserve the test database between runs. If the database does not exist, it will first be created. Any migrations will also be applied in order to keep it up to date.

The default test database names are created by prepending `test_` to the value of each *NAME* in *DATABASES*. When using SQLite, the tests will use an in-memory database by default (i.e., the database will be created in memory, bypassing the filesystem entirely!). The *TEST* dictionary in *DATABASES* offers a number of settings to configure your test database. For example, if you want to use a different database name, specify *NAME* in the *TEST* dictionary for any given database in *DATABASES*.

On PostgreSQL, *USER* will also need read access to the built-in `postgres` database.

Aside from using a separate database, the test runner will otherwise use all of the same database settings you have in your settings file: *ENGINE*, *USER*, *HOST*, etc. The test database is created by the user specified by *USER*, so you'll need to make sure that the given user account has sufficient privileges to create a new database on the system.

For fine-grained control over the character encoding of your test database, use the *CHARSET* TEST option. If you're using MySQL, you can also use the *COLLATION* option to control the particular collation used by the test database. See the settings documentation for details of these and other advanced settings.

If using an SQLite in-memory database with Python 3.4+ and SQLite 3.7.13+, shared cache will be enabled, so you can write tests with ability to share the database between threads.

Finding data from your production database when running tests?

If your code attempts to access the database when its modules are compiled, this will occur *before* the test database is set up, with potentially unexpected results. For example, if you have a database query in module-level code and a real database exists, production data could pollute your tests. *It is a bad idea to have such import-time database queries in your code* anyway - rewrite your code so that it doesn't do this.

This also applies to customized implementations of *ready()*.

:

The *advanced multi-db testing topics*.

Order in which tests are executed

In order to guarantee that all `TestCase` code starts with a clean database, the Django test runner reorders tests in the following way:

- All *TestCase* subclasses are run first.

- Then, all other Django-based tests (test cases based on *SimpleTestCase*, including *TransactionTestCase*) are run with no particular ordering guaranteed nor enforced among them.
- Then any other unittest.TestCase tests (including doctests) that may alter the database without restoring it to its original state are run.

: The new ordering of tests may reveal unexpected dependencies on test case ordering. This is the case with doctests that relied on state left in the database by a given *TransactionTestCase* test, they must be updated to be able to run independently.

You may reverse the execution order inside groups using the *test --reverse* option. This can help with ensuring your tests are independent from each other.

Rollback emulation

Any initial data loaded in migrations will only be available in TestCase tests and not in TransactionTestCase tests, and additionally only on backends where transactions are supported (the most important exception being MyISAM). This is also true for tests which rely on TransactionTestCase such as *LiveServerTestCase* and *StaticLiveServerTestCase*.

Django can reload that data for you on a per-testcase basis by setting the serialized_rollback option to True in the body of the TestCase or TransactionTestCase, but note that this will slow down that test suite by approximately 3x.

Third-party apps or those developing against MyISAM will need to set this; in general, however, you should be developing your own projects against a transactional database and be using TestCase for most tests, and thus not need this setting.

The initial serialization is usually very quick, but if you wish to exclude some apps from this process (and speed up test runs slightly), you may add those apps to *TEST_NON_SERIALIZED_APPS*.

To prevent serialized data from being loaded twice, setting serialized_rollback=True disables the *post_migrate* signal when flushing the test database.

Other test conditions

Regardless of the value of the *DEBUG* setting in your configuration file, all Django tests run with *DEBUG*=False. This is to ensure that the observed output of your code matches what will be seen in a production setting.

Caches are not cleared after each test, and running "manage.py test fooapp" can insert data from the tests into the cache of a live system if you run your tests in production because, unlike databases, a separate "test cache" is not used. This behavior may change in the future.

Understanding the test output

When you run your tests, you'll see a number of messages as the test runner prepares itself. You can control the level of detail of these messages with the verbosity option on the command line:

```
Creating test database...
Creating table myapp_animal
Creating table myapp_mineral
```

This tells you that the test runner is creating a test database, as described in the previous section.

Once the test database has been created, Django will run your tests. If everything goes well, you'll see something like this:

```
----------------------------------------------------------------------
Ran 22 tests in 0.221s

OK
```

If there are test failures, however, you'll see full details about which tests failed:

```
======================================================================
FAIL: test_was_published_recently_with_future_poll (polls.tests.PollMethodTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/dev/mysite/polls/tests.py", line 16, in test_was_published_recently_with_future_poll
    self.assertIs(future_poll.was_published_recently(), False)
AssertionError: True is not False

----------------------------------------------------------------------
Ran 1 test in 0.003s

FAILED (failures=1)
```

A full explanation of this error output is beyond the scope of this document, but it's pretty intuitive. You can consult the documentation of Python's `unittest` library for details.

Note that the return code for the test-runner script is 1 for any number of failed and erroneous tests. If all the tests pass, the return code is 0. This feature is useful if you're using the test-runner script in a shell script and need to test for success or failure at that level.

Speeding up the tests

Running tests in parallel As long as your tests are properly isolated, you can run them in parallel to gain a speed up on multi-core hardware. See *`test --parallel`*.

Password hashing The default password hasher is rather slow by design. If you're authenticating many users in your tests, you may want to use a custom settings file and set the *`PASSWORD_HASHERS`* setting to a faster hashing algorithm:

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.MD5PasswordHasher',
]
```

Don't forget to also include in *`PASSWORD_HASHERS`* any hashing algorithm used in fixtures, if any.

3.9.2 Testing tools

Django provides a small set of tools that come in handy when writing tests.

The test client

The test client is a Python class that acts as a dummy Web browser, allowing you to test your views and interact with your Django-powered application programmatically.

Some of the things you can do with the test client are:

- Simulate GET and POST requests on a URL and observe the response – everything from low-level HTTP (result headers and status codes) to page content.
- See the chain of redirects (if any) and check the URL and status code at each step.
- Test that a given request is rendered by a given Django template, with a template context that contains certain values.

Note that the test client is not intended to be a replacement for Selenium or other "in-browser" frameworks. Django's test client has a different focus. In short:

- Use Django's test client to establish that the correct template is being rendered and that the template is passed the correct context data.
- Use in-browser frameworks like Selenium to test *rendered* HTML and the *behavior* of Web pages, namely JavaScript functionality. Django also provides special support for those frameworks; see the section on `LiveServerTestCase` for more details.

A comprehensive test suite should use a combination of both test types.

Overview and a quick example

To use the test client, instantiate `django.test.Client` and retrieve Web pages:

```
>>> from django.test import Client
>>> c = Client()
>>> response = c.post('/login/', {'username': 'john', 'password': 'smith'})
>>> response.status_code
200
>>> response = c.get('/customer/details/')
>>> response.content
b'<!DOCTYPE html...'
```

As this example suggests, you can instantiate `Client` from within a session of the Python interactive interpreter.

Note a few important things about how the test client works:

- The test client does *not* require the Web server to be running. In fact, it will run just fine with no Web server running at all! That's because it avoids the overhead of HTTP and deals directly with the Django framework. This helps make the unit tests run quickly.
- When retrieving pages, remember to specify the *path* of the URL, not the whole domain. For example, this is correct:

```
>>> c.get('/login/')
```

  This is incorrect:

```
>>> c.get('https://www.example.com/login/')
```

  The test client is not capable of retrieving Web pages that are not powered by your Django project. If you need to retrieve other Web pages, use a Python standard library module such as `urllib`.
- To resolve URLs, the test client uses whatever URLconf is pointed-to by your `ROOT_URLCONF` setting.
- Although the above example would work in the Python interactive interpreter, some of the test client's functionality, notably the template-related functionality, is only available *while tests are running*.

The reason for this is that Django's test runner performs a bit of black magic in order to determine which template was loaded by a given view. This black magic (essentially a patching of Django's template system in memory) only happens during test running.

- By default, the test client will disable any CSRF checks performed by your site.

  If, for some reason, you *want* the test client to perform CSRF checks, you can create an instance of the test client that enforces CSRF checks. To do this, pass in the `enforce_csrf_checks` argument when you construct your client:

```
>>> from django.test import Client
>>> csrf_client = Client(enforce_csrf_checks=True)
```

Making requests

Use the `django.test.Client` class to make requests.

class **`Client`**(*enforce_csrf_checks=False*, ***defaults*)

It requires no arguments at time of construction. However, you can use keywords arguments to specify some default headers. For example, this will send a `User-Agent` HTTP header in each request:

```
>>> c = Client(HTTP_USER_AGENT='Mozilla/5.0')
```

The values from the `extra` keywords arguments passed to *`get()`*, *`post()`*, etc. have precedence over the defaults passed to the class constructor.

The `enforce_csrf_checks` argument can be used to test CSRF protection (see above).

Once you have a `Client` instance, you can call any of the following methods:

`get`(*path*, *data=None*, *follow=False*, *secure=False*, ***extra*)

Makes a GET request on the provided `path` and returns a `Response` object, which is documented below.

The key-value pairs in the `data` dictionary are used to create a GET data payload. For example:

```
>>> c = Client()
>>> c.get('/customers/details/', {'name': 'fred', 'age': 7})
```

...will result in the evaluation of a GET request equivalent to:

```
/customers/details/?name=fred&age=7
```

The `extra` keyword arguments parameter can be used to specify headers to be sent in the request. For example:

```
>>> c = Client()
>>> c.get('/customers/details/', {'name': 'fred', 'age': 7},
...       HTTP_X_REQUESTED_WITH='XMLHttpRequest')
```

...will send the HTTP header `HTTP_X_REQUESTED_WITH` to the details view, which is a good way to test code paths that use the *`django.http.HttpRequest.is_ajax()`* method.

CGI specification

The headers sent via `**extra` should follow CGI specification. For example, emulating a different "Host" header as sent in the HTTP request from the browser to the server should be passed as `HTTP_HOST`.

If you already have the GET arguments in URL-encoded form, you can use that encoding instead of using the data argument. For example, the previous GET request could also be posed as:

```
>>> c = Client()
>>> c.get('/customers/details/?name=fred&age=7')
```

If you provide a URL with both an encoded GET data and a data argument, the data argument will take precedence.

If you set `follow` to `True` the client will follow any redirects and a `redirect_chain` attribute will be set in the response object containing tuples of the intermediate urls and status codes.

If you had a URL `/redirect_me/` that redirected to `/next/`, that redirected to `/final/`, this is what you'd see:

```
>>> response = c.get('/redirect_me/', follow=True)
>>> response.redirect_chain
[('http://testserver/next/', 302), ('http://testserver/final/', 302)]
```

If you set `secure` to `True` the client will emulate an HTTPS request.

post(*path*, *data=None*, *content_type=MULTIPART_CONTENT*, *follow=False*, *secure=False*, ***extra*)

Makes a POST request on the provided `path` and returns a `Response` object, which is documented below.

The key-value pairs in the `data` dictionary are used to submit POST data. For example:

```
>>> c = Client()
>>> c.post('/login/', {'name': 'fred', 'passwd': 'secret'})
```

...will result in the evaluation of a POST request to this URL:

```
/login/
```

...with this POST data:

```
name=fred&passwd=secret
```

If you provide `content_type` (e.g. *`text/xml`* for an XML payload), the contents of `data` will be sent as-is in the POST request, using `content_type` in the HTTP `Content-Type` header.

If you don't provide a value for `content_type`, the values in `data` will be transmitted with a content type of *`multipart/form-data`*. In this case, the key-value pairs in `data` will be encoded as a multipart message and used to create the POST data payload.

To submit multiple values for a given key – for example, to specify the selections for a `<select multiple>` – provide the values as a list or tuple for the required key. For example, this value of `data` would submit three selected values for the field named `choices`:

```
{'choices': ('a', 'b', 'd')}
```

Submitting files is a special case. To POST a file, you need only provide the file field name as a key, and a file handle to the file you wish to upload as a value. For example:

```
>>> c = Client()
>>> with open('wishlist.doc') as fp:
...     c.post('/customers/wishes/', {'name': 'fred', 'attachment': fp})
```

(The name `attachment` here is not relevant; use whatever name your file-processing code expects.)

You may also provide any file-like object (e.g., `StringIO` or `BytesIO`) as a file handle.

Note that if you wish to use the same file handle for multiple `post()` calls then you will need to manually reset the file pointer between posts. The easiest way to do this is to manually close the file after it has been provided to `post()`, as demonstrated above.

You should also ensure that the file is opened in a way that allows the data to be read. If your file contains binary data such as an image, this means you will need to open the file in `rb` (read binary) mode.

The `extra` argument acts the same as for *`Client.get()`*.

If the URL you request with a POST contains encoded parameters, these parameters will be made available in the request.GET data. For example, if you were to make the request:

```
>>> c.post('/login/?visitor=true', {'name': 'fred', 'passwd': 'secret'})
```

... the view handling this request could interrogate request.POST to retrieve the username and password, and could interrogate request.GET to determine if the user was a visitor.

If you set `follow` to `True` the client will follow any redirects and a `redirect_chain` attribute will be set in the response object containing tuples of the intermediate urls and status codes.

If you set `secure` to `True` the client will emulate an HTTPS request.

head(*path*, *data=None*, *follow=False*, *secure=False*, ***extra*)
Makes a HEAD request on the provided `path` and returns a `Response` object. This method works just like *`Client.get()`*, including the `follow`, `secure` and `extra` arguments, except it does not return a message body.

options(*path*, *data=''*, *content_type='application/octet-stream'*, *follow=False*, *secure=False*, ***extra*)
Makes an OPTIONS request on the provided `path` and returns a `Response` object. Useful for testing RESTful interfaces.

When `data` is provided, it is used as the request body, and a `Content-Type` header is set to `content_type`.

The `follow`, `secure` and `extra` arguments act the same as for *`Client.get()`*.

put(*path*, *data=''*, *content_type='application/octet-stream'*, *follow=False*, *secure=False*, ***extra*)
Makes a PUT request on the provided `path` and returns a `Response` object. Useful for testing RESTful interfaces.

When `data` is provided, it is used as the request body, and a `Content-Type` header is set to `content_type`.

The `follow`, `secure` and `extra` arguments act the same as for *`Client.get()`*.

patch(*path*, *data=''*, *content_type='application/octet-stream'*, *follow=False*, *secure=False*, ***extra*)
Makes a PATCH request on the provided `path` and returns a `Response` object. Useful for testing RESTful interfaces.

The `follow`, `secure` and `extra` arguments act the same as for *`Client.get()`*.

delete(*path*, *data=''*, *content_type='application/octet-stream'*, *follow=False*, *secure=False*, ***extra*)
Makes a DELETE request on the provided `path` and returns a `Response` object. Useful for testing RESTful interfaces.

When `data` is provided, it is used as the request body, and a `Content-Type` header is set to `content_type`.

The `follow`, `secure` and `extra` arguments act the same as for *`Client.get()`*.

trace(*path*, *follow=False*, *secure=False*, ***extra*)
Makes a TRACE request on the provided `path` and returns a `Response` object. Useful for simulating diagnostic probes.

Unlike the other request methods, `data` is not provided as a keyword parameter in order to comply with **RFC 7231#section-4.3.8**, which mandates that TRACE requests must not have a body.

The `follow`, `secure`, and `extra` arguments act the same as for *`Client.get()`*.

`login`(***credentials*)

If your site uses Django's authentication system and you deal with logging in users, you can use the test client's `login()` method to simulate the effect of a user logging into the site.

After you call this method, the test client will have all the cookies and session data required to pass any login-based tests that may form part of a view.

The format of the `credentials` argument depends on which *authentication backend* you're using (which is configured by your *`AUTHENTICATION_BACKENDS`* setting). If you're using the standard authentication backend provided by Django (`ModelBackend`), `credentials` should be the user's username and password, provided as keyword arguments:

```
>>> c = Client()
>>> c.login(username='fred', password='secret')

# Now you can access a view that's only available to logged-in users.
```

If you're using a different authentication backend, this method may require different credentials. It requires whichever credentials are required by your backend's `authenticate()` method.

`login()` returns `True` if it the credentials were accepted and login was successful.

Finally, you'll need to remember to create user accounts before you can use this method. As we explained above, the test runner is executed using a test database, which contains no users by default. As a result, user accounts that are valid on your production site will not work under test conditions. You'll need to create users as part of the test suite – either manually (using the Django model API) or with a test fixture. Remember that if you want your test user to have a password, you can't set the user's password by setting the password attribute directly – you must use the *`set_password()`* function to store a correctly hashed password. Alternatively, you can use the *`create_user()`* helper method to create a new user with a correctly hashed password.

In previous versions, inactive users (*`is_active=False`*) were not permitted to login.

`force_login`(*user*, *backend=None*)

If your site uses Django's authentication system, you can use the `force_login()` method to simulate the effect of a user logging into the site. Use this method instead of *`login()`* when a test requires a user be logged in and the details of how a user logged in aren't important.

Unlike `login()`, this method skips the authentication and verification steps: inactive users (*`is_active=False`*) are permitted to login and the user's credentials don't need to be provided.

The user will have its `backend` attribute set to the value of the `backend` argument (which should be a dotted Python path string), or to `settings.AUTHENTICATION_BACKENDS[0]` if a value isn't provided. The *`authenticate()`* function called by *`login()`* normally annotates the user like this.

This method is faster than `login()` since the expensive password hashing algorithms are bypassed. Also, you can speed up `login()` by *using a weaker hasher while testing*.

`logout`()

If your site uses Django's authentication system, the `logout()` method can be used to simulate the effect of a user logging out of your site.

After you call this method, the test client will have all the cookies and session data cleared to defaults. Subsequent requests will appear to come from an *`AnonymousUser`*.

Testing responses

The `get()` and `post()` methods both return a `Response` object. This `Response` object is *not* the same as the `HttpResponse` object returned by Django views; the test response object has some additional data useful for test code to verify.

Specifically, a `Response` object has the following attributes:

class Response

client
The test client that was used to make the request that resulted in the response.

content
The body of the response, as a bytestring. This is the final page content as rendered by the view, or any error message.

context
The template `Context` instance that was used to render the template that produced the response content.

If the rendered page used multiple templates, then `context` will be a list of `Context` objects, in the order in which they were rendered.

Regardless of the number of templates used during rendering, you can retrieve context values using the `[]` operator. For example, the context variable `name` could be retrieved using:

```
>>> response = client.get('/foo/')
>>> response.context['name']
'Arthur'
```

Not using Django templates?

This attribute is only populated when using the *`DjangoTemplates`* backend. If you're using another template engine, *`context_data`* may be a suitable alternative on responses with that attribute.

---

json(***kwargs*)
The body of the response, parsed as JSON. Extra keyword arguments are passed to `json.loads()`. For example:

```
>>> response = client.get('/foo/')
>>> response.json()['name']
'Arthur'
```

If the `Content-Type` header is not `"application/json"`, then a `ValueError` will be raised when trying to parse the response.

request
The request data that stimulated the response.

wsgi_request
The `WSGIRequest` instance generated by the test handler that generated the response.

status_code
The HTTP status of the response, as an integer. For a full list of defined codes, see the IANA status code registry.

templates
A list of `Template` instances used to render the final content, in the order they were rendered. For each

template in the list, use `template.name` to get the template's file name, if the template was loaded from a file. (The name is a string such as `'admin/index.html'`.)

Not using Django templates?

This attribute is only populated when using the *DjangoTemplates* backend. If you're using another template engine, *template_name* may be a suitable alternative if you only need the name of the template used for rendering.

---

resolver_match

An instance of *ResolverMatch* for the response. You can use the *func* attribute, for example, to verify the view that served the response:

```
# my_view here is a function based view
self.assertEqual(response.resolver_match.func, my_view)

# class-based views need to be compared by name, as the functions
# generated by as_view() won't be equal
self.assertEqual(response.resolver_match.func.__name__, MyView.as_view().__name__)
```

If the given URL is not found, accessing this attribute will raise a *Resolver404* exception.

You can also use dictionary syntax on the response object to query the value of any settings in the HTTP headers. For example, you could determine the content type of a response using `response['Content-Type']`.

Exceptions

If you point the test client at a view that raises an exception, that exception will be visible in the test case. You can then use a standard `try ... except` block or `assertRaises()` to test for exceptions.

The only exceptions that are not visible to the test client are *Http404*, *PermissionDenied*, `SystemExit`, and *SuspiciousOperation*. Django catches these exceptions internally and converts them into the appropriate HTTP response codes. In these cases, you can check `response.status_code` in your test.

Persistent state

The test client is stateful. If a response returns a cookie, then that cookie will be stored in the test client and sent with all subsequent `get()` and `post()` requests.

Expiration policies for these cookies are not followed. If you want a cookie to expire, either delete it manually or create a new `Client` instance (which will effectively delete all cookies).

A test client has two attributes that store persistent state information. You can access these properties as part of a test condition.

`Client.`**cookies**

A Python `SimpleCookie` object, containing the current values of all the client cookies. See the documentation of the `http.cookies` module for more.

`Client.`**session**

A dictionary-like object containing session information. See the session documentation for full details.

To modify the session and then save it, it must be stored in a variable first (because a new `SessionStore` is created every time this property is accessed):

```
    def test_something(self):
        session = self.client.session
        session['somekey'] = 'test'
        session.save()
```

Setting the language

When testing applications that support internationalization and localization, you might want to set the language for a test client request. The method for doing so depends on whether or not the *LocaleMiddleware* is enabled.

If the middleware is enabled, the language can be set by creating a cookie with a name of *LANGUAGE_COOKIE_NAME* and a value of the language code:

```
from django.conf import settings

def test_language_using_cookie(self):
    self.client.cookies.load({settings.LANGUAGE_COOKIE_NAME: 'fr'})
    response = self.client.get('/')
    self.assertEqual(response.content, b"Bienvenue sur mon site.")
```

or by including the `Accept-Language` HTTP header in the request:

```
def test_language_using_header(self):
    response = self.client.get('/', HTTP_ACCEPT_LANGUAGE='fr')
    self.assertEqual(response.content, b"Bienvenue sur mon site.")
```

More details are in *How Django discovers language preference*.

If the middleware isn't enabled, the active language may be set using *translation.override()*:

```
from django.utils import translation

def test_language_using_override(self):
    with translation.override('fr'):
        response = self.client.get('/')
    self.assertEqual(response.content, b"Bienvenue sur mon site.")
```

More details are in *Explicitly setting the active language*.

Example

The following is a simple unit test using the test client:

```
import unittest
from django.test import Client

class SimpleTest(unittest.TestCase):
    def setUp(self):
        # Every test needs a client.
        self.client = Client()

    def test_details(self):
        # Issue a GET request.
        response = self.client.get('/customer/details/')

        # Check that the response is 200 OK.
        self.assertEqual(response.status_code, 200)
```

```
        # Check that the rendered context contains 5 customers.
        self.assertEqual(len(response.context['customers']), 5)
```

:

django.test.RequestFactory

Provided test case classes

Normal Python unit test classes extend a base class of `unittest.TestCase`. Django provides a few extensions of this base class:

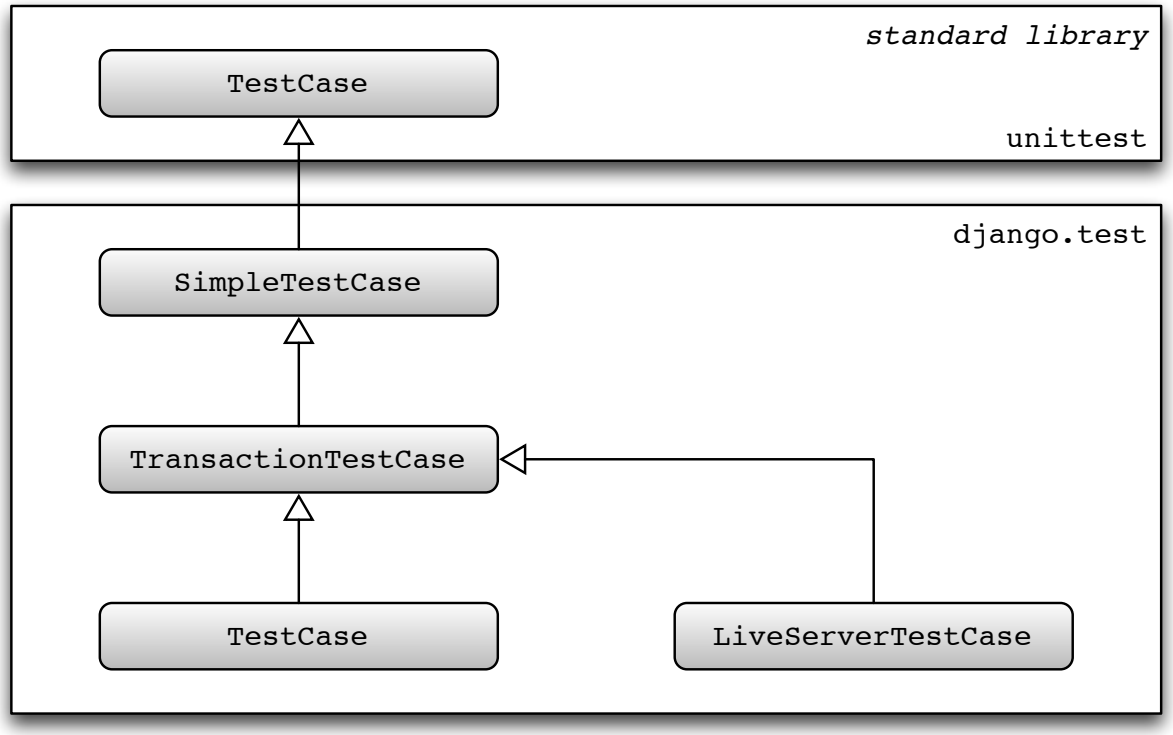


Figure 3.1: Hierarchy of Django unit testing classes

Converting a normal `unittest.TestCase` to any of the subclasses is easy: change the base class of your test from `unittest.TestCase` to the subclass. All of the standard Python unit test functionality will be available, and it will be augmented with some useful additions as described in each section below.

SimpleTestCase

class SimpleTestCase

A subclass of `unittest.TestCase` that adds this functionality:

- Some useful assertions like:
  - Checking that a callable *raises a certain exception*.

  - Testing form field *rendering and error treatment*.
  - Testing *HTML responses for the presence/lack of a given fragment*.
  - Verifying that a template *has/hasn't been used to generate a given response content*.
  - Verifying a HTTP *redirect* is performed by the app.
  - Robustly testing two *HTML fragments* for equality/inequality or *containment*.
  - Robustly testing two *XML fragments* for equality/inequality.
  - Robustly testing two *JSON fragments* for equality.
- The ability to run tests with *modified settings*.
- Using the *client Client*.

If your tests make any database queries, use subclasses *TransactionTestCase* or *TestCase*.

`SimpleTestCase.`**`allow_database_queries`**

SimpleTestCase disallows database queries by default. This helps to avoid executing write queries which will affect other tests since each `SimpleTestCase` test isn't run in a transaction. If you aren't concerned about this problem, you can disable this behavior by setting the `allow_database_queries` class attribute to `True` on your test class.

: `SimpleTestCase` and its subclasses (e.g. `TestCase`, ...) rely on `setUpClass()` and `tearDownClass()` to perform some class-wide initialization (e.g. overriding settings). If you need to override those methods, don't forget to call the `super` implementation:

```
class MyTestCase(TestCase):

    @classmethod
    def setUpClass(cls):
        super(MyTestCase, cls).setUpClass()
        ...

    @classmethod
    def tearDownClass(cls):
        ...
        super(MyTestCase, cls).tearDownClass()
```

Be sure to account for Python's behavior if an exception is raised during `setUpClass()`. If that happens, neither the tests in the class nor `tearDownClass()` are run. In the case of *django.test.TestCase*, this will leak the transaction created in `super()` which results in various symptoms including a segmentation fault on some platforms (reported on OS X). If you want to intentionally raise an exception such as `unittest.SkipTest` in `setUpClass()`, be sure to do it before calling `super()` to avoid this.

TransactionTestCase

class TransactionTestCase

`TransactionTestCase` inherits from *SimpleTestCase* to add some database-specific features:

- Resetting the database to a known state at the beginning of each test to ease testing and using the ORM.
- Database *fixtures*.
- Test *skipping based on database backend features*.

- The remaining specialized *assert** methods.

Django's *TestCase* class is a more commonly used subclass of TransactionTestCase that makes use of database transaction facilities to speed up the process of resetting the database to a known state at the beginning of each test. A consequence of this, however, is that some database behaviors cannot be tested within a Django TestCase class. For instance, you cannot test that a block of code is executing within a transaction, as is required when using *select_for_update()*. In those cases, you should use TransactionTestCase.

TransactionTestCase and TestCase are identical except for the manner in which the database is reset to a known state and the ability for test code to test the effects of commit and rollback:

- A TransactionTestCase resets the database after the test runs by truncating all tables. A TransactionTestCase may call commit and rollback and observe the effects of these calls on the database.
- A TestCase, on the other hand, does not truncate tables after a test. Instead, it encloses the test code in a database transaction that is rolled back at the end of the test. This guarantees that the rollback at the end of the test restores the database to its initial state.

: TestCase running on a database that does not support rollback (e.g. MySQL with the MyISAM storage engine), and all instances of TransactionTestCase, will roll back at the end of the test by deleting all data from the test database.
Apps *will not see their data reloaded*; if you need this functionality (for example, third-party apps should enable this) you can set serialized_rollback = True inside the TestCase body.

TestCase

class TestCase

This is the most common class to use for writing tests in Django. It inherits from *TransactionTestCase* (and by extension *SimpleTestCase*). If your Django application doesn't use a database, use *SimpleTestCase*.

The class:

- Wraps the tests within two nested *atomic()* blocks: one for the whole class and one for each test. Therefore, if you want to test some specific database transaction behavior, use *TransactionTestCase*.
- Checks deferrable database constraints at the end of each test.

The check for deferrable database constraints at the end of each test was added.

It also provides an additional method:

classmethod TestCase.**setUpTestData**()

The class-level atomic block described above allows the creation of initial data at the class level, once for the whole TestCase. This technique allows for faster tests as compared to using setUp().

For example:

```
from django.test import TestCase

class MyTests(TestCase):
    @classmethod
    def setUpTestData(cls):
        # Set up data for the whole TestCase
        cls.foo = Foo.objects.create(bar="Test")
        ...

    def test1(self):
        # Some test using self.foo
        ...
```

```
    def test2(self):
        # Some other test using self.foo
        ...
```

Note that if the tests are run on a database with no transaction support (for instance, MySQL with the MyISAM engine), `setUpTestData()` will be called before each test, negating the speed benefits.

Be careful not to modify any objects created in `setUpTestData()` in your test methods. Modifications to in-memory objects from setup work done at the class level will persist between test methods. If you do need to modify them, you could reload them in the `setUp()` method with *`refresh_from_db()`*, for example.

LiveServerTestCase

class LiveServerTestCase

`LiveServerTestCase` does basically the same as *`TransactionTestCase`* with one extra feature: it launches a live Django server in the background on setup, and shuts it down on teardown. This allows the use of automated test clients other than the *Django dummy client* such as, for example, the Selenium client, to execute a series of functional tests inside a browser and simulate a real user's actions.

By default the live server listens on `localhost` and picks the first available port in the `8081-8179` range. Its full URL can be accessed with `self.live_server_url` during the tests.

In earlier versions, the live server's default address was always `'localhost:8081'`.

If you'd like to select another address, you may pass a different one using the *`test --liveserver`* option, for example:

```
$ ./manage.py test --liveserver=localhost:8082
```

In older versions `live_server_url` could only be accessed from an instance. It now is a class property and can be accessed from class methods like `setUpClass()`.

Another way of changing the default server address is by setting the *DJANGO_LIVE_TEST_SERVER_ADDRESS* environment variable somewhere in your code (for example, in a *custom test runner*):

```
import os
os.environ['DJANGO_LIVE_TEST_SERVER_ADDRESS'] = 'localhost:8082'
```

In the case where the tests are run by multiple processes in parallel (for example, in the context of several simultaneous continuous integration builds), the processes will compete for the same address, and therefore your tests might randomly fail with an "Address already in use" error. To avoid this problem, you can pass a comma-separated list of ports or ranges of ports (at least as many as the number of potential parallel processes). For example:

```
$ ./manage.py test --liveserver=localhost:8082,8090-8100,9000-9200,7041
```

Then, during test execution, each new live test server will try every specified port until it finds one that is free and takes it.

To demonstrate how to use `LiveServerTestCase`, let's write a simple Selenium test. First of all, you need to install the selenium package into your Python path:

```
$ pip install selenium
```

Then, add a `LiveServerTestCase`-based test to your app's tests module (for example: `myapp/tests.py`). For this example, we'll assume you're using the *`staticfiles`* app and want to have static files served during the execution of your tests similar to what we get at development time with `DEBUG=True`, i.e. without having to collect them using *`collectstatic`*. We'll use the *`StaticLiveServerTestCase`* subclass which provides that functionality. Replace it with `django.test.LiveServerTestCase` if you don't need that.

The code for this test may look as follows:

```
from django.contrib.staticfiles.testing import StaticLiveServerTestCase
from selenium.webdriver.firefox.webdriver import WebDriver

class MySeleniumTests(StaticLiveServerTestCase):
    fixtures = ['user-data.json']

    @classmethod
    def setUpClass(cls):
        super(MySeleniumTests, cls).setUpClass()
        cls.selenium = WebDriver()
        cls.selenium.implicitly_wait(10)

    @classmethod
    def tearDownClass(cls):
        cls.selenium.quit()
        super(MySeleniumTests, cls).tearDownClass()

    def test_login(self):
        self.selenium.get('%s%s' % (self.live_server_url, '/login/'))
        username_input = self.selenium.find_element_by_name("username")
        username_input.send_keys('myuser')
        password_input = self.selenium.find_element_by_name("password")
        password_input.send_keys('secret')
        self.selenium.find_element_by_xpath('//input[@value="Log in"]').click()
```

Finally, you may run the test as follows:

```
$ ./manage.py test myapp.tests.MySeleniumTests.test_login
```

This example will automatically open Firefox then go to the login page, enter the credentials and press the "Log in" button. Selenium offers other drivers in case you do not have Firefox installed or wish to use another browser. The example above is just a tiny fraction of what the Selenium client can do; check out the full reference for more details.

: When using an in-memory SQLite database to run the tests, the same database connection will be shared by two threads in parallel: the thread in which the live server is run and the thread in which the test case is run. It's important to prevent simultaneous database queries via this shared connection by the two threads, as that may sometimes randomly cause the tests to fail. So you need to ensure that the two threads don't access the database at the same time. In particular, this means that in some cases (for example, just after clicking a link or submitting a form), you might need to check that a response is received by Selenium and that the next page is loaded before proceeding with further test execution. Do this, for example, by making Selenium wait until the `<body>` HTML tag is found in the response (requires Selenium > 2.13):

```
def test_login(self):
    from selenium.webdriver.support.wait import WebDriverWait
    timeout = 2
    ...
    self.selenium.find_element_by_xpath('//input[@value="Log in"]').click()
    # Wait until the response is received
    WebDriverWait(self.selenium, timeout).until(
        lambda driver: driver.find_element_by_tag_name('body'))
```

The tricky thing here is that there's really no such thing as a "page load," especially in modern Web apps that generate HTML dynamically after the server generates the initial document. So, simply checking for the presence of `<body>` in the response might not necessarily be appropriate for all use cases. Please refer to the Selenium FAQ and Selenium documentation for more information.

Test cases features

Default test client

`SimpleTestCase.client`

Every test case in a `django.test.*TestCase` instance has access to an instance of a Django test client. This client can be accessed as `self.client`. This client is recreated for each test, so you don't have to worry about state (such as cookies) carrying over from one test to another.

This means, instead of instantiating a `Client` in each test:

```
import unittest
from django.test import Client

class SimpleTest(unittest.TestCase):
    def test_details(self):
        client = Client()
        response = client.get('/customer/details/')
        self.assertEqual(response.status_code, 200)

    def test_index(self):
        client = Client()
        response = client.get('/customer/index/')
        self.assertEqual(response.status_code, 200)
```

...you can just refer to `self.client`, like so:

```
from django.test import TestCase

class SimpleTest(TestCase):
    def test_details(self):
        response = self.client.get('/customer/details/')
        self.assertEqual(response.status_code, 200)

    def test_index(self):
        response = self.client.get('/customer/index/')
        self.assertEqual(response.status_code, 200)
```

Customizing the test client

`SimpleTestCase.client_class`

If you want to use a different `Client` class (for example, a subclass with customized behavior), use the *`client_class`* class attribute:

```
from django.test import TestCase, Client

class MyTestClient(Client):
    # Specialized methods for your environment
    ...

class MyTest(TestCase):
    client_class = MyTestClient

    def test_my_stuff(self):
        # Here self.client is an instance of MyTestClient...
        call_some_test_code()
```

Fixture loading

`TransactionTestCase.`**`fixtures`**

A test case for a database-backed website isn't much use if there isn't any data in the database. Tests are more readable and it's more maintainable to create objects using the ORM, for example in *`TestCase.setUpTestData()`*, however, you can also use fixtures.

A fixture is a collection of data that Django knows how to import into a database. For example, if your site has user accounts, you might set up a fixture of fake user accounts in order to populate your database during tests.

The most straightforward way of creating a fixture is to use the *`manage.py dumpdata`* command. This assumes you already have some data in your database. See the *`dumpdata documentation`* for more details.

Once you've created a fixture and placed it in a `fixtures` directory in one of your *`INSTALLED_APPS`*, you can use it in your unit tests by specifying a `fixtures` class attribute on your *`django.test.TestCase`* subclass:

```
from django.test import TestCase
from myapp.models import Animal

class AnimalTestCase(TestCase):
    fixtures = ['mammals.json', 'birds']

    def setUp(self):
        # Test definitions as before.
        call_setup_methods()

    def testFluffyAnimals(self):
        # A test that uses the fixtures.
        call_some_test_code()
```

Here's specifically what will happen:

- At the start of each test, before `setUp()` is run, Django will flush the database, returning the database to the state it was in directly after *`migrate`* was called.
- Then, all the named fixtures are installed. In this example, Django will install any JSON fixture named `mammals`, followed by any fixture named `birds`. See the *`loaddata`* documentation for more details on defining and installing fixtures.

For performance reasons, *`TestCase`* loads fixtures once for the entire test class, before *`setUpTestData()`*, instead of before each test, and it uses transactions to clean the database before each test. In any case, you can be certain that the outcome of a test will not be affected by another test or by the order of test execution.

By default, fixtures are only loaded into the `default` database. If you are using multiple databases and set *`multi_db=True`*, fixtures will be loaded into all databases.

URLconf configuration

If your application provides views, you may want to include tests that use the test client to exercise those views. However, an end user is free to deploy the views in your application at any URL of their choosing. This means that your tests can't rely upon the fact that your views will be available at a particular URL. Decorate your test class or test method with `@override_settings(ROOT_URLCONF=...)` for URLconf configuration.

Multi-database support

`TransactionTestCase.`**`multi_db`**

Django sets up a test database corresponding to every database that is defined in the *DATABASES* definition in your settings file. However, a big part of the time taken to run a Django TestCase is consumed by the call to `flush` that ensures that you have a clean database at the start of each test run. If you have multiple databases, multiple flushes are required (one for each database), which can be a time consuming activity – especially if your tests don't need to test multi-database activity.

As an optimization, Django only flushes the `default` database at the start of each test run. If your setup contains multiple databases, and you have a test that requires every database to be clean, you can use the `multi_db` attribute on the test suite to request a full flush.

For example:

```
class TestMyViews(TestCase):
    multi_db = True

    def test_index_page_view(self):
        call_some_test_code()
```

This test case will flush *all* the test databases before running `test_index_page_view`.

The `multi_db` flag also affects into which databases the attr:*TransactionTestCase.fixtures* are loaded. By default (when `multi_db=False`), fixtures are only loaded into the `default` database. If `multi_db=True`, fixtures are loaded into all databases.

Overriding settings

: Use the functions below to temporarily alter the value of settings in tests. Don't manipulate `django.conf.settings` directly as Django won't restore the original values after such manipulations.

`SimpleTestCase.`**`settings`**`()`

For testing purposes it's often useful to change a setting temporarily and revert to the original value after running the testing code. For this use case Django provides a standard Python context manager (see **PEP 343**) called *settings()*, which can be used like this:

```
from django.test import TestCase

class LoginTestCase(TestCase):

    def test_login(self):

        # First check for the default behavior
        response = self.client.get('/sekrit/')
        self.assertRedirects(response, '/accounts/login/?next=/sekrit/')

        # Then override the LOGIN_URL setting
        with self.settings(LOGIN_URL='/other/login/'):
            response = self.client.get('/sekrit/')
            self.assertRedirects(response, '/other/login/?next=/sekrit/')
```

This example will override the *LOGIN_URL* setting for the code in the `with` block and reset its value to the previous state afterwards.

`SimpleTestCase.`**`modify_settings`**`()`

It can prove unwieldy to redefine settings that contain a list of values. In practice, adding or removing values is often sufficient. The *modify_settings()* context manager makes it easy:

```
from django.test import TestCase

class MiddlewareTestCase(TestCase):

    def test_cache_middleware(self):
        with self.modify_settings(MIDDLEWARE={
            'append': 'django.middleware.cache.FetchFromCacheMiddleware',
            'prepend': 'django.middleware.cache.UpdateCacheMiddleware',
            'remove': [
                'django.contrib.sessions.middleware.SessionMiddleware',
                'django.contrib.auth.middleware.AuthenticationMiddleware',
                'django.contrib.messages.middleware.MessageMiddleware',
            ],
        }):
            response = self.client.get('/')
            # ...
```

For each action, you can supply either a list of values or a string. When the value already exists in the list, `append` and `prepend` have no effect; neither does `remove` when the value doesn't exist.

override_settings()

In case you want to override a setting for a test method, Django provides the *override_settings()* decorator (see **PEP 318**). It's used like this:

```
from django.test import TestCase, override_settings

class LoginTestCase(TestCase):

    @override_settings(LOGIN_URL='/other/login/')
    def test_login(self):
        response = self.client.get('/sekrit/')
        self.assertRedirects(response, '/other/login/?next=/sekrit/')
```

The decorator can also be applied to *TestCase* classes:

```
from django.test import TestCase, override_settings

@override_settings(LOGIN_URL='/other/login/')
class LoginTestCase(TestCase):

    def test_login(self):
        response = self.client.get('/sekrit/')
        self.assertRedirects(response, '/other/login/?next=/sekrit/')
```

modify_settings()

Likewise, Django provides the *modify_settings()* decorator:

```
from django.test import TestCase, modify_settings

class MiddlewareTestCase(TestCase):

    @modify_settings(MIDDLEWARE={
        'append': 'django.middleware.cache.FetchFromCacheMiddleware',
        'prepend': 'django.middleware.cache.UpdateCacheMiddleware',
    })
    def test_cache_middleware(self):
        response = self.client.get('/')
        # ...
```

The decorator can also be applied to test case classes:

```
from django.test import TestCase, modify_settings

@modify_settings(MIDDLEWARE={
    'append': 'django.middleware.cache.FetchFromCacheMiddleware',
    'prepend': 'django.middleware.cache.UpdateCacheMiddleware',
})
class MiddlewareTestCase(TestCase):

    def test_cache_middleware(self):
        response = self.client.get('/')
        # ...
```

: When given a class, these decorators modify the class directly and return it; they don't create and return a modified copy of it. So if you try to tweak the above examples to assign the return value to a different name than `LoginTestCase` or `MiddlewareTestCase`, you may be surprised to find that the original test case classes are still equally affected by the decorator. For a given class, *`modify_settings()`* is always applied after *`override_settings()`*.

: The settings file contains some settings that are only consulted during initialization of Django internals. If you change them with `override_settings`, the setting is changed if you access it via the `django.conf.settings` module, however, Django's internals access it differently. Effectively, using *`override_settings()`* or *`modify_settings()`* with these settings is probably not going to do what you expect it to do.
We do not recommend altering the *`DATABASES`* setting. Altering the *`CACHES`* setting is possible, but a bit tricky if you are using internals that make using of caching, like *`django.contrib.sessions`*. For example, you will have to reinitialize the session backend in a test that uses cached sessions and overrides *`CACHES`*.
Finally, avoid aliasing your settings as module-level constants as `override_settings()` won't work on such values since they are only evaluated the first time the module is imported.

You can also simulate the absence of a setting by deleting it after settings have been overridden, like this:

```
@override_settings()
def test_something(self):
    del settings.LOGIN_URL
    ...
```

When overriding settings, make sure to handle the cases in which your app's code uses a cache or similar feature that retains state even if the setting is changed. Django provides the *`django.test.signals.setting_changed`* signal that lets you register callbacks to clean up and otherwise reset state when settings are changed.

Django itself uses this signal to reset various data:

Overridden settings	Data reset
USE_TZ, TIME_ZONE	Databases timezone
TEMPLATES	Template engines
SERIALIZATION_MODULES	Serializers cache
LOCALE_PATHS, LANGUAGE_CODE	Default translation and loaded translations
MEDIA_ROOT, DEFAULT_FILE_STORAGE	Default file storage

Emptying the test outbox

If you use any of Django's custom `TestCase` classes, the test runner will clear the contents of the test email outbox at the start of each test case.

For more detail on email services during tests, see *Email services* below.

Assertions

As Python's normal `unittest.TestCase` class implements assertion methods such as `assertTrue()` and `assertEqual()`, Django's custom *`TestCase`* class provides a number of custom assertion methods that are useful for testing Web applications:

The failure messages given by most of these assertion methods can be customized with the `msg_prefix` argument. This string will be prefixed to any failure message generated by the assertion. This allows you to provide additional details that may help you to identify the location and cause of a failure in your test suite.

`SimpleTestCase.`**`assertRaisesMessage`**(*expected_exception*, *expected_message*, *callable*, **args*, ***kwargs*)

`SimpleTestCase.`**`assertRaisesMessage`**(*expected_exception*, *expected_message*)

Asserts that execution of `callable` raises `expected_exception` and that `expected_message` is found in the exception's message. Any other outcome is reported as a failure. It's a simpler version of `unittest.TestCase.assertRaisesRegex()` with the difference that `expected_message` isn't treated as a regular expression.

If only the `expected_exception` and `expected_message` parameters are given, returns a context manager so that the code being tested can be written inline rather than as a function:

```
with self.assertRaisesMessage(ValueError, 'invalid literal for int()'):
    int('a')
```

1.9 : Passing `callable` as a keyword argument called `callable_obj` is deprecated. Pass the callable as a positional argument instead.

`SimpleTestCase.`**`assertFieldOutput`**(*fieldclass*, *valid*, *invalid*, *field_args=None*, *field_kwargs=None*, *empty_value=''*)

Asserts that a form field behaves correctly with various inputs.

- **`fieldclass`** – the class of the field to be tested.
- **`valid`** – a dictionary mapping valid inputs to their expected cleaned values.
- **`invalid`** – a dictionary mapping invalid inputs to one or more raised error messages.
- **`field_args`** – the args passed to instantiate the field.
- **`field_kwargs`** – the kwargs passed to instantiate the field.
- **`empty_value`** – the expected clean output for inputs in `empty_values`.

For example, the following code tests that an `EmailField` accepts `a@a.com` as a valid email address, but rejects `aaa` with a reasonable error message:

```
self.assertFieldOutput(EmailField, {'a@a.com': 'a@a.com'}, {'aaa': ['Enter a valid email address
```

`SimpleTestCase.`**`assertFormError`**(*response*, *form*, *field*, *errors*, *msg_prefix=''*)

Asserts that a field on a form raises the provided list of errors when rendered on the form.

`form` is the name the `Form` instance was given in the template context.

`field` is the name of the field on the form to check. If `field` has a value of `None`, non-field errors (errors you can access via *`form.non_field_errors()`*) will be checked.

`errors` is an error string, or a list of error strings, that are expected as a result of form validation.

`SimpleTestCase.`**`assertFormsetError`**(*response*, *formset*, *form_index*, *field*, *errors*, *msg_prefix=''*)
Asserts that the `formset` raises the provided list of errors when rendered.

`formset` is the name the `Formset` instance was given in the template context.

`form_index` is the number of the form within the `Formset`. If `form_index` has a value of `None`, non-form errors (errors you can access via `formset.non_form_errors()`) will be checked.

`field` is the name of the field on the form to check. If `field` has a value of `None`, non-field errors (errors you can access via *`form.non_field_errors()`*) will be checked.

`errors` is an error string, or a list of error strings, that are expected as a result of form validation.

`SimpleTestCase.`**`assertContains`**(*response*, *text*, *count=None*, *status_code=200*, *msg_prefix=''*, *html=False*)
Asserts that a `Response` instance produced the given `status_code` and that `text` appears in the content of the response. If `count` is provided, `text` must occur exactly `count` times in the response.

Set `html` to `True` to handle `text` as HTML. The comparison with the response content will be based on HTML semantics instead of character-by-character equality. Whitespace is ignored in most cases, attribute ordering is not significant. See *`assertHTMLEqual()`* for more details.

`SimpleTestCase.`**`assertNotContains`**(*response*, *text*, *status_code=200*, *msg_prefix=''*, *html=False*)
Asserts that a `Response` instance produced the given `status_code` and that `text` does *not* appear in the content of the response.

Set `html` to `True` to handle `text` as HTML. The comparison with the response content will be based on HTML semantics instead of character-by-character equality. Whitespace is ignored in most cases, attribute ordering is not significant. See *`assertHTMLEqual()`* for more details.

`SimpleTestCase.`**`assertTemplateUsed`**(*response*, *template_name*, *msg_prefix=''*, *count=None*)
Asserts that the template with the given name was used in rendering the response.

The name is a string such as `'admin/index.html'`.

The count argument is an integer indicating the number of times the template should be rendered. Default is `None`, meaning that the template should be rendered one or more times.

You can use this as a context manager, like this:

```
    with self.assertTemplateUsed('index.html'):
        render_to_string('index.html')
    with self.assertTemplateUsed(template_name='index.html'):
        render_to_string('index.html')
```

`SimpleTestCase.`**`assertTemplateNotUsed`**(*response*, *template_name*, *msg_prefix=''*)
Asserts that the template with the given name was *not* used in rendering the response.

You can use this as a context manager in the same way as *`assertTemplateUsed()`*.

`SimpleTestCase.`**`assertRedirects`**(*response*, *expected_url*, *status_code=302*, *target_status_code=200*, *msg_prefix=''*, *fetch_redirect_response=True*)
Asserts that the response returned a `status_code` redirect status, redirected to `expected_url` (including any `GET` data), and that the final page was received with `target_status_code`.

If your request used the `follow` argument, the `expected_url` and `target_status_code` will be the url and status code for the final point of the redirect chain.

If `fetch_redirect_response` is `False`, the final page won't be loaded. Since the test client can't fetch externals URLs, this is particularly useful if `expected_url` isn't part of your Django app.

Scheme is handled correctly when making comparisons between two URLs. If there isn't any scheme specified in the location where we are redirected to, the original request's scheme is used. If present, the scheme in `expected_url` is the one used to make the comparisons to.

1.9 : The `host` argument is deprecated, as redirections are no longer forced to be absolute URLs.

`SimpleTestCase.`**`assertHTMLEqual`**(*html1*, *html2*, *msg=None*)

Asserts that the strings `html1` and `html2` are equal. The comparison is based on HTML semantics. The comparison takes following things into account:

- Whitespace before and after HTML tags is ignored.
- All types of whitespace are considered equivalent.
- All open tags are closed implicitly, e.g. when a surrounding tag is closed or the HTML document ends.
- Empty tags are equivalent to their self-closing version.
- The ordering of attributes of an HTML element is not significant.
- Attributes without an argument are equal to attributes that equal in name and value (see the examples).

The following examples are valid tests and don't raise any `AssertionError`:

```
self.assertHTMLEqual(
    '<p>Hello <b>world!</p>',
    '''<p>
        Hello   <b>world! <b/>
    </p>'''
)
self.assertHTMLEqual(
    '<input type="checkbox" checked="checked" id="id_accept_terms" />',
    '<input id="id_accept_terms" type="checkbox" checked>'
)
```

`html1` and `html2` must be valid HTML. An `AssertionError` will be raised if one of them cannot be parsed.

Output in case of error can be customized with the `msg` argument.

`SimpleTestCase.`**`assertHTMLNotEqual`**(*html1*, *html2*, *msg=None*)

Asserts that the strings `html1` and `html2` are *not* equal. The comparison is based on HTML semantics. See *`assertHTMLEqual()`* for details.

`html1` and `html2` must be valid HTML. An `AssertionError` will be raised if one of them cannot be parsed.

Output in case of error can be customized with the `msg` argument.

`SimpleTestCase.`**`assertXMLEqual`**(*xml1*, *xml2*, *msg=None*)

Asserts that the strings `xml1` and `xml2` are equal. The comparison is based on XML semantics. Similarly to *`assertHTMLEqual()`*, the comparison is made on parsed content, hence only semantic differences are considered, not syntax differences. When invalid XML is passed in any parameter, an `AssertionError` is always raised, even if both string are identical.

Output in case of error can be customized with the `msg` argument.

`SimpleTestCase.`**`assertXMLNotEqual`**(*xml1*, *xml2*, *msg=None*)

Asserts that the strings `xml1` and `xml2` are *not* equal. The comparison is based on XML semantics. See *`assertXMLEqual()`* for details.

Output in case of error can be customized with the `msg` argument.

`SimpleTestCase.`**`assertInHTML`**(*needle*, *haystack*, *count=None*, *msg_prefix=''*)
Asserts that the HTML fragment `needle` is contained in the `haystack` one.

If the `count` integer argument is specified, then additionally the number of `needle` occurrences will be strictly verified.

Whitespace in most cases is ignored, and attribute ordering is not significant. The passed-in arguments must be valid HTML.

`SimpleTestCase.`**`assertJSONEqual`**(*raw*, *expected_data*, *msg=None*)
Asserts that the JSON fragments `raw` and `expected_data` are equal. Usual JSON non-significant whitespace rules apply as the heavyweight is delegated to the `json` library.

Output in case of error can be customized with the `msg` argument.

`SimpleTestCase.`**`assertJSONNotEqual`**(*raw*, *expected_data*, *msg=None*)
Asserts that the JSON fragments `raw` and `expected_data` are *not* equal. See *`assertJSONEqual()`* for further details.

Output in case of error can be customized with the `msg` argument.

`TransactionTestCase.`**`assertQuerysetEqual`**(*qs*, *values*, *transform=repr*, *ordered=True*, *msg=None*)
Asserts that a queryset `qs` returns a particular list of values `values`.

The comparison of the contents of `qs` and `values` is performed using the function `transform`; by default, this means that the `repr()` of each value is compared. Any other callable can be used if `repr()` doesn't provide a unique or helpful comparison.

By default, the comparison is also ordering dependent. If `qs` doesn't provide an implicit ordering, you can set the `ordered` parameter to `False`, which turns the comparison into a `collections.Counter` comparison. If the order is undefined (if the given `qs` isn't ordered and the comparison is against more than one ordered values), a `ValueError` is raised.

Output in case of error can be customized with the `msg` argument.

`TransactionTestCase.`**`assertNumQueries`**(*num*, *func*, **args*, ***kwargs*)
Asserts that when `func` is called with `*args` and `**kwargs` that `num` database queries are executed.

If a `"using"` key is present in `kwargs` it is used as the database alias for which to check the number of queries. If you wish to call a function with a `using` parameter you can do it by wrapping the call with a `lambda` to add an extra parameter:

```
self.assertNumQueries(7, lambda: my_function(using=7))
```

You can also use this as a context manager:

```
with self.assertNumQueries(2):
    Person.objects.create(name="Aaron")
    Person.objects.create(name="Daniel")
```

Tagging tests

You can tag your tests so you can easily run a particular subset. For example, you might label fast or slow tests:

```
from django.test import tag

class SampleTestCase(TestCase):
```

```
    @tag('fast')
    def test_fast(self):
        ...

    @tag('slow')
    def test_slow(self):
        ...

    @tag('slow', 'core')
    def test_slow_but_core(self):
        ...
```

You can also tag a test case:

```
@tag('slow', 'core')
class SampleTestCase(TestCase):
    ...
```

Then you can choose which tests to run. For example, to run only fast tests:

```
$ ./manage.py test --tag=fast
```

Or to run fast tests and the core one (even though it's slow):

```
$ ./manage.py test --tag=fast --tag=core
```

You can also exclude tests by tag. To run core tests if they are not slow:

```
$ ./manage.py test --tag=core --exclude-tag=slow
```

test --exclude-tag has precedence over *test --tag*, so if a test has two tags and you select one of them and exclude the other, the test won't be run.

Email services

If any of your Django views send email using Django's email functionality, you probably don't want to send email each time you run a test using that view. For this reason, Django's test runner automatically redirects all Django-sent email to a dummy outbox. This lets you test every aspect of sending email – from the number of messages sent to the contents of each message – without actually sending the messages.

The test runner accomplishes this by transparently replacing the normal email backend with a testing backend. (Don't worry – this has no effect on any other email senders outside of Django, such as your machine's mail server, if you're running one.)

`django.core.mail.`**`outbox`**

During test running, each outgoing email is saved in `django.core.mail.outbox`. This is a simple list of all *EmailMessage* instances that have been sent. The `outbox` attribute is a special attribute that is created *only* when the `locmem` email backend is used. It doesn't normally exist as part of the *django.core.mail* module and you can't import it directly. The code below shows how to access this attribute correctly.

Here's an example test that examines `django.core.mail.outbox` for length and contents:

```
from django.core import mail
from django.test import TestCase

class EmailTest(TestCase):
    def test_send_email(self):
        # Send message.
```

```
        mail.send_mail(
            'Subject here', 'Here is the message.',
            'from@example.com', ['to@example.com'],
            fail_silently=False,
        )

        # Test that one message has been sent.
        self.assertEqual(len(mail.outbox), 1)

        # Verify that the subject of the first message is correct.
        self.assertEqual(mail.outbox[0].subject, 'Subject here')
```

As noted *previously*, the test outbox is emptied at the start of every test in a Django `*TestCase`. To empty the outbox manually, assign the empty list to `mail.outbox`:

```
from django.core import mail

# Empty the test outbox
mail.outbox = []
```

Management Commands

Management commands can be tested with the *`call_command()`* function. The output can be redirected into a `StringIO` instance:

```
from django.core.management import call_command
from django.test import TestCase
from django.utils.six import StringIO

class ClosepollTest(TestCase):
    def test_command_output(self):
        out = StringIO()
        call_command('closepoll', stdout=out)
        self.assertIn('Expected output', out.getvalue())
```

Skipping tests

The unittest library provides the `@skipIf` and `@skipUnless` decorators to allow you to skip tests if you know ahead of time that those tests are going to fail under certain conditions.

For example, if your test requires a particular optional library in order to succeed, you could decorate the test case with `@skipIf`. Then, the test runner will report that the test wasn't executed and why, instead of failing the test or omitting the test altogether.

To supplement these test skipping behaviors, Django provides two additional skip decorators. Instead of testing a generic boolean, these decorators check the capabilities of the database, and skip the test if the database doesn't support a specific named feature.

The decorators use a string identifier to describe database features. This string corresponds to attributes of the database connection features class. See `django.db.backends.BaseDatabaseFeatures` class for a full list of database features that can be used as a basis for skipping tests.

`skipIfDBFeature`(**feature_name_strings*)

Skip the decorated test or `TestCase` if all of the named database features are supported.

For example, the following test will not be executed if the database supports transactions (e.g., it would *not* run under PostgreSQL, but it would under MySQL with MyISAM tables):

```
class MyTests(TestCase):
    @skipIfDBFeature('supports_transactions')
    def test_transaction_behavior(self):
        # ... conditional test code
        pass
```

skipUnlessDBFeature(**feature_name_strings*)

Skip the decorated test or `TestCase` if any of the named database features are *not* supported.

For example, the following test will only be executed if the database supports transactions (e.g., it would run under PostgreSQL, but *not* under MySQL with MyISAM tables):

```
class MyTests(TestCase):
    @skipUnlessDBFeature('supports_transactions')
    def test_transaction_behavior(self):
        # ... conditional test code
        pass
```

3.9.3 Advanced testing topics

The request factory

class RequestFactory

The *RequestFactory* shares the same API as the test client. However, instead of behaving like a browser, the RequestFactory provides a way to generate a request instance that can be used as the first argument to any view. This means you can test a view function the same way as you would test any other function – as a black box, with exactly known inputs, testing for specific outputs.

The API for the *RequestFactory* is a slightly restricted subset of the test client API:

- It only has access to the HTTP methods *get()*, *post()*, *put()*, *delete()*, *head()*, *options()*, and *trace()*.
- These methods accept all the same arguments *except* for `follows`. Since this is just a factory for producing requests, it's up to you to handle the response.
- It does not support middleware. Session and authentication attributes must be supplied by the test itself if required for the view to function properly.

Example

The following is a simple unit test using the request factory:

```
from django.contrib.auth.models import AnonymousUser, User
from django.test import TestCase, RequestFactory

from .views import MyView, my_view

class SimpleTest(TestCase):
    def setUp(self):
        # Every test needs access to the request factory.
        self.factory = RequestFactory()
        self.user = User.objects.create_user(
```

```
            username='jacob', email='jacob@...', password='top_secret')

    def test_details(self):
        # Create an instance of a GET request.
        request = self.factory.get('/customer/details')

        # Recall that middleware are not supported. You can simulate a
        # logged-in user by setting request.user manually.
        request.user = self.user

        # Or you can simulate an anonymous user by setting request.user to
        # an AnonymousUser instance.
        request.user = AnonymousUser()

        # Test my_view() as if it were deployed at /customer/details
        response = my_view(request)
        # Use this syntax for class-based views.
        response = MyView.as_view()(request)
        self.assertEqual(response.status_code, 200)
```

Tests and multiple databases

Testing primary/replica configurations

If you're testing a multiple database configuration with primary/replica (referred to as master/slave by some databases) replication, this strategy of creating test databases poses a problem. When the test databases are created, there won't be any replication, and as a result, data created on the primary won't be seen on the replica.

To compensate for this, Django allows you to define that a database is a *test mirror*. Consider the following (simplified) example database configuration:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'myproject',
        'HOST': 'dbprimary',
         # ... plus some other settings
    },
    'replica': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'myproject',
        'HOST': 'dbreplica',
        'TEST': {
            'MIRROR': 'default',
        },
        # ... plus some other settings
    }
}
```

In this setup, we have two database servers: `dbprimary`, described by the database alias `default`, and `dbreplica` described by the alias `replica`. As you might expect, `dbreplica` has been configured by the database administrator as a read replica of `dbprimary`, so in normal activity, any write to `default` will appear on `replica`.

If Django created two independent test databases, this would break any tests that expected replication to occur. However, the `replica` database has been configured as a test mirror (using the *MIRROR* test setting), indicating that under testing, `replica` should be treated as a mirror of `default`.

When the test environment is configured, a test version of `replica` will *not* be created. Instead the connection to `replica` will be redirected to point at `default`. As a result, writes to `default` will appear on `replica` – but because they are actually the same database, not because there is data replication between the two databases.

Controlling creation order for test databases

By default, Django will assume all databases depend on the `default` database and therefore always create the `default` database first. However, no guarantees are made on the creation order of any other databases in your test setup.

If your database configuration requires a specific creation order, you can specify the dependencies that exist using the *DEPENDENCIES* test setting. Consider the following (simplified) example database configuration:

```
DATABASES = {
    'default': {
        # ... db settings
        'TEST': {
            'DEPENDENCIES': ['diamonds'],
        },
    },
    'diamonds': {
         ... db settings
        'TEST': {
            'DEPENDENCIES': [],
        },
    },
    'clubs': {
        # ... db settings
        'TEST': {
            'DEPENDENCIES': ['diamonds'],
        },
    },
    'spades': {
        # ... db settings
        'TEST': {
            'DEPENDENCIES': ['diamonds', 'hearts'],
        },
    },
    'hearts': {
        # ... db settings
        'TEST': {
            'DEPENDENCIES': ['diamonds', 'clubs'],
        },
    }
}
```

Under this configuration, the `diamonds` database will be created first, as it is the only database alias without dependencies. The `default` and `clubs` alias will be created next (although the order of creation of this pair is not guaranteed), then `hearts`, and finally `spades`.

If there are any circular dependencies in the *DEPENDENCIES* definition, an *ImproperlyConfigured* exception will be raised.

Advanced features of `TransactionTestCase`

`TransactionTestCase.`**`available_apps`**

> **:** This attribute is a private API. It may be changed or removed without a deprecation period in the future, for instance to accommodate changes in application loading.
> It's used to optimize Django's own test suite, which contains hundreds of models but no relations between models in different applications.

By default, `available_apps` is set to `None`. After each test, Django calls *`flush`* to reset the database state. This empties all tables and emits the *`post_migrate`* signal, which re-creates one content type and three permissions for each model. This operation gets expensive proportionally to the number of models.

Setting `available_apps` to a list of applications instructs Django to behave as if only the models from these applications were available. The behavior of `TransactionTestCase` changes as follows:

- *`post_migrate`* is fired before each test to create the content types and permissions for each model in available apps, in case they're missing.
- After each test, Django empties only tables corresponding to models in available apps. However, at the database level, truncation may cascade to related models in unavailable apps. Furthermore *`post_migrate`* isn't fired; it will be fired by the next `TransactionTestCase`, after the correct set of applications is selected.

Since the database isn't fully flushed, if a test creates instances of models not included in `available_apps`, they will leak and they may cause unrelated tests to fail. Be careful with tests that use sessions; the default session engine stores them in the database.

Since *`post_migrate`* isn't emitted after flushing the database, its state after a `TransactionTestCase` isn't the same as after a `TestCase`: it's missing the rows created by listeners to *`post_migrate`*. Considering the *order in which tests are executed*, this isn't an issue, provided either all `TransactionTestCase` in a given test suite declare `available_apps`, or none of them.

`available_apps` is mandatory in Django's own test suite.

`TransactionTestCase.`**`reset_sequences`**

Setting `reset_sequences = True` on a `TransactionTestCase` will make sure sequences are always reset before the test run:

```
class TestsThatDependsOnPrimaryKeySequences(TransactionTestCase):
    reset_sequences = True

    def test_animal_pk(self):
        lion = Animal.objects.create(name="lion", sound="roar")
        # lion.pk is guaranteed to always be 1
        self.assertEqual(lion.pk, 1)
```

Unless you are explicitly testing primary keys sequence numbers, it is recommended that you do not hard code primary key values in tests.

Using `reset_sequences = True` will slow down the test, since the primary key reset is an relatively expensive database operation.

Using the Django test runner to test reusable applications

If you are writing a reusable application you may want to use the Django test runner to run your own test suite and thus benefit from the Django testing infrastructure.

A common practice is a *tests* directory next to the application code, with the following structure:

```
runtests.py
polls/
    __init__.py
    models.py
    ...
tests/
    __init__.py
    models.py
    test_settings.py
    tests.py
```

Let's take a look inside a couple of those files:

runtests.py

```
#!/usr/bin/env python
import os
import sys

import django
from django.conf import settings
from django.test.utils import get_runner

if __name__ == "__main__":
    os.environ['DJANGO_SETTINGS_MODULE'] = 'tests.test_settings'
    django.setup()
    TestRunner = get_runner(settings)
    test_runner = TestRunner()
    failures = test_runner.run_tests(["tests"])
    sys.exit(bool(failures))
```

This is the script that you invoke to run the test suite. It sets up the Django environment, creates the test database and runs the tests.

For the sake of clarity, this example contains only the bare minimum necessary to use the Django test runner. You may want to add command-line options for controlling verbosity, passing in specific test labels to run, etc.

tests/test_settings.py

```
SECRET_KEY = 'fake-key'
INSTALLED_APPS = [
    "tests",
]
```

This file contains the Django settings required to run your app's tests.

Again, this is a minimal example; your tests may require additional settings to run.

Since the *tests* package is included in *INSTALLED_APPS* when running your tests, you can define test-only models in its `models.py` file.

Using different testing frameworks

Clearly, `unittest` is not the only Python testing framework. While Django doesn't provide explicit support for alternative frameworks, it does provide a way to invoke tests constructed for an alternative framework as if they were normal Django tests.

When you run `./manage.py test`, Django looks at the *TEST_RUNNER* setting to determine what to do. By default, *TEST_RUNNER* points to `'django.test.runner.DiscoverRunner'`. This class defines the default Django testing behavior. This behavior involves:

1. Performing global pre-test setup.
2. Looking for tests in any file below the current directory whose name matches the pattern `test*.py`.
3. Creating the test databases.
4. Running `migrate` to install models and initial data into the test databases.
5. Running the tests that were found.
6. Destroying the test databases.
7. Performing global post-test teardown.

If you define your own test runner class and point *TEST_RUNNER* at that class, Django will execute your test runner whenever you run `./manage.py test`. In this way, it is possible to use any test framework that can be executed from Python code, or to modify the Django test execution process to satisfy whatever testing requirements you may have.

Defining a test runner

A test runner is a class defining a `run_tests()` method. Django ships with a `DiscoverRunner` class that defines the default Django testing behavior. This class defines the `run_tests()` entry point, plus a selection of other methods that are used to by `run_tests()` to set up, execute and tear down the test suite.

class **`DiscoverRunner`**(*pattern='test*.py'*, *top_level=None*, *verbosity=1*, *interactive=True*, *failfast=False*, *keepdb=False*, *reverse=False*, *debug_sql=False*, ***kwargs*)

`DiscoverRunner` will search for tests in any file matching `pattern`.

`top_level` can be used to specify the directory containing your top-level Python modules. Usually Django can figure this out automatically, so it's not necessary to specify this option. If specified, it should generally be the directory containing your `manage.py` file.

`verbosity` determines the amount of notification and debug information that will be printed to the console; `0` is no output, `1` is normal output, and `2` is verbose output.

If `interactive` is `True`, the test suite has permission to ask the user for instructions when the test suite is executed. An example of this behavior would be asking for permission to delete an existing test database. If `interactive` is `False`, the test suite must be able to run without any manual intervention.

If `failfast` is `True`, the test suite will stop running after the first test failure is detected.

If `keepdb` is `True`, the test suite will use the existing database, or create one if necessary. If `False`, a new database will be created, prompting the user to remove the existing one, if present.

If `reverse` is `True`, test cases will be executed in the opposite order. This could be useful to debug tests that aren't properly isolated and have side effects. *Grouping by test class* is preserved when using this option.

If `debug_sql` is `True`, failing test cases will output SQL queries logged to the *django.db.backends logger* as well as the traceback. If `verbosity` is `2`, then queries in all tests are output.

Django may, from time to time, extend the capabilities of the test runner by adding new arguments. The `**kwargs` declaration allows for this expansion. If you subclass `DiscoverRunner` or write your own test runner, ensure it accepts `**kwargs`.

Your test runner may also define additional command-line options. Create or override an `add_arguments(cls, parser)` class method and add custom arguments by calling `parser.add_argument()` inside the method, so that the *test* command will be able to use those arguments.

Attributes

`DiscoverRunner.`**`test_suite`**

The class used to build the test suite. By default it is set to `unittest.TestSuite`. This can be overridden if you wish to implement different logic for collecting tests.

`DiscoverRunner.`**`test_runner`**

This is the class of the low-level test runner which is used to execute the individual tests and format the results. By default it is set to `unittest.TextTestRunner`. Despite the unfortunate similarity in naming conventions, this is not the same type of class as `DiscoverRunner`, which covers a broader set of responsibilities. You can override this attribute to modify the way tests are run and reported.

`DiscoverRunner.`**`test_loader`**

This is the class that loads tests, whether from TestCases or modules or otherwise and bundles them into test suites for the runner to execute. By default it is set to `unittest.defaultTestLoader`. You can override this attribute if your tests are going to be loaded in unusual ways.

Methods

`DiscoverRunner.`**`run_tests`**(*test_labels*, *extra_tests=None*, ***kwargs*)

Run the test suite.

`test_labels` allows you to specify which tests to run and supports several formats (see *`DiscoverRunner.build_suite()`* for a list of supported formats).

`extra_tests` is a list of extra `TestCase` instances to add to the suite that is executed by the test runner. These extra tests are run in addition to those discovered in the modules listed in `test_labels`.

This method should return the number of tests that failed.

classmethod `DiscoverRunner.`**`add_arguments`**(*parser*)

Override this class method to add custom arguments accepted by the *`test`* management command. See `argparse.ArgumentParser.add_argument()` for details about adding arguments to a parser.

`DiscoverRunner.`**`setup_test_environment`**(***kwargs*)

Sets up the test environment by calling *`setup_test_environment()`* and setting *`DEBUG`* to `False`.

`DiscoverRunner.`**`build_suite`**(*test_labels*, *extra_tests=None*, ***kwargs*)

Constructs a test suite that matches the test labels provided.

`test_labels` is a list of strings describing the tests to be run. A test label can take one of four forms:

- `path.to.test_module.TestCase.test_method` – Run a single test method in a test case.
- `path.to.test_module.TestCase` – Run all the test methods in a test case.
- `path.to.module` – Search for and run all tests in the named Python package or module.
- `path/to/directory` – Search for and run all tests below the named directory.

If `test_labels` has a value of `None`, the test runner will search for tests in all files below the current directory whose names match its `pattern` (see above).

`extra_tests` is a list of extra `TestCase` instances to add to the suite that is executed by the test runner. These extra tests are run in addition to those discovered in the modules listed in `test_labels`.

Returns a `TestSuite` instance ready to be run.

`DiscoverRunner.`**`setup_databases`**(***kwargs*)

Creates the test databases.

Returns a data structure that provides enough detail to undo the changes that have been made. This data will be provided to the `teardown_databases()` function at the conclusion of testing.

DiscoverRunner.**run_suite**(*suite*, ***kwargs*)
Runs the test suite.

Returns the result produced by the running the test suite.

DiscoverRunner.**teardown_databases**(*old_config*, ***kwargs*)
Destroys the test databases, restoring pre-test conditions.

`old_config` is a data structure defining the changes in the database configuration that need to be reversed. It is the return value of the `setup_databases()` method.

DiscoverRunner.**teardown_test_environment**(***kwargs*)
Restores the pre-test environment.

DiscoverRunner.**suite_result**(*suite*, *result*, ***kwargs*)
Computes and returns a return code based on a test suite, and the result from that test suite.

Testing utilities

django.test.utils To assist in the creation of your own test runner, Django provides a number of utility methods in the `django.test.utils` module.

setup_test_environment()
Performs global pre-test setup, such as installing instrumentation for the template rendering system and setting up the dummy email outbox.

teardown_test_environment()
Performs global post-test teardown, such as removing instrumentation from the template system and restoring normal email services.

django.db.connection.creation The creation module of the database backend also provides some utilities that can be useful during testing.

create_test_db(*verbosity=1*, *autoclobber=False*, *serialize=True*, *keepdb=False*)
Creates a new test database and runs `migrate` against it.

`verbosity` has the same behavior as in `run_tests()`.

`autoclobber` describes the behavior that will occur if a database with the same name as the test database is discovered:

- If `autoclobber` is `False`, the user will be asked to approve destroying the existing database. `sys.exit` is called if the user does not approve.
- If autoclobber is `True`, the database will be destroyed without consulting the user.

`serialize` determines if Django serializes the database into an in-memory JSON string before running tests (used to restore the database state between tests if you don't have transactions). You can set this to `False` to speed up creation time if you don't have any test classes with *serialized_rollback=True*.

If you are using the default test runner, you can control this with the the *SERIALIZE* entry in the *TEST* dictionary.

`keepdb` determines if the test run should use an existing database, or create a new one. If `True`, the existing database will be used, or created if not present. If `False`, a new database will be created, prompting the user to remove the existing one, if present.

Returns the name of the test database that it created.

`create_test_db()` has the side effect of modifying the value of *NAME* in *DATABASES* to match the name of the test database.

destroy_test_db(*old_database_name*, *verbosity=1*, *keepdb=False*)

Destroys the database whose name is the value of *NAME* in *DATABASES*, and sets *NAME* to the value of `old_database_name`.

The `verbosity` argument has the same behavior as for *DiscoverRunner*.

If the `keepdb` argument is `True`, then the connection to the database will be closed, but the database will not be destroyed.

Integration with `coverage.py`

Code coverage describes how much source code has been tested. It shows which parts of your code are being exercised by tests and which are not. It's an important part of testing applications, so it's strongly recommended to check the coverage of your tests.

Django can be easily integrated with coverage.py, a tool for measuring code coverage of Python programs. First, install coverage.py. Next, run the following from your project folder containing `manage.py`:

```
coverage run --source='.' manage.py test myapp
```

This runs your tests and collects coverage data of the executed files in your project. You can see a report of this data by typing following command:

```
coverage report
```

Note that some Django code was executed while running tests, but it is not listed here because of the `source` flag passed to the previous command.

For more options like annotated HTML listings detailing missed lines, see the coverage.py docs.

3.10 User authentication in Django

3.10.1 Using the Django authentication system

This document explains the usage of Django's authentication system in its default configuration. This configuration has evolved to serve the most common project needs, handling a reasonably wide range of tasks, and has a careful implementation of passwords and permissions. For projects where authentication needs differ from the default, Django supports extensive extension and customization of authentication.

Django authentication provides both authentication and authorization together and is generally referred to as the authentication system, as these features are somewhat coupled.

`User` objects

User objects are the core of the authentication system. They typically represent the people interacting with your site and are used to enable things like restricting access, registering user profiles, associating content with creators etc. Only one class of user exists in Django's authentication framework, i.e., *'superusers'* or admin *'staff'* users are just user objects with special attributes set, not different classes of user objects.

The primary attributes of the default user are:

- *username*
- *password*
- *email*

- *first_name*
- *last_name*

See the *full API documentation* for full reference, the documentation that follows is more task oriented.

Creating users

The most direct way to create users is to use the included *create_user()* helper function:

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.create_user('john', 'lennon@thebeatles.com', 'johnpassword')

# At this point, user is a User object that has already been saved
# to the database. You can continue to change its attributes
# if you want to change other fields.
>>> user.last_name = 'Lennon'
>>> user.save()
```

If you have the Django admin installed, you can also *create users interactively*.

Creating superusers

Create superusers using the *createsuperuser* command:

```
$ python manage.py createsuperuser --username=joe --email=joe@example.com
```

You will be prompted for a password. After you enter one, the user will be created immediately. If you leave off the *--username* or *--email* options, it will prompt you for those values.

Changing passwords

Django does not store raw (clear text) passwords on the user model, but only a hash (see documentation of how passwords are managed for full details). Because of this, do not attempt to manipulate the password attribute of the user directly. This is why a helper function is used when creating a user.

To change a user's password, you have several options:

*manage.py changepassword *username** offers a method of changing a User's password from the command line. It prompts you to change the password of a given user which you must enter twice. If they both match, the new password will be changed immediately. If you do not supply a user, the command will attempt to change the password whose username matches the current system user.

You can also change a password programmatically, using *set_password()*:

```
>>> from django.contrib.auth.models import User
>>> u = User.objects.get(username='john')
>>> u.set_password('new password')
>>> u.save()
```

If you have the Django admin installed, you can also change user's passwords on the *authentication system's admin pages*.

Django also provides *views* and *forms* that may be used to allow users to change their own passwords.

Changing a user's password will log out all their sessions. See *Session invalidation on password change* for details.

Authenticating users

authenticate(***credentials*)

Use *authenticate()* to verify a set of credentials. It takes credentials as keyword arguments, username and password for the default case, checks them against each *authentication backend*, and returns a *User* object if the credentials are valid for a backend. If the credentials aren't valid for any backend or if a backend raises *PermissionDenied*, it returns None. For example:

```
from django.contrib.auth import authenticate
user = authenticate(username='john', password='secret')
if user is not None:
    # A backend authenticated the credentials
else:
    # No backend authenticated the credentials
```

: This is a low level way to authenticate a set of credentials; for example, it's used by the *RemoteUserMiddleware*. Unless you are writing your own authentication system, you probably won't use this. Rather if you are looking for a way to limit access to logged in users, see the *login_required()* decorator.

Permissions and Authorization

Django comes with a simple permissions system. It provides a way to assign permissions to specific users and groups of users.

It's used by the Django admin site, but you're welcome to use it in your own code.

The Django admin site uses permissions as follows:

- Access to view the "add" form and add an object is limited to users with the "add" permission for that type of object.
- Access to view the change list, view the "change" form and change an object is limited to users with the "change" permission for that type of object.
- Access to delete an object is limited to users with the "delete" permission for that type of object.

Permissions can be set not only per type of object, but also per specific object instance. By using the *has_add_permission()*, *has_change_permission()* and *has_delete_permission()* methods provided by the *ModelAdmin* class, it is possible to customize permissions for different object instances of the same type.

User objects have two many-to-many fields: groups and user_permissions. *User* objects can access their related objects in the same way as any other Django model:

```
myuser.groups.set([group_list])
myuser.groups.add(group, group, ...)
myuser.groups.remove(group, group, ...)
myuser.groups.clear()
myuser.user_permissions.set([permission_list])
myuser.user_permissions.add(permission, permission, ...)
myuser.user_permissions.remove(permission, permission, ...)
myuser.user_permissions.clear()
```

Default permissions

When `django.contrib.auth` is listed in your *`INSTALLED_APPS`* setting, it will ensure that three default permissions – add, change and delete – are created for each Django model defined in one of your installed applications.

These permissions will be created when you run *`manage.py migrate`*; the first time you run `migrate` after adding `django.contrib.auth` to *`INSTALLED_APPS`*, the default permissions will be created for all previously-installed models, as well as for any new models being installed at that time. Afterward, it will create default permissions for new models each time you run *`manage.py migrate`* (the function that creates permissions is connected to the *`post_migrate`* signal).

Assuming you have an application with an *`app_label`* `foo` and a model named `Bar`, to test for basic permissions you should use:

- add: `user.has_perm('foo.add_bar')`
- change: `user.has_perm('foo.change_bar')`
- delete: `user.has_perm('foo.delete_bar')`

The *`Permission`* model is rarely accessed directly.

Groups

`django.contrib.auth.models.Group` models are a generic way of categorizing users so you can apply permissions, or some other label, to those users. A user can belong to any number of groups.

A user in a group automatically has the permissions granted to that group. For example, if the group `Site editors` has the permission `can_edit_home_page`, any user in that group will have that permission.

Beyond permissions, groups are a convenient way to categorize users to give them some label, or extended functionality. For example, you could create a group `'Special users'`, and you could write code that could, say, give them access to a members-only portion of your site, or send them members-only email messages.

Programmatically creating permissions

While *custom permissions* can be defined within a model's `Meta` class, you can also create permissions directly. For example, you can create the `can_publish` permission for a `BlogPost` model in `myapp`:

```
from myapp.models import BlogPost
from django.contrib.auth.models import Permission
from django.contrib.contenttypes.models import ContentType

content_type = ContentType.objects.get_for_model(BlogPost)
permission = Permission.objects.create(
    codename='can_publish',
    name='Can Publish Posts',
    content_type=content_type,
)
```

The permission can then be assigned to a *`User`* via its `user_permissions` attribute or to a *`Group`* via its `permissions` attribute.

Permission caching

The *ModelBackend* caches permissions on the `User` object after the first time they need to be fetched for a permissions check. This is typically fine for the request-response cycle since permissions are not typically checked immediately after they are added (in the admin, for example). If you are adding permissions and checking them immediately afterward, in a test or view for example, the easiest solution is to re-fetch the `User` from the database. For example:

```
from django.contrib.auth.models import Permission, User
from django.shortcuts import get_object_or_404

def user_gains_perms(request, user_id):
    user = get_object_or_404(User, pk=user_id)
    # any permission check will cache the current set of permissions
    user.has_perm('myapp.change_bar')

    permission = Permission.objects.get(codename='change_bar')
    user.user_permissions.add(permission)

    # Checking the cached permission set
    user.has_perm('myapp.change_bar')  # False

    # Request new instance of User
    # Be aware that user.refresh_from_db() won't clear the cache.
    user = get_object_or_404(User, pk=user_id)

    # Permission cache is repopulated from the database
    user.has_perm('myapp.change_bar')  # True

    ...
```

Authentication in Web requests

Django uses sessions and middleware to hook the authentication system into *request objects*.

These provide a *request.user* attribute on every request which represents the current user. If the current user has not logged in, this attribute will be set to an instance of *AnonymousUser*, otherwise it will be an instance of *User*.

You can tell them apart with *is_authenticated*, like so:

```
if request.user.is_authenticated:
    # Do something for authenticated users.
    ...
else:
    # Do something for anonymous users.
    ...
```

How to log a user in

If you have an authenticated user you want to attach to the current session - this is done with a *login()* function.

login(*request*, *user*, *backend=None*)

To log a user in, from a view, use *login()*. It takes an *HttpRequest* object and a *User* object. *login()* saves the user's ID in the session, using Django's session framework.

Note that any data set during the anonymous session is retained in the session after a user logs in.

This example shows how you might use both *authenticate()* and *login()*:

```
from django.contrib.auth import authenticate, login

def my_view(request):
    username = request.POST['username']
    password = request.POST['password']
    user = authenticate(username=username, password=password)
    if user is not None:
        login(request, user)
        # Redirect to a success page.
        ...
    else:
        # Return an 'invalid login' error message.
        ...
```

In older versions, when you're manually logging a user in, you *must* successfully authenticate the user with *authenticate()* before you call *login()*. Now you can set the backend using the new `backend` argument.

Selecting the authentication backend When a user logs in, the user's ID and the backend that was used for authentication are saved in the user's session. This allows the same *authentication backend* to fetch the user's details on a future request. The authentication backend to save in the session is selected as follows:

1. Use the value of the optional `backend` argument, if provided.
2. Use the value of the `user.backend` attribute, if present. This allows pairing *authenticate()* and *login()*: *authenticate()* sets the `user.backend` attribute on the `User` object it returns.
3. Use the `backend` in *AUTHENTICATION_BACKENDS*, if there is only one.
4. Otherwise, raise an exception.

In cases 1 and 2, the value of the `backend` argument or the `user.backend` attribute should be a dotted import path string (like that found in *AUTHENTICATION_BACKENDS*), not the actual backend class.

How to log a user out

logout(*request*)

To log out a user who has been logged in via *django.contrib.auth.login()*, use *django.contrib.auth.logout()* within your view. It takes an *HttpRequest* object and has no return value. Example:

```
from django.contrib.auth import logout

def logout_view(request):
    logout(request)
    # Redirect to a success page.
```

Note that *logout()* doesn't throw any errors if the user wasn't logged in.

When you call *logout()*, the session data for the current request is completely cleaned out. All existing data is removed. This is to prevent another person from using the same Web browser to log in and have access to the previous user's session data. If you want to put anything into the session that will be available to the user immediately after logging out, do that *after* calling *django.contrib.auth.logout()*.

Limiting access to logged-in users

The raw way The simple, raw way to limit access to pages is to check *request.user.is_authenticated* and either redirect to a login page:

```
from django.conf import settings
from django.shortcuts import redirect

def my_view(request):
    if not request.user.is_authenticated:
        return redirect('%s?next=%s' % (settings.LOGIN_URL, request.path))
    # ...
```

...or display an error message:

```
from django.shortcuts import render

def my_view(request):
    if not request.user.is_authenticated:
        return render(request, 'myapp/login_error.html')
    # ...
```

The `login_required` decorator

login_required(*redirect_field_name='next'*, *login_url=None*)

As a shortcut, you can use the convenient *login_required()* decorator:

```
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request):
    ...
```

login_required() does the following:

- If the user isn't logged in, redirect to *settings.LOGIN_URL*, passing the current absolute path in the query string. Example: `/accounts/login/?next=/polls/3/`.
- If the user is logged in, execute the view normally. The view code is free to assume the user is logged in.

By default, the path that the user should be redirected to upon successful authentication is stored in a query string parameter called `"next"`. If you would prefer to use a different name for this parameter, *login_required()* takes an optional `redirect_field_name` parameter:

```
from django.contrib.auth.decorators import login_required

@login_required(redirect_field_name='my_redirect_field')
def my_view(request):
    ...
```

Note that if you provide a value to `redirect_field_name`, you will most likely need to customize your login template as well, since the template context variable which stores the redirect path will use the value of `redirect_field_name` as its key rather than `"next"` (the default).

login_required() also takes an optional `login_url` parameter. Example:

```
from django.contrib.auth.decorators import login_required

@login_required(login_url='/accounts/login/')
def my_view(request):
    ...
```

Note that if you don't specify the `login_url` parameter, you'll need to ensure that the *`settings.LOGIN_URL`* and your login view are properly associated. For example, using the defaults, add the following lines to your URLconf:

```
from django.contrib.auth import views as auth_views

url(r'^accounts/login/$', auth_views.login),
```

The *`settings.LOGIN_URL`* also accepts view function names and *named URL patterns*. This allows you to freely remap your login view within your URLconf without having to update the setting.

: The `login_required` decorator does NOT check the `is_active` flag on a user, but the default *`AUTHENTICATION_BACKENDS`* reject inactive users.

:

If you are writing custom views for Django's admin (or need the same authorization check that the built-in views use), you may find the *`django.contrib.admin.views.decorators.staff_member_required()`* decorator a useful alternative to `login_required()`.

The `LoginRequired` mixin When using class-based views, you can achieve the same behavior as with `login_required` by using the `LoginRequiredMixin`. This mixin should be at the leftmost position in the inheritance list.

class `LoginRequiredMixin`

If a view is using this mixin, all requests by non-authenticated users will be redirected to the login page or shown an HTTP 403 Forbidden error, depending on the *`raise_exception`* parameter.

You can set any of the parameters of *`AccessMixin`* to customize the handling of unauthorized users:

```
from django.contrib.auth.mixins import LoginRequiredMixin

class MyView(LoginRequiredMixin, View):
    login_url = '/login/'
    redirect_field_name = 'redirect_to'
```

: Just as the `login_required` decorator, this mixin does NOT check the `is_active` flag on a user, but the default *`AUTHENTICATION_BACKENDS`* reject inactive users.

Limiting access to logged-in users that pass a test To limit access based on certain permissions or some other test, you'd do essentially the same thing as described in the previous section.

The simple way is to run your test on *`request.user`* in the view directly. For example, this view checks to make sure the user has an email in the desired domain and if not, redirects to the login page:

```
from django.shortcuts import redirect

def my_view(request):
    if not request.user.email.endswith('@example.com'):
        return redirect('/login/?next=%s' % request.path)
    # ...
```

`user_passes_test`(*test_func*, *login_url=None*, *redirect_field_name='next'*)

As a shortcut, you can use the convenient `user_passes_test` decorator which performs a redirect when the callable returns `False`:

```
from django.contrib.auth.decorators import user_passes_test

def email_check(user):
    return user.email.endswith('@example.com')

@user_passes_test(email_check)
def my_view(request):
    ...
```

user_passes_test() takes a required argument: a callable that takes a *User* object and returns `True` if the user is allowed to view the page. Note that *user_passes_test()* does not automatically check that the *User* is not anonymous.

user_passes_test() takes two optional arguments:

login_url Lets you specify the URL that users who don't pass the test will be redirected to. It may be a login page and defaults to *settings.LOGIN_URL* if you don't specify one.

redirect_field_name Same as for *login_required()*. Setting it to `None` removes it from the URL, which you may want to do if you are redirecting users that don't pass the test to a non-login page where there's no "next page".

For example:

```
@user_passes_test(email_check, login_url='/login/')
def my_view(request):
    ...
```

class UserPassesTestMixin

When using class-based views, you can use the `UserPassesTestMixin` to do this.

test_func()

You have to override the `test_func()` method of the class to provide the test that is performed. Furthermore, you can set any of the parameters of *AccessMixin* to customize the handling of unauthorized users:

```
from django.contrib.auth.mixins import UserPassesTestMixin

class MyView(UserPassesTestMixin, View):

    def test_func(self):
        return self.request.user.email.endswith('@example.com')
```

get_test_func()

You can also override the `get_test_func()` method to have the mixin use a differently named function for its checks (instead of *test_func()*).

Stacking UserPassesTestMixin

Due to the way `UserPassesTestMixin` is implemented, you cannot stack them in your inheritance list. The following does NOT work:

```
class TestMixin1(UserPassesTestMixin):
    def test_func(self):
        return self.request.user.email.endswith('@example.com')

class TestMixin2(UserPassesTestMixin):
    def test_func(self):
        return self.request.user.username.startswith('django')
```

```
class MyView(TestMixin1, TestMixin2, View):
    ...
```

If `TestMixin1` would call `super()` and take that result into account, `TestMixin1` wouldn't work standalone anymore.

The `permission_required` decorator

`permission_required`(*perm*, *login_url=None*, *raise_exception=False*)

It's a relatively common task to check whether a user has a particular permission. For that reason, Django provides a shortcut for that case: the *`permission_required()`* decorator.:

```
from django.contrib.auth.decorators import permission_required

@permission_required('polls.can_vote')
def my_view(request):
    ...
```

Just like the *`has_perm()`* method, permission names take the form "`<app label>.<permission codename>`" (i.e. `polls.can_vote` for a permission on a model in the `polls` application).

The decorator may also take an iterable of permissions, in which case the user must have all of the permissions in order to access the view.

Note that *`permission_required()`* also takes an optional `login_url` parameter:

```
from django.contrib.auth.decorators import permission_required

@permission_required('polls.can_vote', login_url='/loginpage/')
def my_view(request):
    ...
```

As in the *`login_required()`* decorator, `login_url` defaults to *`settings.LOGIN_URL`*.

If the `raise_exception` parameter is given, the decorator will raise *`PermissionDenied`*, prompting *the 403 (HTTP Forbidden) view* instead of redirecting to the login page.

If you want to use `raise_exception` but also give your users a chance to login first, you can add the *`login_required()`* decorator:

```
from django.contrib.auth.decorators import login_required, permission_required

@login_required
@permission_required('polls.can_vote', raise_exception=True)
def my_view(request):
    ...
```

In older versions, the `permission` parameter only worked with strings, lists, and tuples instead of strings and any iterable.

The `PermissionRequiredMixin` mixin To apply permission checks to class-based views, you can use the `PermissionRequiredMixin`:

class `PermissionRequiredMixin`

This mixin, just like the `permission_required` decorator, checks whether the user accessing a view has all given permissions. You should specify the permission (or an iterable of permissions) using the `permission_required` parameter:

```
from django.contrib.auth.mixins import PermissionRequiredMixin

class MyView(PermissionRequiredMixin, View):
    permission_required = 'polls.can_vote'
    # Or multiple of permissions:
    permission_required = ('polls.can_open', 'polls.can_edit')
```

You can set any of the parameters of *AccessMixin* to customize the handling of unauthorized users.

You may also override these methods:

get_permission_required()
Returns an iterable of permission names used by the mixin. Defaults to the permission_required attribute, converted to a tuple if necessary.

has_permission()
Returns a boolean denoting whether the current user has permission to execute the decorated view. By default, this returns the result of calling *has_perms()* with the list of permissions returned by *get_permission_required()*.

Redirecting unauthorized requests in class-based views

To ease the handling of access restrictions in class-based views, the AccessMixin can be used to redirect a user to the login page or issue an HTTP 403 Forbidden response.

class AccessMixin

login_url
Default return value for *get_login_url()*. Defaults to None in which case *get_login_url()* falls back to *settings.LOGIN_URL*.

permission_denied_message
Default return value for *get_permission_denied_message()*. Defaults to an empty string.

redirect_field_name
Default return value for *get_redirect_field_name()*. Defaults to "next".

raise_exception
If this attribute is set to True, a *PermissionDenied* exception will be raised instead of the redirect. Defaults to False.

get_login_url()
Returns the URL that users who don't pass the test will be redirected to. Returns *login_url* if set, or *settings.LOGIN_URL* otherwise.

get_permission_denied_message()
When *raise_exception* is True, this method can be used to control the error message passed to the error handler for display to the user. Returns the *permission_denied_message* attribute by default.

get_redirect_field_name()
Returns the name of the query parameter that will contain the URL the user should be redirected to after a successful login. If you set this to None, a query parameter won't be added. Returns the *redirect_field_name* attribute by default.

handle_no_permission()
Depending on the value of raise_exception, the method either raises a *PermissionDenied* exception or redirects the user to the login_url, optionally including the redirect_field_name if it is set.

Session invalidation on password change Session verification is enabled and mandatory in Django 1.10 (there's no way to disable it) regardless of whether or not `SessionAuthenticationMiddleware` is enabled. In older versions, this protection only applies if `django.contrib.auth.middleware.SessionAuthenticationMiddleware` is enabled in *MIDDLEWARE*.

If your *AUTH_USER_MODEL* inherits from *AbstractBaseUser* or implements its own *get_session_auth_hash()* method, authenticated sessions will include the hash returned by this function. In the *AbstractBaseUser* case, this is an HMAC of the password field. Django verifies that the hash in the session for each request matches the one that's computed during the request. This allows a user to log out all of their sessions by changing their password.

The default password change views included with Django, *password_change()* and the `user_change_password` view in the *django.contrib.auth* admin, update the session with the new password hash so that a user changing their own password won't log themselves out. If you have a custom password change view and wish to have similar behavior, use the *update_session_auth_hash()* function. In this case, however, if a user also wants to invalidate the session from which they're changing their password (for example, if they believe the session cookie on their machine was stolen), then they also need to log out that session.

update_session_auth_hash(*request*, *user*)

This function takes the current request and the updated user object from which the new session hash will be derived and updates the session hash appropriately. Example usage:

```
from django.contrib.auth import update_session_auth_hash

def password_change(request):
    if request.method == 'POST':
        form = PasswordChangeForm(user=request.user, data=request.POST)
        if form.is_valid():
            form.save()
            update_session_auth_hash(request, form.user)
    else:
        ...
```

: Since *get_session_auth_hash()* is based on *SECRET_KEY*, updating your site to use a new secret will invalidate all existing sessions.

Authentication Views

Django provides several views that you can use for handling login, logout, and password management. These make use of the *stock auth forms* but you can pass in your own forms as well.

Django provides no default template for the authentication views. You should create your own templates for the views you want to use. The template context is documented in each view, see *All authentication views*.

Using the views There are different methods to implement these views in your project. The easiest way is to include the provided URLconf in `django.contrib.auth.urls` in your own URLconf, for example:

```
urlpatterns = [
    url('^', include('django.contrib.auth.urls')),
]
```

This will include the following URL patterns:

```
^login/$ [name='login']
^logout/$ [name='logout']
^password_change/$ [name='password_change']
^password_change/done/$ [name='password_change_done']
^password_reset/$ [name='password_reset']
^password_reset/done/$ [name='password_reset_done']
^reset/(?P<uidb64>[0-9A-Za-z_\-]+)/(?P<token>[0-9A-Za-z]{1,13}-[0-9A-Za-z]{1,20})/$ [name='password_
^reset/done/$ [name='password_reset_complete']
```

The views provide a URL name for easier reference. See the URL documentation for details on using named URL patterns.

If you want more control over your URLs, you can reference a specific view in your URLconf:

```
from django.contrib.auth import views as auth_views

urlpatterns = [
    url('^change-password/$', auth_views.password_change),
]
```

The views have optional arguments you can use to alter the behavior of the view. For example, if you want to change the template name a view uses, you can provide the `template_name` argument. A way to do this is to provide keyword arguments in the URLconf, these will be passed on to the view. For example:

```
urlpatterns = [
    url(
        '^change-password/$',
        auth_views.password_change,
        {'template_name': 'change-password.html'}
    ),
]
```

All views return a *TemplateResponse* instance, which allows you to easily customize the response data before rendering. A way to do this is to wrap a view in your own view:

```
from django.contrib.auth import views

def change_password(request):
    template_response = views.password_change(request)
    # Do something with `template_response`
    return template_response
```

For more details, see the TemplateResponse documentation.

All authentication views This is a list with all the views `django.contrib.auth` provides. For implementation details see *Using the views*.

login(*request*, *template_name='registration/login.html'*, *redirect_field_name='next'*, *authentication_form=AuthenticationForm*, *current_app=None*, *extra_context=None*, *redirect_authenticated_user=False*)

URL name: `login`

See the URL documentation for details on using named URL patterns.

Optional arguments:

- `template_name`: The name of a template to display for the view used to log the user in. Defaults to `registration/login.html`.

- `redirect_field_name`: The name of a `GET` field containing the URL to redirect to after login. Defaults to `next`.
- `authentication_form`: A callable (typically just a form class) to use for authentication. Defaults to *`AuthenticationForm`*.
- `current_app`: A hint indicating which application contains the current view. See the *namespaced URL resolution strategy* for more information.
- `extra_context`: A dictionary of context data that will be added to the default context data passed to the template.
- `redirect_authenticated_user`: A boolean that controls whether or not authenticated users accessing the login page will be redirected as if they had just successfully logged in. Defaults to `False`.

> **:** If you enable `redirect_authenticated_user`, other websites will be able to determine if their visitors are authenticated on your site by requesting redirect URLs to image files on your website. To avoid this "social media fingerprinting" information leakage, host all images and your favicon on a separate domain.

1.9 : The `current_app` parameter is deprecated and will be removed in Django 2.0. Callers should set `request.current_app` instead.

The `redirect_authenticated_user` parameter was added.

Here's what `django.contrib.auth.views.login` does:

- If called via `GET`, it displays a login form that POSTs to the same URL. More on this in a bit.
- If called via `POST` with user submitted credentials, it tries to log the user in. If login is successful, the view redirects to the URL specified in `next`. If `next` isn't provided, it redirects to *`settings.LOGIN_REDIRECT_URL`* (which defaults to `/accounts/profile/`). If login isn't successful, it redisplays the login form.

It's your responsibility to provide the html for the login template , called `registration/login.html` by default. This template gets passed four template context variables:

- `form`: A *`Form`* object representing the *`AuthenticationForm`*.
- `next`: The URL to redirect to after successful login. This may contain a query string, too.
- `site`: The current *`Site`*, according to the *`SITE_ID`* setting. If you don't have the site framework installed, this will be set to an instance of *`RequestSite`*, which derives the site name and domain from the current *`HttpRequest`*.
- `site_name`: An alias for `site.name`. If you don't have the site framework installed, this will be set to the value of *`request.META['SERVER_NAME']`*. For more on sites, see The "sites" framework.

If you'd prefer not to call the template `registration/login.html`, you can pass the `template_name` parameter via the extra arguments to the view in your URLconf. For example, this URLconf line would use `myapp/login.html` instead:

```
url(r'^accounts/login/$', auth_views.login, {'template_name': 'myapp/login.html'}),
```

You can also specify the name of the `GET` field which contains the URL to redirect to after login by passing `redirect_field_name` to the view. By default, the field is called `next`.

Here's a sample `registration/login.html` template you can use as a starting point. It assumes you have a `base.html` template that defines a `content` block:

```
{% extends "base.html" %}

{% block content %}

{% if form.errors %}
<p>Your username and password didn't match. Please try again.</p>
{% endif %}

{% if next %}
    {% if user.is_authenticated %}
    <p>Your account doesn't have access to this page. To proceed,
    please login with an account that has access.</p>
    {% else %}
    <p>Please login to see this page.</p>
    {% endif %}
{% endif %}

<form method="post" action="{% url 'login' %}">
{% csrf_token %}
<table>
<tr>
    <td>{{ form.username.label_tag }}</td>
    <td>{{ form.username }}</td>
</tr>
<tr>
    <td>{{ form.password.label_tag }}</td>
    <td>{{ form.password }}</td>
</tr>
</table>

<input type="submit" value="login" />
<input type="hidden" name="next" value="{{ next }}" />
</form>

{# Assumes you setup the password_reset view in your URLconf #}
<p><a href="{% url 'password_reset' %}">Lost password?</a></p>

{% endblock %}
```

If you have customized authentication (see Customizing Authentication) you can pass a custom authentication form to the login view via the `authentication_form` parameter. This form must accept a `request` keyword argument in its `__init__` method, and provide a `get_user()` method which returns the authenticated user object (this method is only ever called after successful form validation).

logout(*request*, *next_page=None*, *template_name='registration/logged_out.html'*, *redirect_field_name='next'*, *current_app=None*, *extra_context=None*)

Logs a user out.

URL name: `logout`

Optional arguments:

- `next_page`: The URL to redirect to after logout. Defaults to *`settings.LOGOUT_REDIRECT_URL`* if not supplied.
- `template_name`: The full name of a template to display after logging the user out. Defaults to `registration/logged_out.html` if no argument is supplied.
- `redirect_field_name`: The name of a `GET` field containing the URL to redirect to after log out. Defaults to `next`. Overrides the `next_page` URL if the given `GET` parameter is passed.

- `current_app`: A hint indicating which application contains the current view. See the *namespaced URL resolution strategy* for more information.
- `extra_context`: A dictionary of context data that will be added to the default context data passed to the template.

1.9 : The `current_app` parameter is deprecated and will be removed in Django 2.0. Callers should set `request.current_app` instead.

Template context:

- `title`: The string "Logged out", localized.
- `site`: The current *`Site`*, according to the *`SITE_ID`* setting. If you don't have the site framework installed, this will be set to an instance of *`RequestSite`*, which derives the site name and domain from the current *`HttpRequest`*.
- `site_name`: An alias for `site.name`. If you don't have the site framework installed, this will be set to the value of *`request.META['SERVER_NAME']`*. For more on sites, see The "sites" framework.
- `current_app`: A hint indicating which application contains the current view. See the *namespaced URL resolution strategy* for more information.
- `extra_context`: A dictionary of context data that will be added to the default context data passed to the template.

`logout_then_login`(*request*, *login_url=None*, *current_app=None*, *extra_context=None*)

Logs a user out, then redirects to the login page.

URL name: No default URL provided

Optional arguments:

- `login_url`: The URL of the login page to redirect to. Defaults to *`settings.LOGIN_URL`* if not supplied.
- `current_app`: A hint indicating which application contains the current view. See the *namespaced URL resolution strategy* for more information.
- `extra_context`: A dictionary of context data that will be added to the default context data passed to the template.

1.9 : The `current_app` parameter is deprecated and will be removed in Django 2.0. Callers should set `request.current_app` instead.

`password_change`(*request*, *template_name='registration/password_change_form.html'*, *post_change_redirect=None*, *password_change_form=PasswordChangeForm*, *current_app=None*, *extra_context=None*)

Allows a user to change their password.

URL name: `password_change`

Optional arguments:

- `template_name`: The full name of a template to use for displaying the password change form. Defaults to `registration/password_change_form.html` if not supplied.
- `post_change_redirect`: The URL to redirect to after a successful password change.
- `password_change_form`: A custom "change password" form which must accept a `user` keyword argument. The form is responsible for actually changing the user's password. Defaults to *`PasswordChangeForm`*.
- `current_app`: A hint indicating which application contains the current view. See the *namespaced URL resolution strategy* for more information.

- `extra_context`: A dictionary of context data that will be added to the default context data passed to the template.

1.9 : The `current_app` parameter is deprecated and will be removed in Django 2.0. Callers should set `request.current_app` instead.

Template context:

- `form`: The password change form (see `password_change_form` above).

password_change_done(*request*, *template_name='registration/password_change_done.html'*, *current_app=None*, *extra_context=None*)

The page shown after a user has changed their password.

URL name: `password_change_done`

Optional arguments:

- `template_name`: The full name of a template to use. Defaults to `registration/password_change_done.html` if not supplied.
- `current_app`: A hint indicating which application contains the current view. See the *namespaced URL resolution strategy* for more information.
- `extra_context`: A dictionary of context data that will be added to the default context data passed to the template.

1.9 : The `current_app` parameter is deprecated and will be removed in Django 2.0. Callers should set `request.current_app` instead.

password_reset(*request*, *template_name='registration/password_reset_form.html'*, *email_template_name='registration/password_reset_email.html'*, *subject_template_name='registration/password_reset_subject.txt'*, *password_reset_form=PasswordResetForm*, *token_generator=default_token_generator*, *post_reset_redirect=None*, *from_email=None*, *current_app=None*, *extra_context=None*, *html_email_template_name=None*, *extra_email_context=None*)

Allows a user to reset their password by generating a one-time use link that can be used to reset the password, and sending that link to the user's registered email address.

If the email address provided does not exist in the system, this view won't send an email, but the user won't receive any error message either. This prevents information leaking to potential attackers. If you want to provide an error message in this case, you can subclass *PasswordResetForm* and use the `password_reset_form` argument.

Users flagged with an unusable password (see *set_unusable_password()* aren't allowed to request a password reset to prevent misuse when using an external authentication source like LDAP. Note that they won't receive any error message since this would expose their account's existence but no mail will be sent either.

URL name: `password_reset`

Optional arguments:

- `template_name`: The full name of a template to use for displaying the password reset form. Defaults to `registration/password_reset_form.html` if not supplied.
- `email_template_name`: The full name of a template to use for generating the email with the reset password link. Defaults to `registration/password_reset_email.html` if not supplied.
- `subject_template_name`: The full name of a template to use for the subject of the email with the reset password link. Defaults to `registration/password_reset_subject.txt` if not supplied.
- `password_reset_form`: Form that will be used to get the email of the user to reset the password for. Defaults to *PasswordResetForm*.

- `token_generator`: Instance of the class to check the one time link. This will default to `default_token_generator`, it's an instance of `django.contrib.auth.tokens.PasswordResetTokenGenerator`.
- `post_reset_redirect`: The URL to redirect to after a successful password reset request.
- `from_email`: A valid email address. By default Django uses the *DEFAULT_FROM_EMAIL*.
- `current_app`: A hint indicating which application contains the current view. See the *namespaced URL resolution strategy* for more information.
- `extra_context`: A dictionary of context data that will be added to the default context data passed to the template.
- `html_email_template_name`: The full name of a template to use for generating a `text/html` multipart email with the password reset link. By default, HTML email is not sent.
- `extra_email_context`: A dictionary of context data that will be available in the email template.

1.9 : The `current_app` parameter is deprecated and will be removed in Django 2.0. Callers should set `request.current_app` instead.

The `extra_email_context` parameter was added.

Template context:

- `form`: The form (see `password_reset_form` above) for resetting the user's password.

Email template context:

- `email`: An alias for `user.email`
- `user`: The current *User*, according to the `email` form field. Only active users are able to reset their passwords (`User.is_active is True`).
- `site_name`: An alias for `site.name`. If you don't have the site framework installed, this will be set to the value of *request.META['SERVER_NAME']*. For more on sites, see The "sites" framework.
- `domain`: An alias for `site.domain`. If you don't have the site framework installed, this will be set to the value of `request.get_host()`.
- `protocol`: http or https
- `uid`: The user's primary key encoded in base 64.
- `token`: Token to check that the reset link is valid.

Sample `registration/password_reset_email.html` (email body template):

```
Someone asked for password reset for email {{ email }}. Follow the link below:
{{ protocol}}://{{ domain }}{% url 'password_reset_confirm' uidb64=uid token=token %}
```

The same template context is used for subject template. Subject must be single line plain text string.

password_reset_done(*request*, *template_name='registration/password_reset_done.html'*, *current_app=None*, *extra_context=None*)

The page shown after a user has been emailed a link to reset their password. This view is called by default if the *password_reset()* view doesn't have an explicit `post_reset_redirect` URL set.

URL name: `password_reset_done`

: If the email address provided does not exist in the system, the user is inactive, or has an unusable password, the user will still be redirected to this view but no email will be sent.

Optional arguments:

- `template_name`: The full name of a template to use. Defaults to `registration/password_reset_done.html` if not supplied.
- `current_app`: A hint indicating which application contains the current view. See the *namespaced URL resolution strategy* for more information.
- `extra_context`: A dictionary of context data that will be added to the default context data passed to the template.

1.9 : The `current_app` parameter is deprecated and will be removed in Django 2.0. Callers should set `request.current_app` instead.

`password_reset_confirm`(*request*, *uidb64=None*, *token=None*, *template_name='registration/password_reset_confirm.html'*, *token_generator=default_token_generator*, *set_password_form=SetPasswordForm*, *post_reset_redirect=None*, *current_app=None*, *extra_context=None*)

Presents a form for entering a new password.

URL name: `password_reset_confirm`

Optional arguments:

- `uidb64`: The user's id encoded in base 64. Defaults to `None`.
- `token`: Token to check that the password is valid. Defaults to `None`.
- `template_name`: The full name of a template to display the confirm password view. Default value is `registration/password_reset_confirm.html`.
- `token_generator`: Instance of the class to check the password. This will default to `default_token_generator`, it's an instance of `django.contrib.auth.tokens.PasswordResetTokenGenerator`.
- `set_password_form`: Form that will be used to set the password. Defaults to *`SetPasswordForm`*
- `post_reset_redirect`: URL to redirect after the password reset done. Defaults to `None`.
- `current_app`: A hint indicating which application contains the current view. See the *namespaced URL resolution strategy* for more information.
- `extra_context`: A dictionary of context data that will be added to the default context data passed to the template.

Template context:

- `form`: The form (see `set_password_form` above) for setting the new user's password.
- `validlink`: Boolean, True if the link (combination of `uidb64` and `token`) is valid or unused yet.

1.9 : The `current_app` parameter is deprecated and will be removed in Django 2.0. Callers should set `request.current_app` instead.

`password_reset_complete`(*request*, *template_name='registration/password_reset_complete.html'*, *current_app=None*, *extra_context=None*)

Presents a view which informs the user that the password has been successfully changed.

URL name: `password_reset_complete`

Optional arguments:

- `template_name`: The full name of a template to display the view. Defaults to `registration/password_reset_complete.html`.

- `current_app`: A hint indicating which application contains the current view. See the *namespaced URL resolution strategy* for more information.
- `extra_context`: A dictionary of context data that will be added to the default context data passed to the template.

1.9 : The `current_app` parameter is deprecated and will be removed in Django 2.0. Callers should set `request.current_app` instead.

Helper functions

redirect_to_login(*next*, *login_url=None*, *redirect_field_name='next'*)

Redirects to the login page, and then back to another URL after a successful login.

Required arguments:

- `next`: The URL to redirect to after a successful login.

Optional arguments:

- `login_url`: The URL of the login page to redirect to. Defaults to *`settings.LOGIN_URL`* if not supplied.
- `redirect_field_name`: The name of a `GET` field containing the URL to redirect to after log out. Overrides `next` if the given `GET` parameter is passed.

Built-in forms

If you don't want to use the built-in views, but want the convenience of not having to write forms for this functionality, the authentication system provides several built-in forms located in *`django.contrib.auth.forms`*:

: The built-in authentication forms make certain assumptions about the user model that they are working with. If you're using a *custom User model*, it may be necessary to define your own forms for the authentication system. For more information, refer to the documentation about *using the built-in authentication forms with custom user models*.

---

class AdminPasswordChangeForm

A form used in the admin interface to change a user's password.

Takes the `user` as the first positional argument.

class AuthenticationForm

A form for logging a user in.

Takes `request` as its first positional argument, which is stored on the form instance for use by sub-classes.

confirm_login_allowed(*user*)

By default, `AuthenticationForm` rejects users whose `is_active` flag is set to `False`. You may override this behavior with a custom policy to determine which users can log in. Do this with a custom form that subclasses `AuthenticationForm` and overrides the `confirm_login_allowed()` method. This method should raise a *`ValidationError`* if the given user may not log in.

For example, to allow all users to log in regardless of "active" status:

```
from django.contrib.auth.forms import AuthenticationForm

class AuthenticationFormWithInactiveUsersOkay(AuthenticationForm):
    def confirm_login_allowed(self, user):
        pass
```

(In this case, you'll also need to use an authentication backend that allows inactive users, such as as *AllowAllUsersModelBackend*.)

Or to allow only some active users to log in:

```
class PickyAuthenticationForm(AuthenticationForm):
    def confirm_login_allowed(self, user):
        if not user.is_active:
            raise forms.ValidationError(
                _("This account is inactive."),
                code='inactive',
            )
        if user.username.startswith('b'):
            raise forms.ValidationError(
                _("Sorry, accounts starting with 'b' aren't welcome here."),
                code='no_b_users',
            )
```

class PasswordChangeForm

A form for allowing a user to change their password.

class PasswordResetForm

A form for generating and emailing a one-time use link to reset a user's password.

send_email(*subject_template_name*, *email_template_name*, *context*, *from_email*, *to_email*, *html_email_template_name=None*)

Uses the arguments to send an `EmailMultiAlternatives`. Can be overridden to customize how the email is sent to the user.

- **subject_template_name** – the template for the subject.
- **email_template_name** – the template for the email body.
- **context** – context passed to the `subject_template`, `email_template`, and `html_email_template` (if it is not `None`).
- **from_email** – the sender's email.
- **to_email** – the email of the requester.
- **html_email_template_name** – the template for the HTML body; defaults to `None`, in which case a plain text email is sent.

By default, `save()` populates the `context` with the same variables that *password_reset()* passes to its email context.

class SetPasswordForm

A form that lets a user change their password without entering the old password.

class UserChangeForm

A form used in the admin interface to change a user's information and permissions.

class UserCreationForm

A *ModelForm* for creating a new user.

It has three fields: `username` (from the user model), `password1`, and `password2`. It verifies that `password1` and `password2` match, validates the password using *validate_password()*, and sets the user's password using *set_password()*.

Authentication data in templates

The currently logged-in user and their permissions are made available in the template context when you use *RequestContext*.

Technicality

Technically, these variables are only made available in the template context if you use *RequestContext* and the `'django.contrib.auth.context_processors.auth'` context processor is enabled. It is in the default generated settings file. For more, see the *RequestContext docs*.

Users When rendering a template *RequestContext*, the currently logged-in user, either a *User* instance or an *AnonymousUser* instance, is stored in the template variable `{{ user }}`:

```
{% if user.is_authenticated %}
    <p>Welcome, {{ user.username }}. Thanks for logging in.</p>
{% else %}
    <p>Welcome, new user. Please log in.</p>
{% endif %}
```

This template context variable is not available if a `RequestContext` is not being used.

Permissions The currently logged-in user's permissions are stored in the template variable `{{ perms }}`. This is an instance of `django.contrib.auth.context_processors.PermWrapper`, which is a template-friendly proxy of permissions.

In the `{{ perms }}` object, single-attribute lookup is a proxy to *User.has_module_perms*. This example would display `True` if the logged-in user had any permissions in the `foo` app:

```
{{ perms.foo }}
```

Two-level-attribute lookup is a proxy to *User.has_perm*. This example would display `True` if the logged-in user had the permission `foo.can_vote`:

```
{{ perms.foo.can_vote }}
```

Thus, you can check permissions in template `{% if %}` statements:

```
{% if perms.foo %}
    <p>You have permission to do something in the foo app.</p>
    {% if perms.foo.can_vote %}
        <p>You can vote!</p>
    {% endif %}
    {% if perms.foo.can_drive %}
        <p>You can drive!</p>
    {% endif %}
{% else %}
    <p>You don't have permission to do anything in the foo app.</p>
{% endif %}
```

It is possible to also look permissions up by `{% if in %}` statements. For example:

```
{% if 'foo' in perms %}
    {% if 'foo.can_vote' in perms %}
        <p>In lookup works, too.</p>
```

```
    {% endif %}
{% endif %}
```

Managing users in the admin

When you have both `django.contrib.admin` and `django.contrib.auth` installed, the admin provides a convenient way to view and manage users, groups, and permissions. Users can be created and deleted like any Django model. Groups can be created, and permissions can be assigned to users or groups. A log of user edits to models made within the admin is also stored and displayed.

Creating users

You should see a link to "Users" in the "Auth" section of the main admin index page. The "Add user" admin page is different than standard admin pages in that it requires you to choose a username and password before allowing you to edit the rest of the user's fields.

Also note: if you want a user account to be able to create users using the Django admin site, you'll need to give them permission to add users *and* change users (i.e., the "Add user" and "Change user" permissions). If an account has permission to add users but not to change them, that account won't be able to add users. Why? Because if you have permission to add users, you have the power to create superusers, which can then, in turn, change other users. So Django requires add *and* change permissions as a slight security measure.

Be thoughtful about how you allow users to manage permissions. If you give a non-superuser the ability to edit users, this is ultimately the same as giving them superuser status because they will be able to elevate permissions of users including themselves!

Changing passwords

User passwords are not displayed in the admin (nor stored in the database), but the password storage details are displayed. Included in the display of this information is a link to a password change form that allows admins to change user passwords.

3.10.2 Password management in Django

Password management is something that should generally not be reinvented unnecessarily, and Django endeavors to provide a secure and flexible set of tools for managing user passwords. This document describes how Django stores passwords, how the storage hashing can be configured, and some utilities to work with hashed passwords.

:

Even though users may use strong passwords, attackers might be able to eavesdrop on their connections. Use *HTTPS* to avoid sending passwords (or any other sensitive data) over plain HTTP connections because they will be vulnerable to password sniffing.

How Django stores passwords

Django provides a flexible password storage system and uses PBKDF2 by default.

The *password* attribute of a *User* object is a string in this format:

```
<algorithm>$<iterations>$<salt>$<hash>
```

Those are the components used for storing a User's password, separated by the dollar-sign character and consist of: the hashing algorithm, the number of algorithm iterations (work factor), the random salt, and the resulting password hash. The algorithm is one of a number of one-way hashing or password storage algorithms Django can use; see below. Iterations describe the number of times the algorithm is run over the hash. Salt is the random seed used and the hash is the result of the one-way function.

By default, Django uses the PBKDF2 algorithm with a SHA256 hash, a password stretching mechanism recommended by NIST. This should be sufficient for most users: it's quite secure, requiring massive amounts of computing time to break.

However, depending on your requirements, you may choose a different algorithm, or even use a custom algorithm to match your specific security situation. Again, most users shouldn't need to do this – if you're not sure, you probably don't. If you do, please read on:

Django chooses the algorithm to use by consulting the *PASSWORD_HASHERS* setting. This is a list of hashing algorithm classes that this Django installation supports. The first entry in this list (that is, `settings.PASSWORD_HASHERS[0]`) will be used to store passwords, and all the other entries are valid hashers that can be used to check existing passwords. This means that if you want to use a different algorithm, you'll need to modify *PASSWORD_HASHERS* to list your preferred algorithm first in the list.

The default for *PASSWORD_HASHERS* is:

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
    'django.contrib.auth.hashers.Argon2PasswordHasher',
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
    'django.contrib.auth.hashers.BCryptPasswordHasher',
]
```

This means that Django will use PBKDF2 to store all passwords but will support checking passwords stored with PBKDF2SHA1, argon2, and bcrypt.

The next few sections describe a couple of common ways advanced users may want to modify this setting.

Using Argon2 with Django

Argon2 is the winner of the 2015 Password Hashing Competition, a community organized open competition to select a next generation hashing algorithm. It's designed not to be easier to compute on custom hardware than it is to compute on an ordinary CPU.

Argon2 is not the default for Django because it requires a third-party library. The Password Hashing Competition panel, however, recommends immediate use of Argon2 rather than the other algorithms supported by Django.

To use Argon2 as your default storage algorithm, do the following:

1. Install the argon2-cffi library. This can be done by running `pip install django[argon2]`, which is equivalent to `pip install argon2-cffi` (along with any version requirement from Django's `setup.py`).
2. Modify *PASSWORD_HASHERS* to list `Argon2PasswordHasher` first. That is, in your settings file, you'd put:

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.Argon2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
```

```
        'django.contrib.auth.hashers.BCryptPasswordHasher',
    ]
```

Keep and/or add any entries in this list if you need Django to *upgrade passwords*.

Using `bcrypt` with Django

Bcrypt is a popular password storage algorithm that's specifically designed for long-term password storage. It's not the default used by Django since it requires the use of third-party libraries, but since many people may want to use it Django supports bcrypt with minimal effort.

To use Bcrypt as your default storage algorithm, do the following:

1. Install the bcrypt library. This can be done by running `pip install django[bcrypt]`, which is equivalent to `pip install bcrypt` (along with any version requirement from Django's `setup.py`).
2. Modify *`PASSWORD_HASHERS`* to list `BCryptSHA256PasswordHasher` first. That is, in your settings file, you'd put:

```
    PASSWORD_HASHERS = [
        'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
        'django.contrib.auth.hashers.BCryptPasswordHasher',
        'django.contrib.auth.hashers.PBKDF2PasswordHasher',
        'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
        'django.contrib.auth.hashers.Argon2PasswordHasher',
    ]
```

Keep and/or add any entries in this list if you need Django to *upgrade passwords*.

That's it – now your Django install will use Bcrypt as the default storage algorithm.

Password truncation with BCryptPasswordHasher

The designers of bcrypt truncate all passwords at 72 characters which means that `bcrypt(password_with_100_chars) == bcrypt(password_with_100_chars[:72])`. The original `BCryptPasswordHasher` does not have any special handling and thus is also subject to this hidden password length limit. `BCryptSHA256PasswordHasher` fixes this by first hashing the password using sha256. This prevents the password truncation and so should be preferred over the `BCryptPasswordHasher`. The practical ramification of this truncation is pretty marginal as the average user does not have a password greater than 72 characters in length and even being truncated at 72 the compute powered required to brute force bcrypt in any useful amount of time is still astronomical. Nonetheless, we recommend you use `BCryptSHA256PasswordHasher` anyway on the principle of "better safe than sorry".

Other bcrypt implementations

There are several other implementations that allow bcrypt to be used with Django. Django's bcrypt support is NOT directly compatible with these. To upgrade, you will need to modify the hashes in your database to be in the form `bcrypt$(raw bcrypt output)`. For example: `bcrypt$$2a$12$NT0I31Sa7ihGEWpka9ASYrEFkhuTNeBQ2xfZskIiiJeyFXhRgS.Sy`.

Increasing the work factor

PBKDF2 and bcrypt The PBKDF2 and bcrypt algorithms use a number of iterations or rounds of hashing. This deliberately slows down attackers, making attacks against hashed passwords harder. However, as computing power increases, the number of iterations needs to be increased. We've chosen a reasonable default (and will increase it with each release of Django), but you may wish to tune it up or down, depending on your security needs and available processing power. To do so, you'll subclass the appropriate algorithm and override the `iterations` parameters. For example, to increase the number of iterations used by the default PBKDF2 algorithm:

1. Create a subclass of `django.contrib.auth.hashers.PBKDF2PasswordHasher`:

```
from django.contrib.auth.hashers import PBKDF2PasswordHasher

class MyPBKDF2PasswordHasher(PBKDF2PasswordHasher):
    """
    A subclass of PBKDF2PasswordHasher that uses 100 times more iterations.
    """
    iterations = PBKDF2PasswordHasher.iterations * 100
```

   Save this somewhere in your project. For example, you might put this in a file like `myproject/hashers.py`.

2. Add your new hasher as the first entry in *PASSWORD_HASHERS*:

```
PASSWORD_HASHERS = [
    'myproject.hashers.MyPBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
    'django.contrib.auth.hashers.Argon2PasswordHasher',
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
    'django.contrib.auth.hashers.BCryptPasswordHasher',
]
```

That's it – now your Django install will use more iterations when it stores passwords using PBKDF2.

Argon2 Argon2 has three attributes that can be customized:

1. `time_cost` controls the number of iterations within the hash.
2. `memory_cost` controls the size of memory that must be used during the computation of the hash.
3. `parallelism` controls how many CPUs the computation of the hash can be parallelized on.

The default values of these attributes are probably fine for you. If you determine that the password hash is too fast or too slow, you can tweak it as follows:

1. Choose `parallelism` to be the number of threads you can spare computing the hash.
2. Choose `memory_cost` to be the KiB of memory you can spare.
3. Adjust `time_cost` and measure the time hashing a password takes. Pick a `time_cost` that takes an acceptable time for you. If `time_cost` set to 1 is unacceptably slow, lower `memory_cost`.

`memory_cost` interpretation

The argon2 command-line utility and some other libraries interpret the `memory_cost` parameter differently from the value that Django uses. The conversion is given by `memory_cost == 2 ** memory_cost_commandline`.

Password upgrading

When users log in, if their passwords are stored with anything other than the preferred algorithm, Django will automatically upgrade the algorithm to the preferred one. This means that old installs of Django will get automatically more secure as users log in, and it also means that you can switch to new (and better) storage algorithms as they get invented.

However, Django can only upgrade passwords that use algorithms mentioned in *PASSWORD_HASHERS*, so as you upgrade to new systems you should make sure never to *remove* entries from this list. If you do, users using unmentioned algorithms won't be able to upgrade. Hashed passwords will be updated when increasing (or decreasing) the number of PBKDF2 iterations or bcrypt rounds.

Be aware that if all the passwords in your database aren't encoded in the default hasher's algorithm, you may be vulnerable to a user enumeration timing attack due to a difference between the duration of a login request for a user with a password encoded in a non-default algorithm and the duration of a login request for a nonexistent user (which runs the default hasher). You may be able to mitigate this by *upgrading older password hashes*.

Passwords updates when changing the number of bcrypt rounds was added.

Password upgrading without requiring a login

If you have an existing database with an older, weak hash such as MD5 or SHA1, you might want to upgrade those hashes yourself instead of waiting for the upgrade to happen when a user logs in (which may never happen if a user doesn't return to your site). In this case, you can use a "wrapped" password hasher.

For this example, we'll migrate a collection of SHA1 hashes to use PBKDF2(SHA1(password)) and add the corresponding password hasher for checking if a user entered the correct password on login. We assume we're using the built-in `User` model and that our project has an `accounts` app. You can modify the pattern to work with any algorithm or with a custom user model.

First, we'll add the custom hasher:

accounts/hashers.py

```
from django.contrib.auth.hashers import (
    PBKDF2PasswordHasher, SHA1PasswordHasher,
)


class PBKDF2WrappedSHA1PasswordHasher(PBKDF2PasswordHasher):
    algorithm = 'pbkdf2_wrapped_sha1'

    def encode_sha1_hash(self, sha1_hash, salt, iterations=None):
        return super(PBKDF2WrappedSHA1PasswordHasher, self).encode(sha1_hash, salt, iterations)

    def encode(self, password, salt, iterations=None):
        _, _, sha1_hash = SHA1PasswordHasher().encode(password, salt).split('$', 2)
        return self.encode_sha1_hash(sha1_hash, salt, iterations)
```

The data migration might look something like:

accounts/migrations/0002_migrate_sha1_passwords.py

```
from django.db import migrations

from ..hashers import PBKDF2WrappedSHA1PasswordHasher


```

```
def forwards_func(apps, schema_editor):
    User = apps.get_model('auth', 'User')
    users = User.objects.filter(password__startswith='sha1$')
    hasher = PBKDF2WrappedSHA1PasswordHasher()
    for user in users:
        algorithm, salt, sha1_hash = user.password.split('$', 2)
        user.password = hasher.encode_sha1_hash(sha1_hash, salt)
        user.save(update_fields=['password'])


class Migration(migrations.Migration):

    dependencies = [
        ('accounts', '0001_initial'),
        # replace this with the latest migration in contrib.auth
        ('auth', '####_migration_name'),
    ]

    operations = [
        migrations.RunPython(forwards_func),
    ]
```

Be aware that this migration will take on the order of several minutes for several thousand users, depending on the speed of your hardware.

Finally, we'll add a *PASSWORD_HASHERS* setting:

mysite/settings.py

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'accounts.hashers.PBKDF2WrappedSHA1PasswordHasher',
]
```

Include any other hashers that your site uses in this list.

Included hashers

The full list of hashers included in Django is:

```
[
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
    'django.contrib.auth.hashers.Argon2PasswordHasher',
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
    'django.contrib.auth.hashers.BCryptPasswordHasher',
    'django.contrib.auth.hashers.SHA1PasswordHasher',
    'django.contrib.auth.hashers.MD5PasswordHasher',
    'django.contrib.auth.hashers.UnsaltedSHA1PasswordHasher',
    'django.contrib.auth.hashers.UnsaltedMD5PasswordHasher',
    'django.contrib.auth.hashers.CryptPasswordHasher',
]
```

The corresponding algorithm names are:

- `pbkdf2_sha256`
- `pbkdf2_sha1`
- `argon2`

- `bcrypt_sha256`
- `bcrypt`
- `sha1`
- `md5`
- `unsalted_sha1`
- `unsalted_md5`
- `crypt`

Writing your own hasher

If you write your own password hasher that contains a work factor such as a number of iterations, you should implement a `harden_runtime(self, password, encoded)` method to bridge the runtime gap between the work factor supplied in the `encoded` password and the default work factor of the hasher. This prevents a user enumeration timing attack due to difference between a login request for a user with a password encoded in an older number of iterations and a nonexistent user (which runs the default hasher's default number of iterations).

Taking PBKDF2 as example, if `encoded` contains 20,000 iterations and the hasher's default `iterations` is 30,000, the method should run `password` through another 10,000 iterations of PBKDF2.

If your hasher doesn't have a work factor, implement the method as a no-op (`pass`).

Manually managing a user's password

The *`django.contrib.auth.hashers`* module provides a set of functions to create and validate hashed password. You can use them independently from the `User` model.

`check_password`(*password*, *encoded*)
: If you'd like to manually authenticate a user by comparing a plain-text password to the hashed password in the database, use the convenience function *`check_password()`*. It takes two arguments: the plain-text password to check, and the full value of a user's `password` field in the database to check against, and returns `True` if they match, `False` otherwise.

`make_password`(*password*, *salt=None*, *hasher='default'*)
: Creates a hashed password in the format used by this application. It takes one mandatory argument: the password in plain-text. Optionally, you can provide a salt and a hashing algorithm to use, if you don't want to use the defaults (first entry of `PASSWORD_HASHERS` setting). See *Included hashers* for the algorithm name of each hasher. If the password argument is `None`, an unusable password is returned (a one that will be never accepted by *`check_password()`*).

`is_password_usable`(*encoded_password*)
: Checks if the given string is a hashed password that has a chance of being verified against *`check_password()`*.

Password validation

Users often choose poor passwords. To help mitigate this problem, Django offers pluggable password validation. You can configure multiple password validators at the same time. A few validators are included in Django, but it's simple to write your own as well.

Each password validator must provide a help text to explain the requirements to the user, validate a given password and return an error message if it does not meet the requirements, and optionally receive passwords that have been set. Validators can also have optional settings to fine tune their behavior.

Validation is controlled by the *AUTH_PASSWORD_VALIDATORS* setting. The default for the setting is an empty list, which means no validators are applied. In new projects created with the default *startproject* template, a simple set of validators is enabled.

By default, validators are used in the forms to reset or change passwords and in the *createsuperuser* and *changepassword* management commands. Validators aren't applied at the model level, for example in `User.objects.create_user()` and `create_superuser()`, because we assume that developers, not users, interact with Django at that level and also because model validation doesn't automatically run as part of creating models.

: Password validation can prevent the use of many types of weak passwords. However, the fact that a password passes all the validators doesn't guarantee that it is a strong password. There are many factors that can weaken a password that are not detectable by even the most advanced password validators.

Enabling password validation

Password validation is configured in the *AUTH_PASSWORD_VALIDATORS* setting:

```
AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME': 'django.contrib.auth.password_validation.UserAttributeSimilarityValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator',
        'OPTIONS': {
            'min_length': 9,
        }
    },
    {
        'NAME': 'django.contrib.auth.password_validation.CommonPasswordValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.NumericPasswordValidator',
    },
]
```

This example enables all four included validators:

- `UserAttributeSimilarityValidator`, which checks the similarity between the password and a set of attributes of the user.
- `MinimumLengthValidator`, which simply checks whether the password meets a minimum length. This validator is configured with a custom option: it now requires the minimum length to be nine characters, instead of the default eight.
- `CommonPasswordValidator`, which checks whether the password occurs in a list of common passwords. By default, it compares to an included list of 1000 common passwords.
- `NumericPasswordValidator`, which checks whether the password isn't entirely numeric.

For `UserAttributeSimilarityValidator` and `CommonPasswordValidator`, we're simply using the default settings in this example. `NumericPasswordValidator` has no settings.

The help texts and any errors from password validators are always returned in the order they are listed in *AUTH_PASSWORD_VALIDATORS*.

Included validators

Django includes four validators:

class **MinimumLengthValidator**(*min_length=8*)
Validates whether the password meets a minimum length. The minimum length can be customized with the `min_length` parameter.

class **UserAttributeSimilarityValidator**(*user_attributes=DEFAULT_USER_ATTRIBUTES*, *max_similarity=0.7*)
Validates whether the password is sufficiently different from certain attributes of the user.

The `user_attributes` parameter should be an iterable of names of user attributes to compare to. If this argument is not provided, the default is used: `'username'`, `'first_name'`, `'last_name'`, `'email'`. Attributes that don't exist are ignored.

The maximum similarity the password can have, before it is rejected, can be set with the `max_similarity` parameter, on a scale of 0 to 1. A setting of 0 will cause all passwords to be rejected, whereas a setting of 1 will cause it to only reject passwords that are identical to an attribute's value.

class **CommonPasswordValidator**(*password_list_path=DEFAULT_PASSWORD_LIST_PATH*)
Validates whether the password is not a common password. By default, this checks against a list of 1000 common password created by Mark Burnett.

The `password_list_path` can be set to the path of a custom file of common passwords. This file should contain one password per line and may be plain text or gzipped.

class **NumericPasswordValidator**
Validates whether the password is not entirely numeric.

Integrating validation

There are a few functions in `django.contrib.auth.password_validation` that you can call from your own forms or other code to integrate password validation. This can be useful if you use custom forms for password setting, or if you have API calls that allow passwords to be set, for example.

validate_password(*password*, *user=None*, *password_validators=None*)
Validates a password. If all validators find the password valid, returns `None`. If one or more validators reject the password, raises a *ValidationError* with all the error messages from the validators.

The `user` object is optional: if it's not provided, some validators may not be able to perform any validation and will accept any password.

password_changed(*password*, *user=None*, *password_validators=None*)
Informs all validators that the password has been changed. This can be used by validators such as one that prevents password reuse. This should be called once the password has been successfully changed.

For subclasses of *AbstractBaseUser*, the password field will be marked as "dirty" when calling *set_password()* which triggers a call to `password_changed()` after the user is saved.

password_validators_help_texts(*password_validators=None*)
Returns a list of the help texts of all validators. These explain the password requirements to the user.

password_validators_help_text_html(*password_validators=None*)
Returns an HTML string with all help texts in an `<ul>`. This is helpful when adding password validation to forms, as you can pass the output directly to the `help_text` parameter of a form field.

get_password_validators(*validator_config*)
Returns a set of validator objects based on the `validator_config` parameter. By default, all functions use the validators defined in *AUTH_PASSWORD_VALIDATORS*, but by calling this function with an alternate set

of validators and then passing the result into the `password_validators` parameter of the other functions, your custom set of validators will be used instead. This is useful when you have a typical set of validators to use for most scenarios, but also have a special situation that requires a custom set. If you always use the same set of validators, there is no need to use this function, as the configuration from *`AUTH_PASSWORD_VALIDATORS`* is used by default.

The structure of `validator_config` is identical to the structure of *`AUTH_PASSWORD_VALIDATORS`*. The return value of this function can be passed into the `password_validators` parameter of the functions listed above.

Note that where the password is passed to one of these functions, this should always be the clear text password - not a hashed password.

Writing your own validator

If Django's built-in validators are not sufficient, you can write your own password validators. Validators are fairly simple classes. They must implement two methods:

- `validate(self, password, user=None)`: validate a password. Return `None` if the password is valid, or raise a *`ValidationError`* with an error message if the password is not valid. You must be able to deal with `user` being `None` - if that means your validator can't run, simply return `None` for no error.
- `get_help_text()`: provide a help text to explain the requirements to the user.

Any items in the `OPTIONS` in *`AUTH_PASSWORD_VALIDATORS`* for your validator will be passed to the constructor. All constructor arguments should have a default value.

Here's a basic example of a validator, with one optional setting:

```
from django.core.exceptions import ValidationError
from django.utils.translation import ugettext as _

class MinimumLengthValidator(object):
    def __init__(self, min_length=8):
        self.min_length = min_length

    def validate(self, password, user=None):
        if len(password) < self.min_length:
            raise ValidationError(
                _("This password must contain at least %(min_length)d characters."),
                code='password_too_short',
                params={'min_length': self.min_length},
            )

    def get_help_text(self):
        return _(
            "Your password must contain at least %(min_length)d characters."
            % {'min_length': self.min_length}
        )
```

You can also implement `password_changed(password, user=None)`, which will be called after a successful password change. That can be used to prevent password reuse, for example. However, if you decide to store a user's previous passwords, you should never do so in clear text.

3.10.3 Customizing authentication in Django

The authentication that comes with Django is good enough for most common cases, but you may have needs not met by the out-of-the-box defaults. To customize authentication to your projects needs involves understanding what points of the provided system are extensible or replaceable. This document provides details about how the auth system can be customized.

Authentication backends provide an extensible system for when a username and password stored with the User model need to be authenticated against a different service than Django's default.

You can give your models *custom permissions* that can be checked through Django's authorization system.

You can *extend* the default User model, or *substitute* a completely customized model.

Other authentication sources

There may be times you have the need to hook into another authentication source – that is, another source of usernames and passwords or authentication methods.

For example, your company may already have an LDAP setup that stores a username and password for every employee. It'd be a hassle for both the network administrator and the users themselves if users had separate accounts in LDAP and the Django-based applications.

So, to handle situations like this, the Django authentication system lets you plug in other authentication sources. You can override Django's default database-based scheme, or you can use the default system in tandem with other systems.

See the *authentication backend reference* for information on the authentication backends included with Django.

Specifying authentication backends

Behind the scenes, Django maintains a list of "authentication backends" that it checks for authentication. When somebody calls `django.contrib.auth.authenticate()` – as described in *How to log a user in* – Django tries authenticating across all of its authentication backends. If the first authentication method fails, Django tries the second one, and so on, until all backends have been attempted.

The list of authentication backends to use is specified in the `AUTHENTICATION_BACKENDS` setting. This should be a list of Python path names that point to Python classes that know how to authenticate. These classes can be anywhere on your Python path.

By default, `AUTHENTICATION_BACKENDS` is set to:

```
['django.contrib.auth.backends.ModelBackend']
```

That's the basic authentication backend that checks the Django users database and queries the built-in permissions. It does not provide protection against brute force attacks via any rate limiting mechanism. You may either implement your own rate limiting mechanism in a custom auth backend, or use the mechanisms provided by most Web servers.

The order of `AUTHENTICATION_BACKENDS` matters, so if the same username and password is valid in multiple backends, Django will stop processing at the first positive match.

If a backend raises a `PermissionDenied` exception, authentication will immediately fail. Django won't check the backends that follow.

: Once a user has authenticated, Django stores which backend was used to authenticate the user in the user's session, and re-uses the same backend for the duration of that session whenever access to the currently authenticated user is needed. This effectively means that authentication sources are cached on a per-session basis, so if you change

AUTHENTICATION_BACKENDS, you'll need to clear out session data if you need to force users to re-authenticate using different methods. A simple way to do that is simply to execute `Session.objects.all().delete()`.

Writing an authentication backend

An authentication backend is a class that implements two required methods: `get_user(user_id)` and `authenticate(**credentials)`, as well as a set of optional permission related *authorization methods*.

The `get_user` method takes a `user_id` – which could be a username, database ID or whatever, but has to be the primary key of your `User` object – and returns a `User` object.

The `authenticate` method takes credentials as keyword arguments. Most of the time, it'll just look like this:

```
class MyBackend(object):
    def authenticate(self, username=None, password=None):
        # Check the username/password and return a User.
        ...
```

But it could also authenticate a token, like so:

```
class MyBackend(object):
    def authenticate(self, token=None):
        # Check the token and return a User.
        ...
```

Either way, `authenticate` should check the credentials it gets, and it should return a `User` object that matches those credentials, if the credentials are valid. If they're not valid, it should return `None`.

The Django admin is tightly coupled to the Django *User object*. The best way to deal with this is to create a Django `User` object for each user that exists for your backend (e.g., in your LDAP directory, your external SQL database, etc.) You can either write a script to do this in advance, or your `authenticate` method can do it the first time a user logs in.

Here's an example backend that authenticates against a username and password variable defined in your `settings.py` file and creates a Django `User` object the first time a user authenticates:

```
from django.conf import settings
from django.contrib.auth.hashers import check_password
from django.contrib.auth.models import User

class SettingsBackend(object):
    """
    Authenticate against the settings ADMIN_LOGIN and ADMIN_PASSWORD.

    Use the login name and a hash of the password. For example:

    ADMIN_LOGIN = 'admin'
    ADMIN_PASSWORD = 'pbkdf2_sha256$30000$Vo0VlMnkR4Bk$qEvtdyZRWTcOsCnI/oQ7fVOu1XAURIZYoOZ3iq8Dr4M='
    """

    def authenticate(self, username=None, password=None):
        login_valid = (settings.ADMIN_LOGIN == username)
        pwd_valid = check_password(password, settings.ADMIN_PASSWORD)
        if login_valid and pwd_valid:
            try:
                user = User.objects.get(username=username)
            except User.DoesNotExist:
                # Create a new user. There's no need to set a password
```

```
                # because only the password from settings.py is checked.
                user = User(username=username)
                user.is_staff = True
                user.is_superuser = True
                user.save()
            return user
        return None

    def get_user(self, user_id):
        try:
            return User.objects.get(pk=user_id)
        except User.DoesNotExist:
            return None
```

Handling authorization in custom backends

Custom auth backends can provide their own permissions.

The user model will delegate permission lookup functions (*get_group_permissions()*, *get_all_permissions()*, *has_perm()*, and *has_module_perms()*) to any authentication backend that implements these functions.

The permissions given to the user will be the superset of all permissions returned by all backends. That is, Django grants a permission to a user that any one backend grants.

If a backend raises a *PermissionDenied* exception in *has_perm()* or *has_module_perms()*, the authorization will immediately fail and Django won't check the backends that follow.

The simple backend above could implement permissions for the magic admin fairly simply:

```
class SettingsBackend(object):
    ...
    def has_perm(self, user_obj, perm, obj=None):
        return user_obj.username == settings.ADMIN_LOGIN
```

This gives full permissions to the user granted access in the above example. Notice that in addition to the same arguments given to the associated *django.contrib.auth.models.User* functions, the backend auth functions all take the user object, which may be an anonymous user, as an argument.

A full authorization implementation can be found in the `ModelBackend` class in django/contrib/auth/backends.py, which is the default backend and queries the `auth_permission` table most of the time. If you wish to provide custom behavior for only part of the backend API, you can take advantage of Python inheritance and subclass `ModelBackend` instead of implementing the complete API in a custom backend.

Authorization for anonymous users An anonymous user is one that is not authenticated i.e. they have provided no valid authentication details. However, that does not necessarily mean they are not authorized to do anything. At the most basic level, most websites authorize anonymous users to browse most of the site, and many allow anonymous posting of comments etc.

Django's permission framework does not have a place to store permissions for anonymous users. However, the user object passed to an authentication backend may be an *django.contrib.auth.models.AnonymousUser* object, allowing the backend to specify custom authorization behavior for anonymous users. This is especially useful for the authors of re-usable apps, who can delegate all questions of authorization to the auth backend, rather than needing settings, for example, to control anonymous access.

Authorization for inactive users An inactive user is a one that has its *is_active* field set to `False`. The *ModelBackend* and *RemoteUserBackend* authentication backends prohibits these users from authenticating. If a custom user model doesn't have an *is_active* field, all users will be allowed to authenticate.

You can use *AllowAllUsersModelBackend* or *AllowAllUsersRemoteUserBackend* if you want to allow inactive users to authenticate.

The support for anonymous users in the permission system allows for a scenario where anonymous users have permissions to do something while inactive authenticated users do not.

Do not forget to test for the `is_active` attribute of the user in your own backend permission methods.

In older versions, the *ModelBackend* allowed inactive users to authenticate.

Handling object permissions Django's permission framework has a foundation for object permissions, though there is no implementation for it in the core. That means that checking for object permissions will always return `False` or an empty list (depending on the check performed). An authentication backend will receive the keyword parameters `obj` and `user_obj` for each object related authorization method and can return the object level permission as appropriate.

Custom permissions

To create custom permissions for a given model object, use the `permissions` *model Meta attribute*.

This example Task model creates three custom permissions, i.e., actions users can or cannot do with Task instances, specific to your application:

```
class Task(models.Model):
    ...
    class Meta:
        permissions = (
            ("view_task", "Can see available tasks"),
            ("change_task_status", "Can change the status of tasks"),
            ("close_task", "Can remove a task by setting its status as closed"),
        )
```

The only thing this does is create those extra permissions when you run *manage.py migrate* (the function that creates permissions is connected to the *post_migrate* signal). Your code is in charge of checking the value of these permissions when a user is trying to access the functionality provided by the application (viewing tasks, changing the status of tasks, closing tasks.) Continuing the above example, the following checks if a user may view tasks:

```
user.has_perm('app.view_task')
```

Extending the existing `User` model

There are two ways to extend the default *User* model without substituting your own model. If the changes you need are purely behavioral, and don't require any change to what is stored in the database, you can create a *proxy model* based on *User*. This allows for any of the features offered by proxy models including default ordering, custom managers, or custom model methods.

If you wish to store information related to `User`, you can use a *OneToOneField* to a model containing the fields for additional information. This one-to-one model is often called a profile model, as it might store non-auth related information about a site user. For example you might create an Employee model:

```
from django.contrib.auth.models import User

class Employee(models.Model):
```

```
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    department = models.CharField(max_length=100)
```

Assuming an existing Employee Fred Smith who has both a User and Employee model, you can access the related information using Django's standard related model conventions:

```
>>> u = User.objects.get(username='fsmith')
>>> freds_department = u.employee.department
```

To add a profile model's fields to the user page in the admin, define an *InlineModelAdmin* (for this example, we'll use a *StackedInline*) in your app's `admin.py` and add it to a `UserAdmin` class which is registered with the *User* class:

```
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin as BaseUserAdmin
from django.contrib.auth.models import User

from my_user_profile_app.models import Employee

# Define an inline admin descriptor for Employee model
# which acts a bit like a singleton
class EmployeeInline(admin.StackedInline):
    model = Employee
    can_delete = False
    verbose_name_plural = 'employee'

# Define a new User admin
class UserAdmin(BaseUserAdmin):
    inlines = (EmployeeInline, )

# Re-register UserAdmin
admin.site.unregister(User)
admin.site.register(User, UserAdmin)
```

These profile models are not special in any way - they are just Django models that happen to have a one-to-one link with a User model. As such, they do not get auto created when a user is created, but a *django.db.models.signals.post_save* could be used to create or update related models as appropriate.

Note that using related models results in additional queries or joins to retrieve the related data, and depending on your needs substituting the User model and adding the related fields may be your better option. However existing links to the default User model within your project's apps may justify the extra database load.

Substituting a custom `User` model

Some kinds of projects may have authentication requirements for which Django's built-in *User* model is not always appropriate. For instance, on some sites it makes more sense to use an email address as your identification token instead of a username.

Django allows you to override the default User model by providing a value for the *AUTH_USER_MODEL* setting that references a custom model:

```
AUTH_USER_MODEL = 'myapp.MyUser'
```

This dotted pair describes the name of the Django app (which must be in your *INSTALLED_APPS*), and the name of the Django model that you wish to use as your User model.

: Changing *AUTH_USER_MODEL* has a big effect on your database structure. It changes the tables that are available, and it will affect the construction of foreign keys and many-to-many relationships. If you intend to set *AUTH_USER_MODEL*, you should set it before creating any migrations or running `manage.py migrate` for the first time.
Changing this setting after you have tables created is not supported by *makemigrations* and will result in you having to manually fix your schema, port your data from the old user table, and possibly manually reapply some migrations.

: Due to limitations of Django's dynamic dependency feature for swappable models, you must ensure that the model referenced by *AUTH_USER_MODEL* is created in the first migration of its app (usually called `0001_initial`); otherwise, you will have dependency issues.
In addition, you may run into a CircularDependencyError when running your migrations as Django won't be able to automatically break the dependency loop due to the dynamic dependency. If you see this error, you should break the loop by moving the models depended on by your User model into a second migration (you can try making two normal models that have a ForeignKey to each other and seeing how `makemigrations` resolves that circular dependency if you want to see how it's usually done)

Reusable apps and `AUTH_USER_MODEL`

Reusable apps shouldn't implement a custom user model. A project may use many apps, and two reusable apps that implemented a custom user model couldn't be used together. If you need to store per user information in your app, use a *ForeignKey* or *OneToOneField* to `settings.AUTH_USER_MODEL` as described below.

Referencing the `User` model

If you reference *User* directly (for example, by referring to it in a foreign key), your code will not work in projects where the *AUTH_USER_MODEL* setting has been changed to a different User model.

`get_user_model``()`

Instead of referring to *User* directly, you should reference the user model using `django.contrib.auth.get_user_model()`. This method will return the currently active User model – the custom User model if one is specified, or *User* otherwise.

When you define a foreign key or many-to-many relations to the User model, you should specify the custom model using the *AUTH_USER_MODEL* setting. For example:

```
from django.conf import settings
from django.db import models

class Article(models.Model):
    author = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
    )
```

When connecting to signals sent by the `User` model, you should specify the custom model using the *AUTH_USER_MODEL* setting. For example:

```
from django.conf import settings
from django.db.models.signals import post_save

def post_save_receiver(sender, instance, created, **kwargs):
    pass
```

```
post_save.connect(post_save_receiver, sender=settings.AUTH_USER_MODEL)
```

Generally speaking, you should reference the User model with the *AUTH_USER_MODEL* setting in code that is executed at import time. `get_user_model()` only works once Django has imported all models.

Specifying a custom `User` model

Model design considerations

Think carefully before handling information not directly related to authentication in your custom User Model.

It may be better to store app-specific user information in a model that has a relation with the User model. That allows each app to specify its own user data requirements without risking conflicts with other apps. On the other hand, queries to retrieve this related information will involve a database join, which may have an effect on performance.

Django expects your custom User model to meet some minimum requirements.

1. If you use the default authentication backend, then your model must have a single unique field that can be used for identification purposes. This can be a username, an email address, or any other unique attribute. A non-unique username field is allowed if you use a custom authentication backend that can support it.
2. Your model must provide a way to address the user in a "short" and "long" form. The most common interpretation of this would be to use the user's given name as the "short" identifier, and the user's full name as the "long" identifier. However, there are no constraints on what these two methods return - if you want, they can return exactly the same value.

The easiest way to construct a compliant custom User model is to inherit from *AbstractBaseUser*. *AbstractBaseUser* provides the core implementation of a `User` model, including hashed passwords and tokenized password resets. You must then provide some key implementation details:

class `models.`**CustomUser**

USERNAME_FIELD
A string describing the name of the field on the User model that is used as the unique identifier. This will usually be a username of some kind, but it can also be an email address, or any other unique identifier. The field *must* be unique (i.e., have `unique=True` set in its definition), unless you use a custom authentication backend that can support non-unique usernames.

In the following example, the field `identifier` is used as the identifying field:

```
class MyUser(AbstractBaseUser):
    identifier = models.CharField(max_length=40, unique=True)
    ...
    USERNAME_FIELD = 'identifier'
```

USERNAME_FIELD now supports *ForeignKey*s. Since there is no way to pass model instances during the *createsuperuser* prompt, expect the user to enter the value of *to_field* value (the *primary_key* by default) of an existing instance.

REQUIRED_FIELDS
A list of the field names that will be prompted for when creating a user via the *createsuperuser* management command. The user will be prompted to supply a value for each of these fields. It must include any field for which *blank* is `False` or undefined and may include additional fields you want prompted for when a user is created interactively. `REQUIRED_FIELDS` has no effect in other parts of Django, like creating a user in the admin.

For example, here is the partial definition for a `User` model that defines two required fields - a date of birth and height:

```
class MyUser(AbstractBaseUser):
    ...
    date_of_birth = models.DateField()
    height = models.FloatField()
    ...
    REQUIRED_FIELDS = ['date_of_birth', 'height']
```

: `REQUIRED_FIELDS` must contain all required fields on your `User` model, but should *not* contain the `USERNAME_FIELD` or `password` as these fields will always be prompted for.

`REQUIRED_FIELDS` now supports *`ForeignKey`*s. Since there is no way to pass model instances during the *`createsuperuser`* prompt, expect the user to enter the value of *`to_field`* value (the *`primary_key`* by default) of an existing instance.

`is_active`
A boolean attribute that indicates whether the user is considered "active". This attribute is provided as an attribute on `AbstractBaseUser` defaulting to `True`. How you choose to implement it will depend on the details of your chosen auth backends. See the documentation of the *`is_active attribute on the built-in user model`* for details.

`get_full_name``()`
A longer formal identifier for the user. A common interpretation would be the full name of the user, but it can be any string that identifies the user.

`get_short_name``()`
A short, informal identifier for the user. A common interpretation would be the first name of the user, but it can be any string that identifies the user in an informal way. It may also return the same value as *`django.contrib.auth.models.User.get_full_name()`*.

Importing `AbstractBaseUser`

`AbstractBaseUser` and `BaseUserManager` are importable from `django.contrib.auth.base_user` so that they can be imported without including `django.contrib.auth` in *`INSTALLED_APPS`* (this raised a deprecation warning in older versions and is no longer supported in Django 1.9).

The following attributes and methods are available on any subclass of *`AbstractBaseUser`*:

class `models.`**`AbstractBaseUser`**

`get_username``()`
Returns the value of the field nominated by `USERNAME_FIELD`.

`clean``()`
Normalizes the username by calling *`normalize_username()`*. If you override this method, be sure to call `super()` to retain the normalization.

classmethod `normalize_username`(*username*)
Applies NFKC Unicode normalization to usernames so that visually identical characters with different Unicode code points are considered identical.

`is_authenticated`
Read-only attribute which is always `True` (as opposed to `AnonymousUser.is_authenticated`

which is always `False`). This is a way to tell if the user has been authenticated. This does not imply any permissions and doesn't check if the user is active or has a valid session. Even though normally you will check this attribute on `request.user` to find out whether it has been populated by the *`AuthenticationMiddleware`* (representing the currently logged-in user), you should know this attribute is `True` for any *`User`* instance.

In older versions, this was a method. Backwards-compatibility support for using it as a method will be removed in Django 2.0.

`is_anonymous`
Read-only attribute which is always `False`. This is a way of differentiating *`User`* and *`AnonymousUser`* objects. Generally, you should prefer using *`is_authenticated`* to this attribute.

In older versions, this was a method. Backwards-compatibility support for using it as a method will be removed in Django 2.0.

`set_password`(*raw_password*)
Sets the user's password to the given raw string, taking care of the password hashing. Doesn't save the *`AbstractBaseUser`* object.

When the raw_password is `None`, the password will be set to an unusable password, as if *`set_unusable_password()`* were used.

`check_password`(*raw_password*)
Returns `True` if the given raw string is the correct password for the user. (This takes care of the password hashing in making the comparison.)

`set_unusable_password`()
Marks the user as having no password set. This isn't the same as having a blank string for a password. *`check_password()`* for this user will never return `True`. Doesn't save the *`AbstractBaseUser`* object.

You may need this if authentication for your application takes place against an existing external source such as an LDAP directory.

`has_usable_password`()
Returns `False` if *`set_unusable_password()`* has been called for this user.

`get_session_auth_hash`()
Returns an HMAC of the password field. Used for *Session invalidation on password change*.

You should also define a custom manager for your `User` model. If your `User` model defines `username`, `email`, `is_staff`, `is_active`, `is_superuser`, `last_login`, and `date_joined` fields the same as Django's default `User`, you can just install Django's *`UserManager`*; however, if your `User` model defines different fields, you will need to define a custom manager that extends *`BaseUserManager`* providing two additional methods:

class `models.`**`CustomUserManager`**

`create_user`(**username_field**, *password=None*, ***other_fields*)
The prototype of `create_user()` should accept the username field, plus all required fields as arguments. For example, if your user model uses `email` as the username field, and has `date_of_birth` as a required field, then `create_user` should be defined as:

```
def create_user(self, email, date_of_birth, password=None):
    # create user here
    ...
```

`create_superuser`(**username_field**, *password*, ***other_fields*)
The prototype of `create_superuser()` should accept the username field, plus all required fields as

arguments. For example, if your user model uses `email` as the username field, and has `date_of_birth` as a required field, then `create_superuser` should be defined as:

```
def create_superuser(self, email, date_of_birth, password):
    # create superuser here
    ...
```

Unlike `create_user()`, `create_superuser()` *must* require the caller to provide a password.

BaseUserManager provides the following utility methods:

class `models.`**BaseUserManager**

classmethod **normalize_email**(*email*)
Normalizes email addresses by lowercasing the domain portion of the email address.

get_by_natural_key(*username*)
Retrieves a user instance using the contents of the field nominated by `USERNAME_FIELD`.

make_random_password(*length=10*, *allowed_chars='abcdefghjkmnpqrstuvwxyzABCDEFGHJKLMNPQRSTUVWXYZ234*
Returns a random password with the given length and given string of allowed characters. Note that the default value of `allowed_chars` doesn't contain letters that can cause user confusion, including:

- `i`, `l`, `I`, and `1` (lowercase letter i, lowercase letter L, uppercase letter i, and the number one)
- `o`, `O`, and `0` (lowercase letter o, uppercase letter o, and zero)

Extending Django's default `User`

If you're entirely happy with Django's *User* model and you just want to add some additional profile information, you could simply subclass `django.contrib.auth.models.AbstractUser` and add your custom profile fields, although we'd recommend a separate model as described in the "Model design considerations" note of *Specifying a custom User model*. `AbstractUser` provides the full implementation of the default *User* as an *abstract model*.

Custom users and the built-in auth forms

Django's built-in *forms* and *views* make certain assumptions about the user model that they are working with.

The following forms are compatible with any subclass of *AbstractBaseUser*:

- *AuthenticationForm*: Uses the username field specified by *USERNAME_FIELD*.
- *SetPasswordForm*
- *PasswordChangeForm*
- *AdminPasswordChangeForm*

The following forms make assumptions about the user model and can be used as-is if those assumptions are met:

- *PasswordResetForm*: Assumes that the user model has a field named `email` that can be used to identify the user and a boolean field named `is_active` to prevent password resets for inactive users.

Finally, the following forms are tied to *User* and need to be rewritten or extended to work with a custom user model:

- *UserCreationForm*
- *UserChangeForm*

If your custom user model is a simple subclass of `AbstractUser`, then you can extend these forms in this manner:

```
from django.contrib.auth.forms import UserCreationForm
from myapp.models import CustomUser

class CustomUserCreationForm(UserCreationForm):

    class Meta(UserCreationForm.Meta):
        model = CustomUser
        fields = UserCreationForm.Meta.fields + ('custom_field',)
```

Custom users and `django.contrib.admin`

If you want your custom User model to also work with Admin, your User model must define some additional attributes and methods. These methods allow the admin to control access of the User to admin content:

class `models.`**CustomUser**

is_staff
Returns `True` if the user is allowed to have access to the admin site.

is_active
Returns `True` if the user account is currently active.

has_perm(perm, obj=None):
Returns `True` if the user has the named permission. If `obj` is provided, the permission needs to be checked against a specific object instance.

has_module_perms(app_label):
Returns `True` if the user has permission to access models in the given app.

You will also need to register your custom User model with the admin. If your custom User model extends `django.contrib.auth.models.AbstractUser`, you can use Django's existing `django.contrib.auth.admin.UserAdmin` class. However, if your User model extends *AbstractBaseUser*, you'll need to define a custom `ModelAdmin` class. It may be possible to subclass the default `django.contrib.auth.admin.UserAdmin`; however, you'll need to override any of the definitions that refer to fields on `django.contrib.auth.models.AbstractUser` that aren't on your custom User class.

Custom users and permissions

To make it easy to include Django's permission framework into your own User class, Django provides *PermissionsMixin*. This is an abstract model you can include in the class hierarchy for your User model, giving you all the methods and database fields necessary to support Django's permission model.

PermissionsMixin provides the following methods and attributes:

class `models.`**PermissionsMixin**

is_superuser
Boolean. Designates that this user has all permissions without explicitly assigning them.

get_group_permissions(*obj=None*)
Returns a set of permission strings that the user has, through their groups.

If `obj` is passed in, only returns the group permissions for this specific object.

get_all_permissions(*obj=None*)
Returns a set of permission strings that the user has, both through group and user permissions.

If `obj` is passed in, only returns the permissions for this specific object.

has_perm(*perm*, *obj=None*)
Returns `True` if the user has the specified permission, where `perm` is in the format `"<app label>.<permission codename>"` (see *permissions*). If the user is inactive, this method will always return `False`.

If `obj` is passed in, this method won't check for a permission for the model, but for this specific object.

has_perms(*perm_list*, *obj=None*)
Returns `True` if the user has each of the specified permissions, where each perm is in the format `"<app label>.<permission codename>"`. If the user is inactive, this method will always return `False`.

If `obj` is passed in, this method won't check for permissions for the model, but for the specific object.

has_module_perms(*package_name*)
Returns `True` if the user has any permissions in the given package (the Django app label). If the user is inactive, this method will always return `False`.

`PermissionsMixin` and `ModelBackend`

If you don't include the *`PermissionsMixin`*, you must ensure you don't invoke the permissions methods on `ModelBackend`. `ModelBackend` assumes that certain fields are available on your user model. If your `User` model doesn't provide those fields, you will receive database errors when you check permissions.

Custom users and proxy models

One limitation of custom User models is that installing a custom User model will break any proxy model extending *`User`*. Proxy models must be based on a concrete base class; by defining a custom User model, you remove the ability of Django to reliably identify the base class.

If your project uses proxy models, you must either modify the proxy to extend the User model that is currently in use in your project, or merge your proxy's behavior into your User subclass.

A full example

Here is an example of an admin-compliant custom user app. This user model uses an email address as the username, and has a required date of birth; it provides no permission checking, beyond a simple `admin` flag on the user account. This model would be compatible with all the built-in auth forms and views, except for the User creation forms. This example illustrates how most of the components work together, but is not intended to be copied directly into projects for production use.

This code would all live in a `models.py` file for a custom authentication app:

```
from django.db import models
from django.contrib.auth.models import (
    BaseUserManager, AbstractBaseUser
)


class MyUserManager(BaseUserManager):
    def create_user(self, email, date_of_birth, password=None):
        """
        Creates and saves a User with the given email, date of
        birth and password.
        """
```

```
        if not email:
            raise ValueError('Users must have an email address')

        user = self.model(
            email=self.normalize_email(email),
            date_of_birth=date_of_birth,
        )

        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_superuser(self, email, date_of_birth, password):
        """
        Creates and saves a superuser with the given email, date of
        birth and password.
        """
        user = self.create_user(
            email,
            password=password,
            date_of_birth=date_of_birth,
        )
        user.is_admin = True
        user.save(using=self._db)
        return user


class MyUser(AbstractBaseUser):
    email = models.EmailField(
        verbose_name='email address',
        max_length=255,
        unique=True,
    )
    date_of_birth = models.DateField()
    is_active = models.BooleanField(default=True)
    is_admin = models.BooleanField(default=False)

    objects = MyUserManager()

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['date_of_birth']

    def get_full_name(self):
        # The user is identified by their email address
        return self.email

    def get_short_name(self):
        # The user is identified by their email address
        return self.email

    def __str__(self):              # __unicode__ on Python 2
        return self.email

    def has_perm(self, perm, obj=None):
        "Does the user have a specific permission?"
        # Simplest possible answer: Yes, always
        return True
```

```
    def has_module_perms(self, app_label):
        "Does the user have permissions to view the app `app_label`?"
        # Simplest possible answer: Yes, always
        return True

    @property
    def is_staff(self):
        "Is the user a member of staff?"
        # Simplest possible answer: All admins are staff
        return self.is_admin
```

Then, to register this custom User model with Django's admin, the following code would be required in the app's `admin.py` file:

```
from django import forms
from django.contrib import admin
from django.contrib.auth.models import Group
from django.contrib.auth.admin import UserAdmin as BaseUserAdmin
from django.contrib.auth.forms import ReadOnlyPasswordHashField

from customauth.models import MyUser


class UserCreationForm(forms.ModelForm):
    """A form for creating new users. Includes all the required
    fields, plus a repeated password."""
    password1 = forms.CharField(label='Password', widget=forms.PasswordInput)
    password2 = forms.CharField(label='Password confirmation', widget=forms.PasswordInput)

    class Meta:
        model = MyUser
        fields = ('email', 'date_of_birth')

    def clean_password2(self):
        # Check that the two password entries match
        password1 = self.cleaned_data.get("password1")
        password2 = self.cleaned_data.get("password2")
        if password1 and password2 and password1 != password2:
            raise forms.ValidationError("Passwords don't match")
        return password2

    def save(self, commit=True):
        # Save the provided password in hashed format
        user = super(UserCreationForm, self).save(commit=False)
        user.set_password(self.cleaned_data["password1"])
        if commit:
            user.save()
        return user


class UserChangeForm(forms.ModelForm):
    """A form for updating users. Includes all the fields on
    the user, but replaces the password field with admin's
    password hash display field.
    """
    password = ReadOnlyPasswordHashField()

    class Meta:
```

```
        model = MyUser
        fields = ('email', 'password', 'date_of_birth', 'is_active', 'is_admin')

    def clean_password(self):
        # Regardless of what the user provides, return the initial value.
        # This is done here, rather than on the field, because the
        # field does not have access to the initial value
        return self.initial["password"]


class UserAdmin(BaseUserAdmin):
    # The forms to add and change user instances
    form = UserChangeForm
    add_form = UserCreationForm

    # The fields to be used in displaying the User model.
    # These override the definitions on the base UserAdmin
    # that reference specific fields on auth.User.
    list_display = ('email', 'date_of_birth', 'is_admin')
    list_filter = ('is_admin',)
    fieldsets = (
        (None, {'fields': ('email', 'password')}),
        ('Personal info', {'fields': ('date_of_birth',)}),
        ('Permissions', {'fields': ('is_admin',)}),
    )
    # add_fieldsets is not a standard ModelAdmin attribute. UserAdmin
    # overrides get_fieldsets to use this attribute when creating a user.
    add_fieldsets = (
        (None, {
            'classes': ('wide',),
            'fields': ('email', 'date_of_birth', 'password1', 'password2')}
        ),
    )
    search_fields = ('email',)
    ordering = ('email',)
    filter_horizontal = ()

# Now register the new UserAdmin...
admin.site.register(MyUser, UserAdmin)
# ... and, since we're not using Django's built-in permissions,
# unregister the Group model from admin.
admin.site.unregister(Group)
```

Finally, specify the custom model as the default user model for your project using the *AUTH_USER_MODEL* setting in your `settings.py`:

```
AUTH_USER_MODEL = 'customauth.MyUser'
```

Django comes with a user authentication system. It handles user accounts, groups, permissions and cookie-based user sessions. This section of the documentation explains how the default implementation works out of the box, as well as how to extend and customize it to suit your project's needs.

3.10.4 Overview

The Django authentication system handles both authentication and authorization. Briefly, authentication verifies a user is who they claim to be, and authorization determines what an authenticated user is allowed to do. Here the term authentication is used to refer to both tasks.

The auth system consists of:

- Users
- Permissions: Binary (yes/no) flags designating whether a user may perform a certain task.
- Groups: A generic way of applying labels and permissions to more than one user.
- A configurable password hashing system
- Forms and view tools for logging in users, or restricting content
- A pluggable backend system

The authentication system in Django aims to be very generic and doesn't provide some features commonly found in web authentication systems. Solutions for some of these common problems have been implemented in third-party packages:

- Password strength checking
- Throttling of login attempts
- Authentication against third-parties (OAuth, for example)

3.10.5 Installation

Authentication support is bundled as a Django contrib module in `django.contrib.auth`. By default, the required configuration is already included in the `settings.py` generated by *`django-admin startproject`*, these consist of two items listed in your *`INSTALLED_APPS`* setting:

1. `'django.contrib.auth'` contains the core of the authentication framework, and its default models.
2. `'django.contrib.contenttypes'` is the Django content type system, which allows permissions to be associated with models you create.

and these items in your *`MIDDLEWARE`* setting:

1. *`SessionMiddleware`* manages sessions across requests.
2. *`AuthenticationMiddleware`* associates users with requests using sessions.

With these settings in place, running the command `manage.py migrate` creates the necessary database tables for auth related models and permissions for any models defined in your installed apps.

3.10.6 Usage

Using Django's default implementation

- *Working with User objects*
- *Permissions and authorization*
- *Authentication in web requests*
- *Managing users in the admin*

API reference for the default implementation

Customizing Users and authentication

Password management in Django

3.11 Django's cache framework

A fundamental trade-off in dynamic websites is, well, they're dynamic. Each time a user requests a page, the Web server makes all sorts of calculations – from database queries to template rendering to business logic – to create the page that your site's visitor sees. This is a lot more expensive, from a processing-overhead perspective, than your standard read-a-file-off-the-filesystem server arrangement.

For most Web applications, this overhead isn't a big deal. Most Web applications aren't `washingtonpost.com` or `slashdot.org`; they're simply small- to medium-sized sites with so-so traffic. But for medium- to high-traffic sites, it's essential to cut as much overhead as possible.

That's where caching comes in.

To cache something is to save the result of an expensive calculation so that you don't have to perform the calculation next time. Here's some pseudocode explaining how this would work for a dynamically generated Web page:

```
given a URL, try finding that page in the cache
if the page is in the cache:
    return the cached page
else:
    generate the page
    save the generated page in the cache (for next time)
    return the generated page
```

Django comes with a robust cache system that lets you save dynamic pages so they don't have to be calculated for each request. For convenience, Django offers different levels of cache granularity: You can cache the output of specific views, you can cache only the pieces that are difficult to produce, or you can cache your entire site.

Django also works well with "downstream" caches, such as Squid and browser-based caches. These are the types of caches that you don't directly control but to which you can provide hints (via HTTP headers) about which parts of your site should be cached, and how.

:

The *Cache Framework design philosophy* explains a few of the design decisions of the framework.

3.11.1 Setting up the cache

The cache system requires a small amount of setup. Namely, you have to tell it where your cached data should live – whether in a database, on the filesystem or directly in memory. This is an important decision that affects your cache's performance; yes, some cache types are faster than others.

Your cache preference goes in the *CACHES* setting in your settings file. Here's an explanation of all available values for *CACHES*.

Memcached

The fastest, most efficient type of cache supported natively by Django, Memcached is an entirely memory-based cache server, originally developed to handle high loads at LiveJournal.com and subsequently open-sourced by Danga Interactive. It is used by sites such as Facebook and Wikipedia to reduce database access and dramatically increase site performance.

Memcached runs as a daemon and is allotted a specified amount of RAM. All it does is provide a fast interface for adding, retrieving and deleting data in the cache. All data is stored directly in memory, so there's no overhead of database or filesystem usage.

After installing Memcached itself, you'll need to install a Memcached binding. There are several Python Memcached bindings available; the two most common are python-memcached and pylibmc.

To use Memcached with Django:

- Set *BACKEND* to `django.core.cache.backends.memcached.MemcachedCache` or `django.core.cache.backends.memcached.PyLibMCCache` (depending on your chosen memcached binding)
- Set *LOCATION* to `ip:port` values, where `ip` is the IP address of the Memcached daemon and `port` is the port on which Memcached is running, or to a `unix:path` value, where `path` is the path to a Memcached Unix socket file.

In this example, Memcached is running on localhost (127.0.0.1) port 11211, using the `python-memcached` binding:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
    }
}
```

In this example, Memcached is available through a local Unix socket file `/tmp/memcached.sock` using the `python-memcached` binding:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': 'unix:/tmp/memcached.sock',
    }
}
```

When using the `pylibmc` binding, do not include the `unix:/` prefix:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.PyLibMCCache',
        'LOCATION': '/tmp/memcached.sock',
    }
}
```

One excellent feature of Memcached is its ability to share a cache over multiple servers. This means you can run Memcached daemons on multiple machines, and the program will treat the group of machines as a *single* cache, without the need to duplicate cache values on each machine. To take advantage of this feature, include all server addresses in *LOCATION*, either separated by semicolons or as a list.

In this example, the cache is shared over Memcached instances running on IP address 172.19.26.240 and 172.19.26.242, both on port 11211:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': [
            '172.19.26.240:11211',
            '172.19.26.242:11211',
        ]
    }
}
```

In the following example, the cache is shared over Memcached instances running on the IP addresses 172.19.26.240 (port 11211), 172.19.26.242 (port 11212), and 172.19.26.244 (port 11213):

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': [
            '172.19.26.240:11211',
            '172.19.26.242:11212',
            '172.19.26.244:11213',
        ]
    }
}
```

A final point about Memcached is that memory-based caching has a disadvantage: because the cached data is stored in memory, the data will be lost if your server crashes. Clearly, memory isn't intended for permanent data storage, so don't rely on memory-based caching as your only data storage. Without a doubt, *none* of the Django caching backends should be used for permanent storage – they're all intended to be solutions for caching, not storage – but we point this out here because memory-based caching is particularly temporary.

Database caching

Django can store its cached data in your database. This works best if you've got a fast, well-indexed database server.

To use a database table as your cache backend:

- Set *BACKEND* to `django.core.cache.backends.db.DatabaseCache`
- Set *LOCATION* to `tablename`, the name of the database table. This name can be whatever you want, as long as it's a valid table name that's not already being used in your database.

In this example, the cache table's name is `my_cache_table`:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.db.DatabaseCache',
        'LOCATION': 'my_cache_table',
    }
}
```

Creating the cache table

Before using the database cache, you must create the cache table with this command:

```
python manage.py createcachetable
```

This creates a table in your database that is in the proper format that Django's database-cache system expects. The name of the table is taken from *LOCATION*.

If you are using multiple database caches, *createcachetable* creates one table for each cache.

If you are using multiple databases, *createcachetable* observes the `allow_migrate()` method of your database routers (see below).

Like *migrate*, *createcachetable* won't touch an existing table. It will only create missing tables.

To print the SQL that would be run, rather than run it, use the *createcachetable --dry-run* option.

Multiple databases

If you use database caching with multiple databases, you'll also need to set up routing instructions for your database cache table. For the purposes of routing, the database cache table appears as a model named `CacheEntry`, in an application named `django_cache`. This model won't appear in the models cache, but the model details can be used for routing purposes.

For example, the following router would direct all cache read operations to `cache_replica`, and all write operations to `cache_primary`. The cache table will only be synchronized onto `cache_primary`:

```
class CacheRouter(object):
    """A router to control all database cache operations"""

    def db_for_read(self, model, **hints):
        "All cache read operations go to the replica"
        if model._meta.app_label == 'django_cache':
            return 'cache_replica'
        return None

    def db_for_write(self, model, **hints):
        "All cache write operations go to primary"
        if model._meta.app_label == 'django_cache':
            return 'cache_primary'
        return None

    def allow_migrate(self, db, app_label, model_name=None, **hints):
        "Only install the cache model on primary"
        if app_label == 'django_cache':
            return db == 'cache_primary'
        return None
```

If you don't specify routing directions for the database cache model, the cache backend will use the `default` database.

Of course, if you don't use the database cache backend, you don't need to worry about providing routing instructions for the database cache model.

Filesystem caching

The file-based backend serializes and stores each cache value as a separate file. To use this backend set *BACKEND* to `"django.core.cache.backends.filebased.FileBasedCache"` and *LOCATION* to a suitable directory. For example, to store cached data in `/var/tmp/django_cache`, use this setting:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': '/var/tmp/django_cache',
    }
}
```

If you're on Windows, put the drive letter at the beginning of the path, like this:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': 'c:/foo/bar',
    }
}
```

The directory path should be absolute – that is, it should start at the root of your filesystem. It doesn't matter whether you put a slash at the end of the setting.

Make sure the directory pointed-to by this setting exists and is readable and writable by the system user under which your Web server runs. Continuing the above example, if your server runs as the user `apache`, make sure the directory `/var/tmp/django_cache` exists and is readable and writable by the user `apache`.

Local-memory caching

This is the default cache if another is not specified in your settings file. If you want the speed advantages of in-memory caching but don't have the capability of running Memcached, consider the local-memory cache backend. This cache is per-process (see below) and thread-safe. To use it, set *BACKEND* to `"django.core.cache.backends.locmem.LocMemCache"`. For example:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
        'LOCATION': 'unique-snowflake',
    }
}
```

The cache *LOCATION* is used to identify individual memory stores. If you only have one `locmem` cache, you can omit the *LOCATION*; however, if you have more than one local memory cache, you will need to assign a name to at least one of them in order to keep them separate.

Note that each process will have its own private cache instance, which means no cross-process caching is possible. This obviously also means the local memory cache isn't particularly memory-efficient, so it's probably not a good choice for production environments. It's nice for development.

Dummy caching (for development)

Finally, Django comes with a "dummy" cache that doesn't actually cache – it just implements the cache interface without doing anything.

This is useful if you have a production site that uses heavy-duty caching in various places but a development/test environment where you don't want to cache and don't want to have to change your code to special-case the latter. To activate dummy caching, set *BACKEND* like so:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.dummy.DummyCache',
    }
}
```

Using a custom cache backend

While Django includes support for a number of cache backends out-of-the-box, sometimes you might want to use a customized cache backend. To use an external cache backend with Django, use the Python import path as the *BACKEND* of the *CACHES* setting, like so:

```
CACHES = {
    'default': {
        'BACKEND': 'path.to.backend',
    }
}
```

If you're building your own backend, you can use the standard cache backends as reference implementations. You'll find the code in the `django/core/cache/backends/` directory of the Django source.

Note: Without a really compelling reason, such as a host that doesn't support them, you should stick to the cache backends included with Django. They've been well-tested and are easy to use.

Cache arguments

Each cache backend can be given additional arguments to control caching behavior. These arguments are provided as additional keys in the *CACHES* setting. Valid arguments are as follows:

- *TIMEOUT*: The default timeout, in seconds, to use for the cache. This argument defaults to `300` seconds (5 minutes). You can set `TIMEOUT` to `None` so that, by default, cache keys never expire. A value of `0` causes keys to immediately expire (effectively "don't cache").
- *OPTIONS*: Any options that should be passed to the cache backend. The list of valid options will vary with each backend, and cache backends backed by a third-party library will pass their options directly to the underlying cache library.

  Cache backends that implement their own culling strategy (i.e., the `locmem`, `filesystem` and `database` backends) will honor the following options:

  - `MAX_ENTRIES`: The maximum number of entries allowed in the cache before old values are deleted. This argument defaults to `300`.
  - `CULL_FREQUENCY`: The fraction of entries that are culled when `MAX_ENTRIES` is reached. The actual ratio is `1 / CULL_FREQUENCY`, so set `CULL_FREQUENCY` to `2` to cull half the entries when `MAX_ENTRIES` is reached. This argument should be an integer and defaults to `3`.

    A value of `0` for `CULL_FREQUENCY` means that the entire cache will be dumped when `MAX_ENTRIES` is reached. On some backends (`database` in particular) this makes culling *much* faster at the expense of more cache misses.
- *KEY_PREFIX*: A string that will be automatically included (prepended by default) to all cache keys used by the Django server.

  See the *cache documentation* for more information.
- *VERSION*: The default version number for cache keys generated by the Django server.

  See the *cache documentation* for more information.
- *KEY_FUNCTION* A string containing a dotted path to a function that defines how to compose a prefix, version and key into a final cache key.

  See the *cache documentation* for more information.

In this example, a filesystem backend is being configured with a timeout of 60 seconds, and a maximum capacity of 1000 items:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': '/var/tmp/django_cache',
        'TIMEOUT': 60,
        'OPTIONS': {
            'MAX_ENTRIES': 1000
        }
    }
}
```

Invalid arguments are silently ignored, as are invalid values of known arguments.

3.11.2 The per-site cache

Once the cache is set up, the simplest way to use caching is to cache your entire site. You'll need to add `'django.middleware.cache.UpdateCacheMiddleware'` and `'django.middleware.cache.FetchFromCacheMiddleware'` to your *MIDDLEWARE* setting, as in this example:

```
MIDDLEWARE = [
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
]
```

: No, that's not a typo: the "update" middleware must be first in the list, and the "fetch" middleware must be last. The details are a bit obscure, but see *Order of MIDDLEWARE* below if you'd like the full story.

Then, add the following required settings to your Django settings file:

- *CACHE_MIDDLEWARE_ALIAS* – The cache alias to use for storage.
- *CACHE_MIDDLEWARE_SECONDS* – The number of seconds each page should be cached.
- *CACHE_MIDDLEWARE_KEY_PREFIX* – If the cache is shared across multiple sites using the same Django installation, set this to the name of the site, or some other string that is unique to this Django instance, to prevent key collisions. Use an empty string if you don't care.

`FetchFromCacheMiddleware` caches GET and HEAD responses with status 200, where the request and response headers allow. Responses to requests for the same URL with different query parameters are considered to be unique pages and are cached separately. This middleware expects that a HEAD request is answered with the same response headers as the corresponding GET request; in which case it can return a cached GET response for HEAD request.

Additionally, `UpdateCacheMiddleware` automatically sets a few headers in each *HttpResponse*:

- Sets the `Last-Modified` header to the current date/time when a fresh (not cached) version of the page is requested.
- Sets the `Expires` header to the current date/time plus the defined *CACHE_MIDDLEWARE_SECONDS*.
- Sets the `Cache-Control` header to give a max age for the page – again, from the *CACHE_MIDDLEWARE_SECONDS* setting.

See Middleware for more on middleware.

If a view sets its own cache expiry time (i.e. it has a `max-age` section in its `Cache-Control` header) then the page will be cached until the expiry time, rather than *CACHE_MIDDLEWARE_SECONDS*. Using the decorators in `django.views.decorators.cache` you can easily set a view's expiry time (using the *cache_control()* decorator) or disable caching for a view (using the *never_cache()* decorator). See the *using other headers* section for more on these decorators. If *USE_I18N* is set to `True` then the generated cache key will include the name of the active *language* – see also *How Django discovers language preference*). This allows you to easily cache multilingual sites without having to create the cache key yourself.

Cache keys also include the active *language* when *USE_L10N* is set to `True` and the *current time zone* when *USE_TZ* is set to `True`.

3.11.3 The per-view cache

`django.views.decorators.cache.`**`cache_page`**`()`

A more granular way to use the caching framework is by caching the output of individual views. `django.views.decorators.cache` defines a `cache_page` decorator that will automatically cache the view's response for you. It's easy to use:

```
from django.views.decorators.cache import cache_page

@cache_page(60 * 15)
def my_view(request):
    ...
```

`cache_page` takes a single argument: the cache timeout, in seconds. In the above example, the result of the `my_view()` view will be cached for 15 minutes. (Note that we've written it as `60 * 15` for the purpose of readability. `60 * 15` will be evaluated to `900` – that is, 15 minutes multiplied by 60 seconds per minute.)

The per-view cache, like the per-site cache, is keyed off of the URL. If multiple URLs point at the same view, each URL will be cached separately. Continuing the `my_view` example, if your URLconf looks like this:

```
urlpatterns = [
    url(r'^foo/([0-9]{1,2})/$', my_view),
]
```

then requests to `/foo/1/` and `/foo/23/` will be cached separately, as you may expect. But once a particular URL (e.g., `/foo/23/`) has been requested, subsequent requests to that URL will use the cache.

`cache_page` can also take an optional keyword argument, `cache`, which directs the decorator to use a specific cache (from your *CACHES* setting) when caching view results. By default, the `default` cache will be used, but you can specify any cache you want:

```
@cache_page(60 * 15, cache="special_cache")
def my_view(request):
    ...
```

You can also override the cache prefix on a per-view basis. `cache_page` takes an optional keyword argument, `key_prefix`, which works in the same way as the *CACHE_MIDDLEWARE_KEY_PREFIX* setting for the middleware. It can be used like this:

```
@cache_page(60 * 15, key_prefix="site1")
def my_view(request):
    ...
```

The `key_prefix` and `cache` arguments may be specified together. The `key_prefix` argument and the *KEY_PREFIX* specified under *CACHES* will be concatenated.

Specifying per-view cache in the URLconf

The examples in the previous section have hard-coded the fact that the view is cached, because `cache_page` alters the `my_view` function in place. This approach couples your view to the cache system, which is not ideal for several reasons. For instance, you might want to reuse the view functions on another, cache-less site, or you might want to distribute the views to people who might want to use them without being cached. The solution to these problems is to specify the per-view cache in the URLconf rather than next to the view functions themselves.

Doing so is easy: simply wrap the view function with `cache_page` when you refer to it in the URLconf. Here's the old URLconf from earlier:

```
urlpatterns = [
    url(r'^foo/([0-9]{1,2})/$', my_view),
]
```

Here's the same thing, with `my_view` wrapped in `cache_page`:

```
from django.views.decorators.cache import cache_page

urlpatterns = [
    url(r'^foo/([0-9]{1,2})/$', cache_page(60 * 15)(my_view)),
]
```

3.11.4 Template fragment caching

If you're after even more control, you can also cache template fragments using the `cache` template tag. To give your template access to this tag, put `{% load cache %}` near the top of your template.

The `{% cache %}` template tag caches the contents of the block for a given amount of time. It takes at least two arguments: the cache timeout, in seconds, and the name to give the cache fragment. The name will be taken as is, do not use a variable. For example:

```
{% load cache %}
{% cache 500 sidebar %}
    .. sidebar ..
{% endcache %}
```

Sometimes you might want to cache multiple copies of a fragment depending on some dynamic data that appears inside the fragment. For example, you might want a separate cached copy of the sidebar used in the previous example for every user of your site. Do this by passing additional arguments to the `{% cache %}` template tag to uniquely identify the cache fragment:

```
{% load cache %}
{% cache 500 sidebar request.user.username %}
    .. sidebar for logged in user ..
{% endcache %}
```

It's perfectly fine to specify more than one argument to identify the fragment. Simply pass as many arguments to `{% cache %}` as you need.

If *USE_I18N* is set to `True` the per-site middleware cache will *respect the active language*. For the `cache` template tag you could use one of the *translation-specific variables* available in templates to achieve the same result:

```
{% load i18n %}
{% load cache %}

{% get_current_language as LANGUAGE_CODE %}

{% cache 600 welcome LANGUAGE_CODE %}
    {% trans "Welcome to example.com" %}
{% endcache %}
```

The cache timeout can be a template variable, as long as the template variable resolves to an integer value. For example, if the template variable `my_timeout` is set to the value `600`, then the following two examples are equivalent:

```
{% cache 600 sidebar %} ... {% endcache %}
{% cache my_timeout sidebar %} ... {% endcache %}
```

This feature is useful in avoiding repetition in templates. You can set the timeout in a variable, in one place, and just reuse that value.

By default, the cache tag will try to use the cache called "template_fragments". If no such cache exists, it will fall back to using the default cache. You may select an alternate cache backend to use with the `using` keyword argument, which must be the last argument to the tag.

```
{% cache 300 local-thing ...  using="localcache" %}
```

It is considered an error to specify a cache name that is not configured.

`django.core.cache.utils.`**`make_template_fragment_key`**(*fragment_name*, *vary_on=None*)

If you want to obtain the cache key used for a cached fragment, you can use `make_template_fragment_key`. `fragment_name` is the same as second argument to the `cache` template tag; `vary_on` is a list of all additional arguments passed to the tag. This function can be useful for invalidating or overwriting a cached item, for example:

```
>>> from django.core.cache import cache
>>> from django.core.cache.utils import make_template_fragment_key
# cache key for {% cache 500 sidebar username %}
>>> key = make_template_fragment_key('sidebar', [username])
>>> cache.delete(key) # invalidates cached template fragment
```

3.11.5 The low-level cache API

Sometimes, caching an entire rendered page doesn't gain you very much and is, in fact, inconvenient overkill.

Perhaps, for instance, your site includes a view whose results depend on several expensive queries, the results of which change at different intervals. In this case, it would not be ideal to use the full-page caching that the per-site or per-view cache strategies offer, because you wouldn't want to cache the entire result (since some of the data changes often), but you'd still want to cache the results that rarely change.

For cases like this, Django exposes a simple, low-level cache API. You can use this API to store objects in the cache with any level of granularity you like. You can cache any Python object that can be pickled safely: strings, dictionaries, lists of model objects, and so forth. (Most common Python objects can be pickled; refer to the Python documentation for more information about pickling.)

Accessing the cache

`django.core.cache.`**`caches`**

You can access the caches configured in the *CACHES* setting through a dict-like object: `django.core.cache.caches`. Repeated requests for the same alias in the same thread will return the same object.

```
>>> from django.core.cache import caches
>>> cache1 = caches['myalias']
>>> cache2 = caches['myalias']
>>> cache1 is cache2
True
```

If the named key does not exist, `InvalidCacheBackendError` will be raised.

To provide thread-safety, a different instance of the cache backend will be returned for each thread.

`django.core.cache.`**`cache`**

As a shortcut, the default cache is available as `django.core.cache.cache`:

```
>>> from django.core.cache import cache
```

This object is equivalent to `caches['default']`.

Basic usage

The basic interface is `set(key, value, timeout)` and `get(key)`:

```
>>> cache.set('my_key', 'hello, world!', 30)
>>> cache.get('my_key')
'hello, world!'
```

`key` should be a `str` (or `unicode` on Python 2), and `value` can be any picklable Python object.

The `timeout` argument is optional and defaults to the `timeout` argument of the appropriate backend in the *CACHES* setting (explained above). It's the number of seconds the value should be stored in the cache. Passing in `None` for `timeout` will cache the value forever. A `timeout` of `0` won't cache the value.

If the object doesn't exist in the cache, `cache.get()` returns `None`:

```
>>> # Wait 30 seconds for 'my_key' to expire...
>>> cache.get('my_key')
None
```

We advise against storing the literal value `None` in the cache, because you won't be able to distinguish between your stored `None` value and a cache miss signified by a return value of `None`.

`cache.get()` can take a `default` argument. This specifies which value to return if the object doesn't exist in the cache:

```
>>> cache.get('my_key', 'has expired')
'has expired'
```

To add a key only if it doesn't already exist, use the `add()` method. It takes the same parameters as `set()`, but it will not attempt to update the cache if the key specified is already present:

```
>>> cache.set('add_key', 'Initial value')
>>> cache.add('add_key', 'New value')
>>> cache.get('add_key')
'Initial value'
```

If you need to know whether `add()` stored a value in the cache, you can check the return value. It will return `True` if the value was stored, `False` otherwise.

If you want to get a key's value or set a value if the key isn't in the cache, there is the `get_or_set()` method. It takes the same parameters as `get()` but the default is set as the new cache value for that key, rather than simply returned:

```
>>> cache.get('my_new_key')  # returns None
>>> cache.get_or_set('my_new_key', 'my new value', 100)
'my new value'
```

You can also pass any callable as a *default* value:

```
>>> import datetime
>>> cache.get_or_set('some-timestamp-key', datetime.datetime.now)
datetime.datetime(2014, 12, 11, 0, 15, 49, 457920)
```

The `get_or_set()` method was added.

There's also a `get_many()` interface that only hits the cache once. `get_many()` returns a dictionary with all the keys you asked for that actually exist in the cache (and haven't expired):

```
>>> cache.set('a', 1)
>>> cache.set('b', 2)
>>> cache.set('c', 3)
```

```
>>> cache.get_many(['a', 'b', 'c'])
{'a': 1, 'b': 2, 'c': 3}
```

To set multiple values more efficiently, use `set_many()` to pass a dictionary of key-value pairs:

```
>>> cache.set_many({'a': 1, 'b': 2, 'c': 3})
>>> cache.get_many(['a', 'b', 'c'])
{'a': 1, 'b': 2, 'c': 3}
```

Like `cache.set()`, `set_many()` takes an optional `timeout` parameter.

You can delete keys explicitly with `delete()`. This is an easy way of clearing the cache for a particular object:

```
>>> cache.delete('a')
```

If you want to clear a bunch of keys at once, `delete_many()` can take a list of keys to be cleared:

```
>>> cache.delete_many(['a', 'b', 'c'])
```

Finally, if you want to delete all the keys in the cache, use `cache.clear()`. Be careful with this; `clear()` will remove *everything* from the cache, not just the keys set by your application.

```
>>> cache.clear()
```

You can also increment or decrement a key that already exists using the `incr()` or `decr()` methods, respectively. By default, the existing cache value will incremented or decremented by 1. Other increment/decrement values can be specified by providing an argument to the increment/decrement call. A ValueError will be raised if you attempt to increment or decrement a nonexistent cache key.:

```
>>> cache.set('num', 1)
>>> cache.incr('num')
2
>>> cache.incr('num', 10)
12
>>> cache.decr('num')
11
>>> cache.decr('num', 5)
6
```

: `incr()`/`decr()` methods are not guaranteed to be atomic. On those backends that support atomic increment/decrement (most notably, the memcached backend), increment and decrement operations will be atomic. However, if the backend doesn't natively provide an increment/decrement operation, it will be implemented using a two-step retrieve/update.

You can close the connection to your cache with `close()` if implemented by the cache backend.

```
>>> cache.close()
```

: For caches that don't implement `close` methods it is a no-op.

Cache key prefixing

If you are sharing a cache instance between servers, or between your production and development environments, it's possible for data cached by one server to be used by another server. If the format of cached data is different between servers, this can lead to some very hard to diagnose problems.

To prevent this, Django provides the ability to prefix all cache keys used by a server. When a particular cache key is saved or retrieved, Django will automatically prefix the cache key with the value of the *KEY_PREFIX* cache setting.

By ensuring each Django instance has a different *KEY_PREFIX*, you can ensure that there will be no collisions in cache values.

Cache versioning

When you change running code that uses cached values, you may need to purge any existing cached values. The easiest way to do this is to flush the entire cache, but this can lead to the loss of cache values that are still valid and useful.

Django provides a better way to target individual cache values. Django's cache framework has a system-wide version identifier, specified using the *VERSION* cache setting. The value of this setting is automatically combined with the cache prefix and the user-provided cache key to obtain the final cache key.

By default, any key request will automatically include the site default cache key version. However, the primitive cache functions all include a `version` argument, so you can specify a particular cache key version to set or get. For example:

```
>>> # Set version 2 of a cache key
>>> cache.set('my_key', 'hello world!', version=2)
>>> # Get the default version (assuming version=1)
>>> cache.get('my_key')
None
>>> # Get version 2 of the same key
>>> cache.get('my_key', version=2)
'hello world!'
```

The version of a specific key can be incremented and decremented using the `incr_version()` and `decr_version()` methods. This enables specific keys to be bumped to a new version, leaving other keys unaffected. Continuing our previous example:

```
>>> # Increment the version of 'my_key'
>>> cache.incr_version('my_key')
>>> # The default version still isn't available
>>> cache.get('my_key')
None
# Version 2 isn't available, either
>>> cache.get('my_key', version=2)
None
>>> # But version 3 *is* available
>>> cache.get('my_key', version=3)
'hello world!'
```

Cache key transformation

As described in the previous two sections, the cache key provided by a user is not used verbatim – it is combined with the cache prefix and key version to provide a final cache key. By default, the three parts are joined using colons to produce a final string:

```
def make_key(key, key_prefix, version):
    return ':'.join([key_prefix, str(version), key])
```

If you want to combine the parts in different ways, or apply other processing to the final key (e.g., taking a hash digest of the key parts), you can provide a custom key function.

The *KEY_FUNCTION* cache setting specifies a dotted-path to a function matching the prototype of `make_key()` above. If provided, this custom key function will be used instead of the default key combining function.

Cache key warnings

Memcached, the most commonly-used production cache backend, does not allow cache keys longer than 250 characters or containing whitespace or control characters, and using such keys will cause an exception. To encourage cache-portable code and minimize unpleasant surprises, the other built-in cache backends issue a warning (`django.core.cache.backends.base.CacheKeyWarning`) if a key is used that would cause an error on memcached.

If you are using a production backend that can accept a wider range of keys (a custom backend, or one of the non-memcached built-in backends), and want to use this wider range without warnings, you can silence `CacheKeyWarning` with this code in the `management` module of one of your *INSTALLED_APPS*:

```
import warnings

from django.core.cache import CacheKeyWarning

warnings.simplefilter("ignore", CacheKeyWarning)
```

If you want to instead provide custom key validation logic for one of the built-in backends, you can subclass it, override just the `validate_key` method, and follow the instructions for *using a custom cache backend*. For instance, to do this for the `locmem` backend, put this code in a module:

```
from django.core.cache.backends.locmem import LocMemCache

class CustomLocMemCache(LocMemCache):
    def validate_key(self, key):
        """Custom validation, raising exceptions or warnings as needed."""
        ...
```

...and use the dotted Python path to this class in the *BACKEND* portion of your *CACHES* setting.

3.11.6 Downstream caches

So far, this document has focused on caching your *own* data. But another type of caching is relevant to Web development, too: caching performed by "downstream" caches. These are systems that cache pages for users even before the request reaches your website.

Here are a few examples of downstream caches:

- Your ISP may cache certain pages, so if you requested a page from https://example.com/, your ISP would send you the page without having to access example.com directly. The maintainers of example.com have no knowledge of this caching; the ISP sits between example.com and your Web browser, handling all of the caching transparently.
- Your Django website may sit behind a *proxy cache*, such as Squid Web Proxy Cache (http://www.squid-cache.org/), that caches pages for performance. In this case, each request first would be handled by the proxy, and it would be passed to your application only if needed.
- Your Web browser caches pages, too. If a Web page sends out the appropriate headers, your browser will use the local cached copy for subsequent requests to that page, without even contacting the Web page again to see whether it has changed.

Downstream caching is a nice efficiency boost, but there's a danger to it: Many Web pages' contents differ based on authentication and a host of other variables, and cache systems that blindly save pages based purely on URLs could expose incorrect or sensitive data to subsequent visitors to those pages.

For example, say you operate a Web email system, and the contents of the "inbox" page obviously depend on which user is logged in. If an ISP blindly cached your site, then the first user who logged in through that ISP would have their user-specific inbox page cached for subsequent visitors to the site. That's not cool.

Fortunately, HTTP provides a solution to this problem. A number of HTTP headers exist to instruct downstream caches to differ their cache contents depending on designated variables, and to tell caching mechanisms not to cache particular pages. We'll look at some of these headers in the sections that follow.

3.11.7 Using `Vary` headers

The `Vary` header defines which request headers a cache mechanism should take into account when building its cache key. For example, if the contents of a Web page depend on a user's language preference, the page is said to "vary on language."

By default, Django's cache system creates its cache keys using the requested fully-qualified URL – e.g., `"https://www.example.com/stories/2005/?order_by=author"`. This means every request to that URL will use the same cached version, regardless of user-agent differences such as cookies or language preferences. However, if this page produces different content based on some difference in request headers – such as a cookie, or a language, or a user-agent – you'll need to use the `Vary` header to tell caching mechanisms that the page output depends on those things.

To do this in Django, use the convenient *`django.views.decorators.vary.vary_on_headers()`* view decorator, like so:

```
from django.views.decorators.vary import vary_on_headers

@vary_on_headers('User-Agent')
def my_view(request):
    ...
```

In this case, a caching mechanism (such as Django's own cache middleware) will cache a separate version of the page for each unique user-agent.

The advantage to using the `vary_on_headers` decorator rather than manually setting the `Vary` header (using something like `response['Vary'] = 'user-agent'`) is that the decorator *adds* to the `Vary` header (which may already exist), rather than setting it from scratch and potentially overriding anything that was already in there.

You can pass multiple headers to `vary_on_headers()`:

```
@vary_on_headers('User-Agent', 'Cookie')
def my_view(request):
    ...
```

This tells downstream caches to vary on *both*, which means each combination of user-agent and cookie will get its own cache value. For example, a request with the user-agent `Mozilla` and the cookie value `foo=bar` will be considered different from a request with the user-agent `Mozilla` and the cookie value `foo=ham`.

Because varying on cookie is so common, there's a *`django.views.decorators.vary.vary_on_cookie()`* decorator. These two views are equivalent:

```
@vary_on_cookie
def my_view(request):
    ...

@vary_on_headers('Cookie')
def my_view(request):
    ...
```

The headers you pass to `vary_on_headers` are not case sensitive; `"User-Agent"` is the same thing as `"user-agent"`.

You can also use a helper function, *`django.utils.cache.patch_vary_headers()`*, directly. This function sets, or adds to, the `Vary header`. For example:

```
from django.shortcuts import render
from django.utils.cache import patch_vary_headers

def my_view(request):
    ...
    response = render(request, 'template_name', context)
    patch_vary_headers(response, ['Cookie'])
    return response
```

`patch_vary_headers` takes an *`HttpResponse`* instance as its first argument and a list/tuple of case-insensitive header names as its second argument.

For more on Vary headers, see the **official Vary spec**.

3.11.8 Controlling cache: Using other headers

Other problems with caching are the privacy of data and the question of where data should be stored in a cascade of caches.

A user usually faces two kinds of caches: their own browser cache (a private cache) and their provider's cache (a public cache). A public cache is used by multiple users and controlled by someone else. This poses problems with sensitive data–you don't want, say, your bank account number stored in a public cache. So Web applications need a way to tell caches which data is private and which is public.

The solution is to indicate a page's cache should be "private." To do this in Django, use the *`cache_control()`* view decorator. Example:

```
from django.views.decorators.cache import cache_control

@cache_control(private=True)
def my_view(request):
    ...
```

This decorator takes care of sending out the appropriate HTTP header behind the scenes.

Note that the cache control settings "private" and "public" are mutually exclusive. The decorator ensures that the "public" directive is removed if "private" should be set (and vice versa). An example use of the two directives would be a blog site that offers both private and public entries. Public entries may be cached on any shared cache. The following code uses *`patch_cache_control()`*, the manual way to modify the cache control header (it is internally called by the *`cache_control()`* decorator):

```
from django.views.decorators.cache import patch_cache_control
from django.views.decorators.vary import vary_on_cookie

@vary_on_cookie
def list_blog_entries_view(request):
    if request.user.is_anonymous:
        response = render_only_public_entries()
        patch_cache_control(response, public=True)
    else:
        response = render_private_and_public_entries(request.user)
        patch_cache_control(response, private=True)
```

```
    return response
```

You can control downstream caches in other ways as well (see **RFC 7234** for details on HTTP caching). For example, even if you don't use Django's server-side cache framework, you can still tell clients to cache a view for a certain amount of time with the **max-age** directive:

```
from django.views.decorators.cache import cache_control

@cache_control(max_age=3600)
def my_view(request):
    ...
```

(If you *do* use the caching middleware, it already sets the `max-age` with the value of the *CACHE_MIDDLEWARE_SECONDS* setting. In that case, the custom `max_age` from the *cache_control()* decorator will take precedence, and the header values will be merged correctly.)

Any valid `Cache-Control` response directive is valid in `cache_control()`. Here are some more examples:

- `no_transform=True`
- `must_revalidate=True`
- `stale_while_revalidate=num_seconds`

The full list of known directives can be found in the IANA registry (note that not all of them apply to responses).

If you want to use headers to disable caching altogether, *never_cache()* is a view decorator that adds headers to ensure the response won't be cached by browsers or other caches. Example:

```
from django.views.decorators.cache import never_cache

@never_cache
def myview(request):
    ...
```

3.11.9 Order of MIDDLEWARE

If you use caching middleware, it's important to put each half in the right place within the *MIDDLEWARE* setting. That's because the cache middleware needs to know which headers by which to vary the cache storage. Middleware always adds something to the `Vary` response header when it can.

`UpdateCacheMiddleware` runs during the response phase, where middleware is run in reverse order, so an item at the top of the list runs *last* during the response phase. Thus, you need to make sure that `UpdateCacheMiddleware` appears *before* any other middleware that might add something to the `Vary` header. The following middleware modules do so:

- `SessionMiddleware` adds `Cookie`
- `GZipMiddleware` adds `Accept-Encoding`
- `LocaleMiddleware` adds `Accept-Language`

`FetchFromCacheMiddleware`, on the other hand, runs during the request phase, where middleware is applied first-to-last, so an item at the top of the list runs *first* during the request phase. The `FetchFromCacheMiddleware` also needs to run after other middleware updates the `Vary` header, so `FetchFromCacheMiddleware` must be *after* any item that does so.

3.12 Conditional View Processing

HTTP clients can send a number of headers to tell the server about copies of a resource that they have already seen. This is commonly used when retrieving a Web page (using an HTTP `GET` request) to avoid sending all the data for something the client has already retrieved. However, the same headers can be used for all HTTP methods (`POST`, `PUT`, `DELETE`, etc.).

For each page (response) that Django sends back from a view, it might provide two HTTP headers: the `ETag` header and the `Last-Modified` header. These headers are optional on HTTP responses. They can be set by your view function, or you can rely on the *`CommonMiddleware`* middleware to set the `ETag` header.

When the client next requests the same resource, it might send along a header such as either If-modified-since or If-unmodified-since, containing the date of the last modification time it was sent, or either If-match or If-none-match, containing the last `ETag` it was sent. If the current version of the page matches the `ETag` sent by the client, or if the resource has not been modified, a 304 status code can be sent back, instead of a full response, telling the client that nothing has changed. Depending on the header, if the page has been modified or does not match the `ETag` sent by the client, a 412 status code (Precondition Failed) may be returned.

When you need more fine-grained control you may use per-view conditional processing functions.

3.12.1 The `condition` decorator

Sometimes (in fact, quite often) you can create functions to rapidly compute the ETag value or the last-modified time for a resource, **without** needing to do all the computations needed to construct the full view. Django can then use these functions to provide an "early bailout" option for the view processing. Telling the client that the content has not been modified since the last request, perhaps.

These two functions are passed as parameters to the `django.views.decorators.http.condition` decorator. This decorator uses the two functions (you only need to supply one, if you can't compute both quantities easily and quickly) to work out if the headers in the HTTP request match those on the resource. If they don't match, a new copy of the resource must be computed and your normal view is called.

The `condition` decorator's signature looks like this:

```
condition(etag_func=None, last_modified_func=None)
```

The two functions, to compute the ETag and the last modified time, will be passed the incoming `request` object and the same parameters, in the same order, as the view function they are helping to wrap. The function passed `last_modified_func` should return a standard datetime value specifying the last time the resource was modified, or `None` if the resource doesn't exist. The function passed to the `etag` decorator should return a string representing the ETag for the resource, or `None` if it doesn't exist.

Using this feature usefully is probably best explained with an example. Suppose you have this pair of models, representing a simple blog system:

```
import datetime
from django.db import models

class Blog(models.Model):
    ...

class Entry(models.Model):
    blog = models.ForeignKey(Blog)
    published = models.DateTimeField(default=datetime.datetime.now)
    ...
```

If the front page, displaying the latest blog entries, only changes when you add a new blog entry, you can compute the last modified time very quickly. You need the latest `published` date for every entry associated with that blog. One way to do this would be:

```
def latest_entry(request, blog_id):
    return Entry.objects.filter(blog=blog_id).latest("published").published
```

You can then use this function to provide early detection of an unchanged page for your front page view:

```
from django.views.decorators.http import condition

@condition(last_modified_func=latest_entry)
def front_page(request, blog_id):
    ...
```

Be careful with the order of decorators

When `condition()` returns a conditional response, any decorators below it will be skipped and won't apply to the response. Therefore, any decorators that need to apply to both the regular view response and a conditional response must be above `condition()`. In particular, *`vary_on_cookie()`*, *`vary_on_headers()`*, and *`cache_control()`* should come first because **RFC 7232** requires that the headers they set be present on 304 responses.

3.12.2 Shortcuts for only computing one value

As a general rule, if you can provide functions to compute *both* the ETag and the last modified time, you should do so. You don't know which headers any given HTTP client will send you, so be prepared to handle both. However, sometimes only one value is easy to compute and Django provides decorators that handle only ETag or only last-modified computations.

The `django.views.decorators.http.etag` and `django.views.decorators.http.last_modified` decorators are passed the same type of functions as the `condition` decorator. Their signatures are:

```
etag(etag_func)
last_modified(last_modified_func)
```

We could write the earlier example, which only uses a last-modified function, using one of these decorators:

```
@last_modified(latest_entry)
def front_page(request, blog_id):
    ...
```

...or:

```
def front_page(request, blog_id):
    ...
front_page = last_modified(latest_entry)(front_page)
```

Use `condition` when testing both conditions

It might look nicer to some people to try and chain the `etag` and `last_modified` decorators if you want to test both preconditions. However, this would lead to incorrect behavior.

```
# Bad code. Don't do this!
@etag(etag_func)
@last_modified(last_modified_func)
def my_view(request):
    # ...

# End of bad code.
```

The first decorator doesn't know anything about the second and might answer that the response is not modified even if the second decorators would determine otherwise. The `condition` decorator uses both callback functions simultaneously to work out the right action to take.

3.12.3 Using the decorators with other HTTP methods

The `condition` decorator is useful for more than only `GET` and `HEAD` requests (`HEAD` requests are the same as `GET` in this situation). It can also be used to provide checking for `POST`, `PUT` and `DELETE` requests. In these situations, the idea isn't to return a "not modified" response, but to tell the client that the resource they are trying to change has been altered in the meantime.

For example, consider the following exchange between the client and server:

1. Client requests `/foo/`.
2. Server responds with some content with an ETag of `"abcd1234"`.
3. Client sends an HTTP `PUT` request to `/foo/` to update the resource. It also sends an `If-Match: "abcd1234"` header to specify the version it is trying to update.
4. Server checks to see if the resource has changed, by computing the ETag the same way it does for a `GET` request (using the same function). If the resource *has* changed, it will return a 412 status code, meaning "precondition failed".
5. Client sends a `GET` request to `/foo/`, after receiving a 412 response, to retrieve an updated version of the content before updating it.

The important thing this example shows is that the same functions can be used to compute the ETag and last modification values in all situations. In fact, you **should** use the same functions, so that the same values are returned every time.

3.12.4 Comparison with middleware conditional processing

You may notice that Django already provides simple and straightforward conditional `GET` handling via the *`django.middleware.http.ConditionalGetMiddleware`* and *`CommonMiddleware`*. While certainly being easy to use and suitable for many situations, those pieces of middleware functionality have limitations for advanced usage:

- They are applied globally to all views in your project
- They don't save you from generating the response itself, which may be expensive
- They are only appropriate for HTTP `GET` requests.

You should choose the most appropriate tool for your particular problem here. If you have a way to compute ETags and modification times quickly and if some view takes a while to generate the content, you should consider using the `condition` decorator described in this document. If everything already runs fairly quickly, stick to using the middleware and the amount of network traffic sent back to the clients will still be reduced if the view hasn't changed.

3.13 Cryptographic signing

The golden rule of Web application security is to never trust data from untrusted sources. Sometimes it can be useful to pass data through an untrusted medium. Cryptographically signed values can be passed through an untrusted channel safe in the knowledge that any tampering will be detected.

Django provides both a low-level API for signing values and a high-level API for setting and reading signed cookies, one of the most common uses of signing in Web applications.

You may also find signing useful for the following:

- Generating "recover my account" URLs for sending to users who have lost their password.
- Ensuring data stored in hidden form fields has not been tampered with.
- Generating one-time secret URLs for allowing temporary access to a protected resource, for example a downloadable file that a user has paid for.

3.13.1 Protecting the SECRET_KEY

When you create a new Django project using *startproject*, the `settings.py` file is generated automatically and gets a random *SECRET_KEY* value. This value is the key to securing signed data – it is vital you keep this secure, or attackers could use it to generate their own signed values.

3.13.2 Using the low-level API

Django's signing methods live in the `django.core.signing` module. To sign a value, first instantiate a `Signer` instance:

```
>>> from django.core.signing import Signer
>>> signer = Signer()
>>> value = signer.sign('My string')
>>> value
'My string:GdMGD6HNQ_qdgxYP8yBZAdAIV1w'
```

The signature is appended to the end of the string, following the colon. You can retrieve the original value using the `unsign` method:

```
>>> original = signer.unsign(value)
>>> original
'My string'
```

If the signature or value have been altered in any way, a `django.core.signing.BadSignature` exception will be raised:

```
>>> from django.core import signing
>>> value += 'm'
>>> try:
...     original = signer.unsign(value)
... except signing.BadSignature:
...     print("Tampering detected!")
```

By default, the `Signer` class uses the *SECRET_KEY* setting to generate signatures. You can use a different secret by passing it to the `Signer` constructor:

```
>>> signer = Signer('my-other-secret')
>>> value = signer.sign('My string')
>>> value
'My string:EkfQJafvGyiofrdGnuthdxImIJw'
```

class **Signer**(*key=None*, *sep=':'*, *salt=None*)

Returns a signer which uses `key` to generate signatures and `sep` to separate values. `sep` cannot be in the URL safe base64 alphabet. This alphabet contains alphanumeric characters, hyphens, and underscores.

Using the `salt` argument

If you do not wish for every occurrence of a particular string to have the same signature hash, you can use the optional `salt` argument to the `Signer` class. Using a salt will seed the signing hash function with both the salt and your *SECRET_KEY*:

```
>>> signer = Signer()
>>> signer.sign('My string')
'My string:GdMGD6HNQ_qdgxYP8yBZAdAIV1w'
>>> signer = Signer(salt='extra')
>>> signer.sign('My string')
'My string:Ee7vGi-ING6n02gkcJ-QLHg6vFw'
>>> signer.unsign('My string:Ee7vGi-ING6n02gkcJ-QLHg6vFw')
'My string'
```

Using salt in this way puts the different signatures into different namespaces. A signature that comes from one namespace (a particular salt value) cannot be used to validate the same plaintext string in a different namespace that is using a different salt setting. The result is to prevent an attacker from using a signed string generated in one place in the code as input to another piece of code that is generating (and verifying) signatures using a different salt.

Unlike your *SECRET_KEY*, your salt argument does not need to stay secret.

Verifying timestamped values

`TimestampSigner` is a subclass of *Signer* that appends a signed timestamp to the value. This allows you to confirm that a signed value was created within a specified period of time:

```
>>> from datetime import timedelta
>>> from django.core.signing import TimestampSigner
>>> signer = TimestampSigner()
>>> value = signer.sign('hello')
>>> value
'hello:1NMg5H:oPVuCqlJWmChm1rA2lyTUtelC-c'
>>> signer.unsign(value)
'hello'
>>> signer.unsign(value, max_age=10)
...
SignatureExpired: Signature age 15.5289158821 > 10 seconds
>>> signer.unsign(value, max_age=20)
'hello'
>>> signer.unsign(value, max_age=timedelta(seconds=20))
'hello'
```

class **TimestampSigner**(*key=None*, *sep=':'*, *salt=None*)

sign(*value*)

Sign `value` and append current timestamp to it.

unsign(*value*, *max_age=None*)
Checks if `value` was signed less than `max_age` seconds ago, otherwise raises `SignatureExpired`. The `max_age` parameter can accept an integer or a `datetime.timedelta` object.

Protecting complex data structures

If you wish to protect a list, tuple or dictionary you can do so using the signing module's `dumps` and `loads` functions. These imitate Python's pickle module, but use JSON serialization under the hood. JSON ensures that even if your *SECRET_KEY* is stolen an attacker will not be able to execute arbitrary commands by exploiting the pickle format:

```
>>> from django.core import signing
>>> value = signing.dumps({"foo": "bar"})
>>> value
'eyJmb28iOiJiYXIifQ:1NMg1b:zGcDE4-TCkaeGzLeW9UQwZesciI'
>>> signing.loads(value)
{'foo': 'bar'}
```

Because of the nature of JSON (there is no native distinction between lists and tuples) if you pass in a tuple, you will get a list from `signing.loads(object)`:

```
>>> from django.core import signing
>>> value = signing.dumps(('a','b','c'))
>>> signing.loads(value)
['a', 'b', 'c']
```

dumps(*obj*, *key=None*, *salt='django.core.signing'*, *compress=False*)
Returns URL-safe, sha1 signed base64 compressed JSON string. Serialized object is signed using *TimestampSigner*.

loads(*string*, *key=None*, *salt='django.core.signing'*, *max_age=None*)
Reverse of `dumps()`, raises `BadSignature` if signature fails. Checks `max_age` (in seconds) if given.

3.14 Sending email

Although Python makes sending email relatively easy via the `smtplib` module, Django provides a couple of light wrappers over it. These wrappers are provided to make sending email extra quick, to make it easy to test email sending during development, and to provide support for platforms that can't use SMTP.

The code lives in the `django.core.mail` module.

3.14.1 Quick example

In two lines:

```
from django.core.mail import send_mail

send_mail(
    'Subject here',
    'Here is the message.',
    'from@example.com',
    ['to@example.com'],
    fail_silently=False,
)
```

Mail is sent using the SMTP host and port specified in the *EMAIL_HOST* and *EMAIL_PORT* settings. The *EMAIL_HOST_USER* and *EMAIL_HOST_PASSWORD* settings, if set, are used to authenticate to the SMTP server, and the *EMAIL_USE_TLS* and *EMAIL_USE_SSL* settings control whether a secure connection is used.

: The character set of email sent with `django.core.mail` will be set to the value of your *DEFAULT_CHARSET* setting.

3.14.2 send_mail()

send_mail(*subject*, *message*, *from_email*, *recipient_list*, *fail_silently=False*, *auth_user=None*, *auth_password=None*, *connection=None*, *html_message=None*)

The simplest way to send email is using `django.core.mail.send_mail()`.

The `subject`, `message`, `from_email` and `recipient_list` parameters are required.

- `subject`: A string.
- `message`: A string.
- `from_email`: A string.
- `recipient_list`: A list of strings, each an email address. Each member of `recipient_list` will see the other recipients in the "To:" field of the email message.
- `fail_silently`: A boolean. If it's `False`, `send_mail` will raise an `smtplib.SMTPException`. See the `smtplib` docs for a list of possible exceptions, all of which are subclasses of `SMTPException`.
- `auth_user`: The optional username to use to authenticate to the SMTP server. If this isn't provided, Django will use the value of the *EMAIL_HOST_USER* setting.
- `auth_password`: The optional password to use to authenticate to the SMTP server. If this isn't provided, Django will use the value of the *EMAIL_HOST_PASSWORD* setting.
- `connection`: The optional email backend to use to send the mail. If unspecified, an instance of the default backend will be used. See the documentation on *Email backends* for more details.
- `html_message`: If `html_message` is provided, the resulting email will be a *multipart/alternative* email with `message` as the *text/plain* content type and `html_message` as the *text/html* content type.

The return value will be the number of successfully delivered messages (which can be `0` or `1` since it can only send one message).

3.14.3 send_mass_mail()

send_mass_mail(*datatuple*, *fail_silently=False*, *auth_user=None*, *auth_password=None*, *connection=None*)

`django.core.mail.send_mass_mail()` is intended to handle mass emailing.

`datatuple` is a tuple in which each element is in this format:

```
(subject, message, from_email, recipient_list)
```

`fail_silently`, `auth_user` and `auth_password` have the same functions as in *send_mail()*.

Each separate element of `datatuple` results in a separate email message. As in *send_mail()*, recipients in the same `recipient_list` will all see the other addresses in the email messages' "To:" field.

For example, the following code would send two different messages to two different sets of recipients; however, only one connection to the mail server would be opened:

```
message1 = ('Subject here', 'Here is the message', 'from@example.com', ['first@example.com', 'other@e
message2 = ('Another Subject', 'Here is another message', 'from@example.com', ['second@test.com'])
send_mass_mail((message1, message2), fail_silently=False)
```

The return value will be the number of successfully delivered messages.

`send_mass_mail()` vs. `send_mail()`

The main difference between *`send_mass_mail()`* and *`send_mail()`* is that *`send_mail()`* opens a connection to the mail server each time it's executed, while *`send_mass_mail()`* uses a single connection for all of its messages. This makes *`send_mass_mail()`* slightly more efficient.

3.14.4 `mail_admins()`

`mail_admins`(*subject*, *message*, *fail_silently=False*, *connection=None*, *html_message=None*)

`django.core.mail.mail_admins()` is a shortcut for sending an email to the site admins, as defined in the *`ADMINS`* setting.

`mail_admins()` prefixes the subject with the value of the *`EMAIL_SUBJECT_PREFIX`* setting, which is `"[Django] "` by default.

The "From:" header of the email will be the value of the *`SERVER_EMAIL`* setting.

This method exists for convenience and readability.

If `html_message` is provided, the resulting email will be a *`multipart/alternative`* email with `message` as the *`text/plain`* content type and `html_message` as the *`text/html`* content type.

3.14.5 `mail_managers()`

`mail_managers`(*subject*, *message*, *fail_silently=False*, *connection=None*, *html_message=None*)

`django.core.mail.mail_managers()` is just like `mail_admins()`, except it sends an email to the site managers, as defined in the *`MANAGERS`* setting.

3.14.6 Examples

This sends a single email to john@example.com and jane@example.com, with them both appearing in the "To:":

```
send_mail(
    'Subject',
    'Message.',
    'from@example.com',
    ['john@example.com', 'jane@example.com'],
)
```

This sends a message to john@example.com and jane@example.com, with them both receiving a separate email:

```
datatuple = (
    ('Subject', 'Message.', 'from@example.com', ['john@example.com']),
    ('Subject', 'Message.', 'from@example.com', ['jane@example.com']),
```

```
)
send_mass_mail(datatuple)
```

3.14.7 Preventing header injection

Header injection is a security exploit in which an attacker inserts extra email headers to control the "To:" and "From:" in email messages that your scripts generate.

The Django email functions outlined above all protect against header injection by forbidding newlines in header values. If any `subject`, `from_email` or `recipient_list` contains a newline (in either Unix, Windows or Mac style), the email function (e.g. *`send_mail()`*) will raise `django.core.mail.BadHeaderError` (a subclass of `ValueError`) and, hence, will not send the email. It's your responsibility to validate all data before passing it to the email functions.

If a `message` contains headers at the start of the string, the headers will simply be printed as the first bit of the email message.

Here's an example view that takes a `subject`, `message` and `from_email` from the request's POST data, sends that to admin@example.com and redirects to "/contact/thanks/" when it's done:

```
from django.core.mail import send_mail, BadHeaderError
from django.http import HttpResponse, HttpResponseRedirect

def send_email(request):
    subject = request.POST.get('subject', '')
    message = request.POST.get('message', '')
    from_email = request.POST.get('from_email', '')
    if subject and message and from_email:
        try:
            send_mail(subject, message, from_email, ['admin@example.com'])
        except BadHeaderError:
            return HttpResponse('Invalid header found.')
        return HttpResponseRedirect('/contact/thanks/')
    else:
        # In reality we'd use a form class
        # to get proper validation errors.
        return HttpResponse('Make sure all fields are entered and valid.')
```

3.14.8 The `EmailMessage` class

Django's *`send_mail()`* and *`send_mass_mail()`* functions are actually thin wrappers that make use of the *`EmailMessage`* class.

Not all features of the *`EmailMessage`* class are available through the *`send_mail()`* and related wrapper functions. If you wish to use advanced features, such as BCC'ed recipients, file attachments, or multi-part email, you'll need to create *`EmailMessage`* instances directly.

: This is a design feature. *`send_mail()`* and related functions were originally the only interface Django provided. However, the list of parameters they accepted was slowly growing over time. It made sense to move to a more object-oriented design for email messages and retain the original functions only for backwards compatibility.

`EmailMessage` is responsible for creating the email message itself. The *email backend* is then responsible for sending the email.

For convenience, *EmailMessage* provides a simple `send()` method for sending a single email. If you need to send multiple messages, the email backend API *provides an alternative*.

`EmailMessage` Objects

class `EmailMessage`

The *EmailMessage* class is initialized with the following parameters (in the given order, if positional arguments are used). All parameters are optional and can be set at any time prior to calling the `send()` method.

- `subject`: The subject line of the email.
- `body`: The body text. This should be a plain text message.
- `from_email`: The sender's address. Both `fred@example.com` and `Fred <fred@example.com>` forms are legal. If omitted, the *DEFAULT_FROM_EMAIL* setting is used.
- `to`: A list or tuple of recipient addresses.
- `bcc`: A list or tuple of addresses used in the "Bcc" header when sending the email.
- `connection`: An email backend instance. Use this parameter if you want to use the same connection for multiple messages. If omitted, a new connection is created when `send()` is called.
- `attachments`: A list of attachments to put on the message. These can be either `email.MIMEBase.MIMEBase` instances, or `(filename, content, mimetype)` triples.
- `headers`: A dictionary of extra headers to put on the message. The keys are the header name, values are the header values. It's up to the caller to ensure header names and values are in the correct format for an email message. The corresponding attribute is `extra_headers`.
- `cc`: A list or tuple of recipient addresses used in the "Cc" header when sending the email.
- `reply_to`: A list or tuple of recipient addresses used in the "Reply-To" header when sending the email.

For example:

```
from django.core.mail import EmailMessage

email = EmailMessage(
    'Hello',
    'Body goes here',
    'from@example.com',
    ['to1@example.com', 'to2@example.com'],
    ['bcc@example.com'],
    reply_to=['another@example.com'],
    headers={'Message-ID': 'foo'},
)
```

The class has the following methods:

- `send(fail_silently=False)` sends the message. If a connection was specified when the email was constructed, that connection will be used. Otherwise, an instance of the default backend will be instantiated and used. If the keyword argument `fail_silently` is `True`, exceptions raised while sending the message will be quashed. An empty list of recipients will not raise an exception.
- `message()` constructs a `django.core.mail.SafeMIMEText` object (a subclass of Python's `email.MIMEText.MIMEText` class) or a `django.core.mail.SafeMIMEMultipart` object holding the message to be sent. If you ever need to extend the *EmailMessage* class, you'll probably want to override this method to put the content you want into the MIME object.

- `recipients()` returns a list of all the recipients of the message, whether they're recorded in the `to`, `cc` or `bcc` attributes. This is another method you might need to override when subclassing, because the SMTP server needs to be told the full list of recipients when the message is sent. If you add another way to specify recipients in your class, they need to be returned from this method as well.
- `attach()` creates a new file attachment and adds it to the message. There are two ways to call `attach()`:
  - You can pass it a single argument that is an `email.MIMEBase.MIMEBase` instance. This will be inserted directly into the resulting message.
  - Alternatively, you can pass `attach()` three arguments: `filename`, `content` and `mimetype`. `filename` is the name of the file attachment as it will appear in the email, `content` is the data that will be contained inside the attachment and `mimetype` is the optional MIME type for the attachment. If you omit `mimetype`, the MIME content type will be guessed from the filename of the attachment.

    For example:

```
message.attach('design.png', img_data, 'image/png')
```

    If you specify a `mimetype` of `message/rfc822`, it will also accept *`django.core.mail.EmailMessage`* and `email.message.Message`.

    In addition, `message/rfc822` attachments will no longer be base64-encoded in violation of **RFC 2046#section-5.2.1**, which can cause issues with displaying the attachments in Evolution and Thunderbird.
- `attach_file()` creates a new attachment using a file from your filesystem. Call it with the path of the file to attach and, optionally, the MIME type to use for the attachment. If the MIME type is omitted, it will be guessed from the filename. The simplest use would be:

```
message.attach_file('/images/weather_map.png')
```

Sending alternative content types

It can be useful to include multiple versions of the content in an email; the classic example is to send both text and HTML versions of a message. With Django's email library, you can do this using the `EmailMultiAlternatives` class. This subclass of *`EmailMessage`* has an `attach_alternative()` method for including extra versions of the message body in the email. All the other methods (including the class initialization) are inherited directly from *`EmailMessage`*.

To send a text and HTML combination, you could write:

```
from django.core.mail import EmailMultiAlternatives

subject, from_email, to = 'hello', 'from@example.com', 'to@example.com'
text_content = 'This is an important message.'
html_content = '<p>This is an <strong>important</strong> message.</p>'
msg = EmailMultiAlternatives(subject, text_content, from_email, [to])
msg.attach_alternative(html_content, "text/html")
msg.send()
```

By default, the MIME type of the `body` parameter in an *`EmailMessage`* is `"text/plain"`. It is good practice to leave this alone, because it guarantees that any recipient will be able to read the email, regardless of their mail client. However, if you are confident that your recipients can handle an alternative content type, you can use the `content_subtype` attribute on the *`EmailMessage`* class to change the main content type. The major type will always be `"text"`, but you can change the subtype. For example:

```
msg = EmailMessage(subject, html_content, from_email, [to])
msg.content_subtype = "html"  # Main content is now text/html
msg.send()
```

3.14.9 Email backends

The actual sending of an email is handled by the email backend.

The email backend class has the following methods:

- `open()` instantiates a long-lived email-sending connection.
- `close()` closes the current email-sending connection.
- `send_messages(email_messages)` sends a list of *`EmailMessage`* objects. If the connection is not open, this call will implicitly open the connection, and close the connection afterwards. If the connection is already open, it will be left open after mail has been sent.

It can also be used as a context manager, which will automatically call `open()` and `close()` as needed:

```
from django.core import mail

with mail.get_connection() as connection:
    mail.EmailMessage(
        subject1, body1, from1, [to1],
        connection=connection,
    ).send()
    mail.EmailMessage(
        subject2, body2, from2, [to2],
        connection=connection,
    ).send()
```

Obtaining an instance of an email backend

The *`get_connection()`* function in `django.core.mail` returns an instance of the email backend that you can use.

`get_connection`(*backend=None*, *fail_silently=False*, **args*, ***kwargs*)

By default, a call to `get_connection()` will return an instance of the email backend specified in *`EMAIL_BACKEND`*. If you specify the `backend` argument, an instance of that backend will be instantiated.

The `fail_silently` argument controls how the backend should handle errors. If `fail_silently` is True, exceptions during the email sending process will be silently ignored.

All other arguments are passed directly to the constructor of the email backend.

Django ships with several email sending backends. With the exception of the SMTP backend (which is the default), these backends are only useful during testing and development. If you have special email sending requirements, you can *write your own email backend*.

SMTP backend

class `backends.smtp.`**`EmailBackend`**(*host=None*, *port=None*, *username=None*, *password=None*, *use_tls=None*, *fail_silently=False*, *use_ssl=None*, *timeout=None*, *ssl_keyfile=None*, *ssl_certfile=None*, ***kwargs*)

This is the default backend. Email will be sent through a SMTP server.

The value for each argument is retrieved from the matching setting if the argument is `None`:

- `host`: *EMAIL_HOST*
- `port`: *EMAIL_PORT*
- `username`: *EMAIL_HOST_USER*
- `password`: *EMAIL_HOST_PASSWORD*
- `use_tls`: *EMAIL_USE_TLS*
- `use_ssl`: *EMAIL_USE_SSL*
- `timeout`: *EMAIL_TIMEOUT*
- `ssl_keyfile`: *EMAIL_SSL_KEYFILE*
- `ssl_certfile`: *EMAIL_SSL_CERTFILE*

The SMTP backend is the default configuration inherited by Django. If you want to specify it explicitly, put the following in your settings:

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
```

If unspecified, the default `timeout` will be the one provided by `socket.getdefaulttimeout()`, which defaults to `None` (no timeout).

Console backend

Instead of sending out real emails the console backend just writes the emails that would be sent to the standard output. By default, the console backend writes to `stdout`. You can use a different stream-like object by providing the `stream` keyword argument when constructing the connection.

To specify this backend, put the following in your settings:

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

This backend is not intended for use in production – it is provided as a convenience that can be used during development.

File backend

The file backend writes emails to a file. A new file is created for each new session that is opened on this backend. The directory to which the files are written is either taken from the *EMAIL_FILE_PATH* setting or from the `file_path` keyword when creating a connection with *get_connection()*.

To specify this backend, put the following in your settings:

```
EMAIL_BACKEND = 'django.core.mail.backends.filebased.EmailBackend'
EMAIL_FILE_PATH = '/tmp/app-messages' # change this to a proper location
```

This backend is not intended for use in production – it is provided as a convenience that can be used during development.

In-memory backend

The `'locmem'` backend stores messages in a special attribute of the `django.core.mail` module. The `outbox` attribute is created when the first message is sent. It's a list with an *EmailMessage* instance for each message that would be sent.

To specify this backend, put the following in your settings:

```
EMAIL_BACKEND = 'django.core.mail.backends.locmem.EmailBackend'
```

This backend is not intended for use in production – it is provided as a convenience that can be used during development and testing.

Dummy backend

As the name suggests the dummy backend does nothing with your messages. To specify this backend, put the following in your settings:

```
EMAIL_BACKEND = 'django.core.mail.backends.dummy.EmailBackend'
```

This backend is not intended for use in production – it is provided as a convenience that can be used during development.

Defining a custom email backend

If you need to change how emails are sent you can write your own email backend. The *EMAIL_BACKEND* setting in your settings file is then the Python import path for your backend class.

Custom email backends should subclass `BaseEmailBackend` that is located in the `django.core.mail.backends.base` module. A custom email backend must implement the `send_messages(email_messages)` method. This method receives a list of *EmailMessage* instances and returns the number of successfully delivered messages. If your backend has any concept of a persistent session or connection, you should also implement the `open()` and `close()` methods. Refer to `smtp.EmailBackend` for a reference implementation.

Sending multiple emails

Establishing and closing an SMTP connection (or any other network connection, for that matter) is an expensive process. If you have a lot of emails to send, it makes sense to reuse an SMTP connection, rather than creating and destroying a connection every time you want to send an email.

There are two ways you tell an email backend to reuse a connection.

Firstly, you can use the `send_messages()` method. `send_messages()` takes a list of *EmailMessage* instances (or subclasses), and sends them all using a single connection.

For example, if you have a function called `get_notification_email()` that returns a list of *EmailMessage* objects representing some periodic email you wish to send out, you could send these emails using a single call to send_messages:

```
from django.core import mail
connection = mail.get_connection()   # Use default email connection
messages = get_notification_email()
connection.send_messages(messages)
```

In this example, the call to `send_messages()` opens a connection on the backend, sends the list of messages, and then closes the connection again.

The second approach is to use the `open()` and `close()` methods on the email backend to manually control the connection. `send_messages()` will not manually open or close the connection if it is already open, so if you manually open the connection, you can control when it is closed. For example:

```
from django.core import mail
connection = mail.get_connection()

# Manually open the connection
connection.open()

# Construct an email message that uses the connection
email1 = mail.EmailMessage(
    'Hello',
    'Body goes here',
    'from@example.com',
    ['to1@example.com'],
    connection=connection,
)
email1.send() # Send the email

# Construct two more messages
email2 = mail.EmailMessage(
    'Hello',
    'Body goes here',
    'from@example.com',
    ['to2@example.com'],
)
email3 = mail.EmailMessage(
    'Hello',
    'Body goes here',
    'from@example.com',
    ['to3@example.com'],
)

# Send the two emails in a single call -
connection.send_messages([email2, email3])
# The connection was already open so send_messages() doesn't close it.
# We need to manually close the connection.
connection.close()
```

3.14.10 Configuring email for development

There are times when you do not want Django to send emails at all. For example, while developing a website, you probably don't want to send out thousands of emails – but you may want to validate that emails will be sent to the right people under the right conditions, and that those emails will contain the correct content.

The easiest way to configure email for local development is to use the *console* email backend. This backend redirects all email to stdout, allowing you to inspect the content of mail.

The *file* email backend can also be useful during development – this backend dumps the contents of every SMTP connection to a file that can be inspected at your leisure.

Another approach is to use a "dumb" SMTP server that receives the emails locally and displays them to the terminal, but does not actually send anything. Python has a built-in way to accomplish this with a single command:

```
python -m smtpd -n -c DebuggingServer localhost:1025
```

This command will start a simple SMTP server listening on port 1025 of localhost. This server simply prints to standard output all email headers and the email body. You then only need to set the *EMAIL_HOST* and *EMAIL_PORT* accordingly. For a more detailed discussion of SMTP server options, see the Python documentation for the `smtpd` module.

For information about unit-testing the sending of emails in your application, see the *Email services* section of the testing documentation.

3.15 Internationalization and localization

3.15.1 Translation

Overview

In order to make a Django project translatable, you have to add a minimal number of hooks to your Python code and templates. These hooks are called *translation strings*. They tell Django: "This text should be translated into the end user's language, if a translation for this text is available in that language." It's your responsibility to mark translatable strings; the system can only translate strings it knows about.

Django then provides utilities to extract the translation strings into a *message file*. This file is a convenient way for translators to provide the equivalent of the translation strings in the target language. Once the translators have filled in the message file, it must be compiled. This process relies on the GNU gettext toolset.

Once this is done, Django takes care of translating Web apps on the fly in each available language, according to users' language preferences.

Django's internationalization hooks are on by default, and that means there's a bit of i18n-related overhead in certain places of the framework. If you don't use internationalization, you should take the two seconds to set *USE_I18N = False* in your settings file. Then Django will make some optimizations so as not to load the internationalization machinery.

: There is also an independent but related *USE_L10N* setting that controls if Django should implement format localization. See Format localization for more details.

: Make sure you've activated translation for your project (the fastest way is to check if *MIDDLEWARE* includes *django.middleware.locale.LocaleMiddleware*). If you haven't yet, see *How Django discovers language preference*.

Internationalization: in Python code

Standard translation

Specify a translation string by using the function *ugettext()*. It's convention to import this as a shorter alias, _, to save typing.

: Python's standard library `gettext` module installs `_()` into the global namespace, as an alias for `gettext()`. In Django, we have chosen not to follow this practice, for a couple of reasons:

1. For international character set (Unicode) support, *ugettext()* is more useful than `gettext()`. Sometimes, you should be using *ugettext_lazy()* as the default translation method for a particular file. Without `_()` in the global namespace, the developer has to think about which is the most appropriate translation function.
2. The underscore character (_) is used to represent "the previous result" in Python's interactive shell and doctest tests. Installing a global `_()` function causes interference. Explicitly importing `ugettext()` as `_()` avoids this problem.

What functions may be aliased as _?

Because of how `xgettext` (used by *makemessages*) works, only functions that take a single string argument can be imported as _:

- *gettext()*
- *gettext_lazy()*
- *ugettext()*
- *ugettext_lazy()*

In this example, the text `"Welcome to my site."` is marked as a translation string:

```
from django.utils.translation import ugettext as _
from django.http import HttpResponse

def my_view(request):
    output = _("Welcome to my site.")
    return HttpResponse(output)
```

Obviously, you could code this without using the alias. This example is identical to the previous one:

```
from django.utils.translation import ugettext
from django.http import HttpResponse

def my_view(request):
    output = ugettext("Welcome to my site.")
    return HttpResponse(output)
```

Translation works on computed values. This example is identical to the previous two:

```
def my_view(request):
    words = ['Welcome', 'to', 'my', 'site.']
    output = _(' '.join(words))
    return HttpResponse(output)
```

Translation works on variables. Again, here's an identical example:

```
def my_view(request):
    sentence = 'Welcome to my site.'
    output = _(sentence)
    return HttpResponse(output)
```

(The caveat with using variables or computed values, as in the previous two examples, is that Django's translation-string-detecting utility, *django-admin makemessages*, won't be able to find these strings. More on *makemessages* later.)

The strings you pass to `_()` or `ugettext()` can take placeholders, specified with Python's standard named-string interpolation syntax. Example:

```
def my_view(request, m, d):
    output = _('Today is %(month)s %(day)s.') % {'month': m, 'day': d}
    return HttpResponse(output)
```

This technique lets language-specific translations reorder the placeholder text. For example, an English translation may be `"Today is November 26."`, while a Spanish translation may be `"Hoy es 26 de Noviembre."` – with the month and the day placeholders swapped.

For this reason, you should use named-string interpolation (e.g., `%(day)s`) instead of positional interpolation (e.g., `%s` or `%d`) whenever you have more than a single parameter. If you used positional interpolation, translations wouldn't be able to reorder placeholder text.

Comments for translators

If you would like to give translators hints about a translatable string, you can add a comment prefixed with the `Translators` keyword on the line preceding the string, e.g.:

```
def my_view(request):
    # Translators: This message appears on the home page only
    output = ugettext("Welcome to my site.")
```

The comment will then appear in the resulting `.po` file associated with the translatable construct located below it and should also be displayed by most translation tools.

: Just for completeness, this is the corresponding fragment of the resulting `.po` file:

```
#. Translators: This message appears on the home page only
# path/to/python/file.py:123
msgid "Welcome to my site."
msgstr ""
```

This also works in templates. See *Comments for translators in templates* for more details.

Marking strings as no-op

Use the function *`django.utils.translation.ugettext_noop()`* to mark a string as a translation string without translating it. The string is later translated from a variable.

Use this if you have constant strings that should be stored in the source language because they are exchanged over systems or users – such as strings in a database – but should be translated at the last possible point in time, such as when the string is presented to the user.

Pluralization

Use the function *`django.utils.translation.ungettext()`* to specify pluralized messages.

`ungettext` takes three arguments: the singular translation string, the plural translation string and the number of objects.

This function is useful when you need your Django application to be localizable to languages where the number and complexity of plural forms is greater than the two forms used in English ('object' for the singular and 'objects' for all the cases where `count` is different from one, irrespective of its value.)

For example:

```
from django.utils.translation import ungettext
from django.http import HttpResponse

def hello_world(request, count):
    page = ungettext(
        'there is %(count)d object',
        'there are %(count)d objects',
    count) % {
        'count': count,
    }
    return HttpResponse(page)
```

In this example the number of objects is passed to the translation languages as the `count` variable.

Note that pluralization is complicated and works differently in each language. Comparing `count` to 1 isn't always the correct rule. This code looks sophisticated, but will produce incorrect results for some languages:

```
from django.utils.translation import ungettext
from myapp.models import Report

count = Report.objects.count()
if count == 1:
    name = Report._meta.verbose_name
else:
    name = Report._meta.verbose_name_plural

text = ungettext(
    'There is %(count)d %(name)s available.',
    'There are %(count)d %(name)s available.',
    count
) % {
    'count': count,
    'name': name
}
```

Don't try to implement your own singular-or-plural logic, it won't be correct. In a case like this, consider something like the following:

```
text = ungettext(
    'There is %(count)d %(name)s object available.',
    'There are %(count)d %(name)s objects available.',
    count
) % {
    'count': count,
    'name': Report._meta.verbose_name,
}
```

: When using `ungettext()`, make sure you use a single name for every extrapolated variable included in the literal. In the examples above, note how we used the `name` Python variable in both translation strings. This example, besides being incorrect in some languages as noted above, would fail:

```
text = ungettext(
    'There is %(count)d %(name)s available.',
    'There are %(count)d %(plural_name)s available.',
    count
) % {
    'count': Report.objects.count(),
    'name': Report._meta.verbose_name,
```

```
    'plural_name': Report._meta.verbose_name_plural
}
```

You would get an error when running *django-admin compilemessages*:

```
a format specification for argument 'name', as in 'msgstr[0]', doesn't exist in 'msgid'
```

: Plural form and po files

Django does not support custom plural equations in po files. As all translation catalogs are merged, only the plural form for the main Django po file (in `django/conf/locale/<lang_code>/LC_MESSAGES/django.po`) is considered. Plural forms in all other po files are ignored. Therefore, you should not use different plural equations in your project or application po files.

Contextual markers

Sometimes words have several meanings, such as `"May"` in English, which refers to a month name and to a verb. To enable translators to translate these words correctly in different contexts, you can use the *django.utils.translation.pgettext()* function, or the *django.utils.translation.npgettext()* function if the string needs pluralization. Both take a context string as the first variable.

In the resulting `.po` file, the string will then appear as often as there are different contextual markers for the same string (the context will appear on the `msgctxt` line), allowing the translator to give a different translation for each of them.

For example:

```
from django.utils.translation import pgettext

month = pgettext("month name", "May")
```

or:

```
from django.db import models
from django.utils.translation import pgettext_lazy

class MyThing(models.Model):
    name = models.CharField(help_text=pgettext_lazy(
        'help text for MyThing model', 'This is the help text'))
```

will appear in the `.po` file as:

```
msgctxt "month name"
msgid "May"
msgstr ""
```

Contextual markers are also supported by the *trans* and *blocktrans* template tags.

Lazy translation

Use the lazy versions of translation functions in *django.utils.translation* (easily recognizable by the `lazy` suffix in their names) to translate strings lazily – when the value is accessed rather than when they're called.

These functions store a lazy reference to the string – not the actual translation. The translation itself will be done when the string is used in a string context, such as in template rendering.

This is essential when calls to these functions are located in code paths that are executed at module load time.

This is something that can easily happen when defining models, forms and model forms, because Django implements these such that their fields are actually class-level attributes. For that reason, make sure to use lazy translations in the following cases:

Model fields and relationships `verbose_name` and `help_text` option values For example, to translate the help text of the *name* field in the following model, do the following:

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

class MyThing(models.Model):
    name = models.CharField(help_text=_('This is the help text'))
```

You can mark names of *ForeignKey*, *ManyToManyField* or *OneToOneField* relationship as translatable by using their *verbose_name* options:

```
class MyThing(models.Model):
    kind = models.ForeignKey(
        ThingKind,
        on_delete=models.CASCADE,
        related_name='kinds',
        verbose_name=_('kind'),
    )
```

Just like you would do in *verbose_name* you should provide a lowercase verbose name text for the relation as Django will automatically titlecase it when required.

Model verbose names values It is recommended to always provide explicit *verbose_name* and *verbose_name_plural* options rather than relying on the fallback English-centric and somewhat naïve determination of verbose names Django performs by looking at the model's class name:

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

class MyThing(models.Model):
    name = models.CharField(_('name'), help_text=_('This is the help text'))

    class Meta:
        verbose_name = _('my thing')
        verbose_name_plural = _('my things')
```

Model methods `short_description` attribute values For model methods, you can provide translations to Django and the admin site with the `short_description` attribute:

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

class MyThing(models.Model):
    kind = models.ForeignKey(
        ThingKind,
        on_delete=models.CASCADE,
        related_name='kinds',
```

```
        verbose_name=_('kind'),
    )

    def is_mouse(self):
        return self.kind.type == MOUSE_TYPE
    is_mouse.short_description = _('Is it a mouse?')
```

Working with lazy translation objects

The result of a `ugettext_lazy()` call can be used wherever you would use a unicode string (a `str` object) in other Django code, but it may not work with arbitrary Python code. For example, the following won't work because the requests library doesn't handle `ugettext_lazy` objects:

```
body = ugettext_lazy("I \u2764 Django")  # (unicode :heart:)
requests.post('https://example.com/send', data={'body': body})
```

You can avoid such problems by casting `ugettext_lazy()` objects to text strings before passing them to non-Django code:

```
requests.post('https://example.com/send', data={'body': str(body)})
```

Use `unicode` in place of `str` on Python 2, or `six.text_type` to support Python 2 and 3.

If you try to use a `ugettext_lazy()` result where a bytestring (a `bytes` object) is expected, things won't work as expected since a `ugettext_lazy()` object doesn't know how to convert itself to a bytestring. You can't use a unicode string inside a bytestring, either, so this is consistent with normal Python behavior. For example, putting a unicode proxy into a unicode string is fine:

```
"Hello %s" % ugettext_lazy("people")
```

But you can't insert a unicode object into a bytestring and nor can you insert a unicode proxy there:

```
b"Hello %s" % ugettext_lazy("people")
```

If you ever see output that looks like `"hello <django.utils.functional...>"`, you have tried to insert the result of `ugettext_lazy()` into a bytestring. That's a bug in your code.

If you don't like the long `ugettext_lazy` name, you can just alias it as _ (underscore), like so:

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

class MyThing(models.Model):
    name = models.CharField(help_text=_('This is the help text'))
```

Using `ugettext_lazy()` and `ungettext_lazy()` to mark strings in models and utility functions is a common operation. When you're working with these objects elsewhere in your code, you should ensure that you don't accidentally convert them to strings, because they should be converted as late as possible (so that the correct locale is in effect). This necessitates the use of the helper function described next.

Lazy translations and plural When using lazy translation for a plural string (`[u]n[p]gettext_lazy`), you generally don't know the `number` argument at the time of the string definition. Therefore, you are authorized to pass a key name instead of an integer as the `number` argument. Then `number` will be looked up in the dictionary under that key during string interpolation. Here's example:

```
from django import forms
from django.utils.translation import ungettext_lazy

class MyForm(forms.Form):
    error_message = ungettext_lazy("You only provided %(num)d argument",
        "You only provided %(num)d arguments", 'num')

    def clean(self):
        # ...
        if error:
            raise forms.ValidationError(self.error_message % {'num': number})
```

If the string contains exactly one unnamed placeholder, you can interpolate directly with the `number` argument:

```
class MyForm(forms.Form):
    error_message = ungettext_lazy(
        "You provided %d argument",
        "You provided %d arguments",
    )

    def clean(self):
        # ...
        if error:
            raise forms.ValidationError(self.error_message % number)
```

Joining strings: `string_concat()` Standard Python string joins (`''.join([...])`) will not work on lists containing lazy translation objects. Instead, you can use *`django.utils.translation.string_concat()`*, which creates a lazy object that concatenates its contents *and* converts them to strings only when the result is included in a string. For example:

```
from django.utils.translation import string_concat
from django.utils.translation import ugettext_lazy
...
name = ugettext_lazy('John Lennon')
instrument = ugettext_lazy('guitar')
result = string_concat(name, ': ', instrument)
```

In this case, the lazy translations in `result` will only be converted to strings when `result` itself is used in a string (usually at template rendering time).

Other uses of lazy in delayed translations For any other case where you would like to delay the translation, but have to pass the translatable string as argument to another function, you can wrap this function inside a lazy call yourself. For example:

```
from django.utils import six  # Python 3 compatibility
from django.utils.functional import lazy
from django.utils.safestring import mark_safe
from django.utils.translation import ugettext_lazy as _

mark_safe_lazy = lazy(mark_safe, six.text_type)
```

And then later:

```
lazy_string = mark_safe_lazy(_("<p>My <strong>string!</strong></p>"))
```

Localized names of languages

get_language_info()

The `get_language_info()` function provides detailed information about languages:

```
>>> from django.utils.translation import activate, get_language_info
>>> activate('fr')
>>> li = get_language_info('de')
>>> print(li['name'], li['name_local'], li['name_translated'], li['bidi'])
German Deutsch Allemand False
```

The `name`, `name_local`, and `name_translated` attributes of the dictionary contain the name of the language in English, in the language itself, and in your current active language respectively. The `bidi` attribute is True only for bi-directional languages.

The source of the language information is the `django.conf.locale` module. Similar access to this information is available for template code. See below.

The `'name_translated'` attribute was added.

Internationalization: in template code

Translations in Django templates uses two template tags and a slightly different syntax than in Python code. To give your template access to these tags, put `{% load i18n %}` toward the top of your template. As with all template tags, this tag needs to be loaded in all templates which use translations, even those templates that extend from other templates which have already loaded the `i18n` tag.

trans template tag

The `{% trans %}` template tag translates either a constant string (enclosed in single or double quotes) or variable content:

```
<title>{% trans "This is the title." %}</title>
<title>{% trans myvar %}</title>
```

If the `noop` option is present, variable lookup still takes place but the translation is skipped. This is useful when "stubbing out" content that will require translation in the future:

```
<title>{% trans "myvar" noop %}</title>
```

Internally, inline translations use an *ugettext()* call.

In case a template var (`myvar` above) is passed to the tag, the tag will first resolve such variable to a string at run-time and then look up that string in the message catalogs.

It's not possible to mix a template variable inside a string within `{% trans %}`. If your translations require strings with variables (placeholders), use *{% blocktrans %}* instead.

If you'd like to retrieve a translated string without displaying it, you can use the following syntax:

```
{% trans "This is the title" as the_title %}

<title>{{ the_title }}</title>
<meta name="description" content="{{ the_title }}">
```

In practice you'll use this to get a string you can use in multiple places in a template or so you can use the output as an argument for other template tags or filters:

```
{% trans "starting point" as start %}
{% trans "end point" as end %}
{% trans "La Grande Boucle" as race %}

<h1>
  <a href="/" title="{% blocktrans %}Back to '{{ race }}' homepage{% endblocktrans %}">{{ race }}</a
</h1>
<p>
{% for stage in tour_stages %}
    {% cycle start end %}: {{ stage }}{% if forloop.counter|divisibleby:2 %}<br />{% else %}, {% end
{% endfor %}
</p>
```

`{% trans %}` also supports *contextual markers* using the `context` keyword:

```
{% trans "May" context "month name" %}
```

`blocktrans` template tag

Contrarily to the *`trans`* tag, the `blocktrans` tag allows you to mark complex sentences consisting of literals and variable content for translation by making use of placeholders:

```
{% blocktrans %}This string will have {{ value }} inside.{% endblocktrans %}
```

To translate a template expression – say, accessing object attributes or using template filters – you need to bind the expression to a local variable for use within the translation block. Examples:

```
{% blocktrans with amount=article.price %}
That will cost $ {{ amount }}.
{% endblocktrans %}

{% blocktrans with myvar=value|filter %}
This will have {{ myvar }} inside.
{% endblocktrans %}
```

You can use multiple expressions inside a single `blocktrans` tag:

```
{% blocktrans with book_t=book|title author_t=author|title %}
This is {{ book_t }} by {{ author_t }}
{% endblocktrans %}
```

: The previous more verbose format is still supported: `{% blocktrans with book|title as book_t and author|title as author_t %}`

Other block tags (for example `{% for %}` or `{% if %}`) are not allowed inside a `blocktrans` tag.

If resolving one of the block arguments fails, blocktrans will fall back to the default language by deactivating the currently active language temporarily with the *`deactivate_all()`* function.

This tag also provides for pluralization. To use it:

- Designate and bind a counter value with the name `count`. This value will be the one used to select the right plural form.
- Specify both the singular and plural forms separating them with the `{% plural %}` tag within the `{% blocktrans %}` and `{% endblocktrans %}` tags.

An example:

```
{% blocktrans count counter=list|length %}
There is only one {{ name }} object.
{% plural %}
There are {{ counter }} {{ name }} objects.
{% endblocktrans %}
```

A more complex example:

```
{% blocktrans with amount=article.price count years=i.length %}
That will cost $ {{ amount }} per year.
{% plural %}
That will cost $ {{ amount }} per {{ years }} years.
{% endblocktrans %}
```

When you use both the pluralization feature and bind values to local variables in addition to the counter value, keep in mind that the `blocktrans` construct is internally converted to an `ungettext` call. This means the same *notes regarding ungettext variables* apply.

Reverse URL lookups cannot be carried out within the `blocktrans` and should be retrieved (and stored) beforehand:

```
{% url 'path.to.view' arg arg2 as the_url %}
{% blocktrans %}
This is a URL: {{ the_url }}
{% endblocktrans %}
```

If you'd like to retrieve a translated string without displaying it, you can use the following syntax:

```
{% blocktrans asvar the_title %}The title is {{ title }}.{% endblocktrans %}
<title>{{ the_title }}</title>
<meta name="description" content="{{ the_title }}">
```

In practice you'll use this to get a string you can use in multiple places in a template or so you can use the output as an argument for other template tags or filters.

The `asvar` syntax was added.

`{% blocktrans %}` also supports *contextual markers* using the `context` keyword:

```
{% blocktrans with name=user.username context "greeting" %}Hi {{ name }}{% endblocktrans %}
```

Another feature `{% blocktrans %}` supports is the `trimmed` option. This option will remove newline characters from the beginning and the end of the content of the `{% blocktrans %}` tag, replace any whitespace at the beginning and end of a line and merge all lines into one using a space character to separate them. This is quite useful for indenting the content of a `{% blocktrans %}` tag without having the indentation characters end up in the corresponding entry in the PO file, which makes the translation process easier.

For instance, the following `{% blocktrans %}` tag:

```
{% blocktrans trimmed %}
  First sentence.
  Second paragraph.
{% endblocktrans %}
```

will result in the entry `"First sentence.  Second paragraph."` in the PO file, compared to `"\n First sentence.\n Second sentence.\n"`, if the `trimmed` option had not been specified.

String literals passed to tags and filters

You can translate string literals passed as arguments to tags and filters by using the familiar `_()` syntax:

```
{% some_tag _("Page not found") value|yesno:_("yes,no") %}
```

In this case, both the tag and the filter will see the translated string, so they don't need to be aware of translations.

: In this example, the translation infrastructure will be passed the string `"yes,no"`, not the individual strings `"yes"` and `"no"`. The translated string will need to contain the comma so that the filter parsing code knows how to split up the arguments. For example, a German translator might translate the string `"yes,no"` as `"ja,nein"` (keeping the comma intact).

Comments for translators in templates

Just like with *Python code*, these notes for translators can be specified using comments, either with the *`comment`* tag:

```
{% comment %}Translators: View verb{% endcomment %}
{% trans "View" %}

{% comment %}Translators: Short intro blurb{% endcomment %}
<p>{% blocktrans %}A multiline translatable
literal.{% endblocktrans %}</p>
```

or with the `{# ... #}` *one-line comment constructs*:

```
{# Translators: Label of a button that triggers search #}
<button type="submit">{% trans "Go" %}</button>

{# Translators: This is a text of the base template #}
{% blocktrans %}Ambiguous translatable block of text{% endblocktrans %}
```

: Just for completeness, these are the corresponding fragments of the resulting `.po` file:

```
#. Translators: View verb
# path/to/template/file.html:10
msgid "View"
msgstr ""

#. Translators: Short intro blurb
# path/to/template/file.html:13
msgid ""
"A multiline translatable"
"literal."
msgstr ""

# ...

#. Translators: Label of a button that triggers search
# path/to/template/file.html:100
msgid "Go"
msgstr ""

#. Translators: This is a text of the base template
```

```
# path/to/template/file.html:103
msgid "Ambiguous translatable block of text"
msgstr ""
```

Switching language in templates

If you want to select a language within a template, you can use the `language` template tag:

```
{% load i18n %}

{% get_current_language as LANGUAGE_CODE %}
<!-- Current language: {{ LANGUAGE_CODE }} -->
<p>{% trans "Welcome to our page" %}</p>

{% language 'en' %}
    {% get_current_language as LANGUAGE_CODE %}
    <!-- Current language: {{ LANGUAGE_CODE }} -->
    <p>{% trans "Welcome to our page" %}</p>
{% endlanguage %}
```

While the first occurrence of "Welcome to our page" uses the current language, the second will always be in English.

Other tags

These tags also require a `{% load i18n %}`.

`get_available_languages` `{% get_available_languages as LANGUAGES %}` returns a list of tuples in which the first element is the *language code* and the second is the language name (translated into the currently active locale).

`get_current_language` `{% get_current_language as LANGUAGE_CODE %}` returns the current user's preferred language as a string. Example: `en-us`. See *How Django discovers language preference*.

`get_current_language_bidi` `{% get_current_language_bidi as LANGUAGE_BIDI %}` returns the current locale's direction. If `True`, it's a right-to-left language, e.g. Hebrew, Arabic. If `False` it's a left-to-right language, e.g. English, French, German, etc.

If you enable the `django.template.context_processors.i18n` context processor then each `RequestContext` will have access to `LANGUAGES`, `LANGUAGE_CODE`, and `LANGUAGE_BIDI` as defined above.

`get_language_info` You can also retrieve information about any of the available languages using provided template tags and filters. To get information about a single language, use the `{% get_language_info %}` tag:

```
{% get_language_info for LANGUAGE_CODE as lang %}
{% get_language_info for "pl" as lang %}
```

You can then access the information:

```
Language code: {{ lang.code }}<br />
Name of language: {{ lang.name_local }}<br />
Name in English: {{ lang.name }}<br />
Bi-directional: {{ lang.bidi }}
Name in the active language: {{ lang.name_translated }}
```

The `name_translated` attribute was added.

`get_language_info_list` You can also use the `{% get_language_info_list %}` template tag to retrieve information for a list of languages (e.g. active languages as specified in *`LANGUAGES`*). See *the section about the set_language redirect view* for an example of how to display a language selector using `{% get_language_info_list %}`.

In addition to *`LANGUAGES`* style list of tuples, `{% get_language_info_list %}` supports simple lists of language codes. If you do this in your view:

```
context = {'available_languages': ['en', 'es', 'fr']}
return render(request, 'mytemplate.html', context)
```

you can iterate over those languages in the template:

```
{% get_language_info_list for available_languages as langs %}
{% for lang in langs %} ... {% endfor %}
```

Template filters There are also simple filters available for convenience:

- `{{ LANGUAGE_CODE|language_name }}` ("German")
- `{{ LANGUAGE_CODE|language_name_local }}` ("Deutsch")
- `{{ LANGUAGE_CODE|language_bidi }}` (False)
- `{{ LANGUAGE_CODE|language_name_translated }}` ("německy", when active language is Czech)

The `language_name_translated` filter was added.

Internationalization: in JavaScript code

Adding translations to JavaScript poses some problems:

- JavaScript code doesn't have access to a `gettext` implementation.
- JavaScript code doesn't have access to `.po` or `.mo` files; they need to be delivered by the server.
- The translation catalogs for JavaScript should be kept as small as possible.

Django provides an integrated solution for these problems: It passes the translations into JavaScript, so you can call `gettext`, etc., from within JavaScript.

The main solution to these problems is the following `JavaScriptCatalog` view, which generates a JavaScript code library with functions that mimic the `gettext` interface, plus an array of translation strings.

The `JavaScriptCatalog` view

class `JavaScriptCatalog`

A view that produces a JavaScript code library with functions that mimic the `gettext` interface, plus an array of translation strings.

Attributes

domain
Translation domain containing strings to add in the view output. Defaults to `'djangojs'`.

packages
A list of *application names* among installed applications. Those apps should contain a `locale` directory. All those catalogs plus all catalogs found in *LOCALE_PATHS* (which are always included) are merged into one catalog. Defaults to `None`, which means that all available translations from all *INSTALLED_APPS* are provided in the JavaScript output.

Example with default values:

```
from django.views.i18n import JavaScriptCatalog

urlpatterns = [
    url(r'^jsi18n/$', JavaScriptCatalog.as_view(), name='javascript-catalog'),
]
```

Example with custom packages:

```
urlpatterns = [
    url(r'^jsi18n/myapp/$',
        JavaScriptCatalog.as_view(packages=['your.app.label']),
        name='javascript-catalog'),
]
```

The precedence of translations is such that the packages appearing later in the `packages` argument have higher precedence than the ones appearing at the beginning. This is important in the case of clashing translations for the same literal.

If you use more than one `JavaScriptCatalog` view on a site and some of them define the same strings, the strings in the catalog that was loaded last take precedence.

The `javascript_catalog` view

javascript_catalog(*request*, *domain='djangojs'*, *packages=None*)

1.10 : `javascript_catalog()` is deprecated in favor of *JavaScriptCatalog* and will be removed in Django 2.0.

The main solution to these problems is the *django.views.i18n.javascript_catalog()* view, which sends out a JavaScript code library with functions that mimic the `gettext` interface, plus an array of translation strings. Those translation strings are taken from applications or Django core, according to what you specify in either the `info_dict` or the URL. Paths listed in *LOCALE_PATHS* are also included.

You hook it up like this:

```
from django.views.i18n import javascript_catalog

js_info_dict = {
    'packages': ('your.app.package',),
}

urlpatterns = [
    url(r'^jsi18n/$', javascript_catalog, js_info_dict, name='javascript-catalog'),
]
```

Each string in `packages` should be in Python dotted-package syntax (the same format as the strings in *INSTALLED_APPS*) and should refer to a package that contains a `locale` directory. If you specify multiple packages, all those catalogs are merged into one catalog. This is useful if you have JavaScript that uses strings from different applications.

The precedence of translations is such that the packages appearing later in the `packages` argument have higher precedence than the ones appearing at the beginning, this is important in the case of clashing translations for the same literal.

By default, the view uses the `djangojs` gettext domain. This can be changed by altering the `domain` argument.

You can make the view dynamic by putting the packages into the URL pattern:

```
urlpatterns = [
    url(r'^jsi18n/(?P<packages>\S+?)/$', javascript_catalog, name='javascript-catalog'),
]
```

With this, you specify the packages as a list of package names delimited by '+' signs in the URL. This is especially useful if your pages use code from different apps and this changes often and you don't want to pull in one big catalog file. As a security measure, these values can only be either `django.conf` or any package from the *INSTALLED_APPS* setting.

You can also split the catalogs in multiple URLs and load them as you need in your sites:

```
js_info_dict_app = {
    'packages': ('your.app.package',),
}

js_info_dict_other_app = {
    'packages': ('your.other.app.package',),
}

urlpatterns = [
    url(r'^jsi18n/app/$', javascript_catalog, js_info_dict_app),
    url(r'^jsi18n/other_app/$', javascript_catalog, js_info_dict_other_app),
]
```

If you use more than one `javascript_catalog` on a site and some of them define the same strings, the strings in the catalog that was loaded last take precedence.

Before Django 1.9, the catalogs completely overwrote each other and you could only use one at a time.

The JavaScript translations found in the paths listed in the *LOCALE_PATHS* setting are also always included. To keep consistency with the translations lookup order algorithm used for Python and templates, the directories listed in *LOCALE_PATHS* have the highest precedence with the ones appearing first having higher precedence than the ones appearing later.

Using the JavaScript translation catalog

To use the catalog, just pull in the dynamically generated script like this:

```
<script type="text/javascript" src="{% url 'javascript-catalog' %}"></script>
```

This uses reverse URL lookup to find the URL of the JavaScript catalog view. When the catalog is loaded, your JavaScript code can use the following methods:

- `gettext`
- `ngettext`
- `interpolate`

- get_format
- gettext_noop
- pgettext
- npgettext
- pluralidx

gettext The gettext function behaves similarly to the standard gettext interface within your Python code:

```
document.write(gettext('this is to be translated'));
```

ngettext The ngettext function provides an interface to pluralize words and phrases:

```
var object_count = 1 // or 0, or 2, or 3, ...
s = ngettext('literal for the singular case',
        'literal for the plural case', object_count);
```

interpolate The interpolate function supports dynamically populating a format string. The interpolation syntax is borrowed from Python, so the interpolate function supports both positional and named interpolation:

- Positional interpolation: obj contains a JavaScript Array object whose elements values are then sequentially interpolated in their corresponding fmt placeholders in the same order they appear. For example:

```
fmts = ngettext('There is %s object. Remaining: %s',
        'There are %s objects. Remaining: %s', 11);
s = interpolate(fmts, [11, 20]);
// s is 'There are 11 objects. Remaining: 20'
```

- Named interpolation: This mode is selected by passing the optional boolean named parameter as true. obj contains a JavaScript object or associative array. For example:

```
d = {
    count: 10,
    total: 50
};

fmts = ngettext('Total: %(total)s, there is %(count)s object',
'there are %(count)s of a total of %(total)s objects', d.count);
s = interpolate(fmts, d, true);
```

You shouldn't go over the top with string interpolation, though: this is still JavaScript, so the code has to make repeated regular-expression substitutions. This isn't as fast as string interpolation in Python, so keep it to those cases where you really need it (for example, in conjunction with ngettext to produce proper pluralizations).

get_format The get_format function has access to the configured i18n formatting settings and can retrieve the format string for a given setting name:

```
document.write(get_format('DATE_FORMAT'));
// 'N j, Y'
```

It has access to the following settings:

- *DATE_FORMAT*
- *DATE_INPUT_FORMATS*

- *DATETIME_FORMAT*
- *DATETIME_INPUT_FORMATS*
- *DECIMAL_SEPARATOR*
- *FIRST_DAY_OF_WEEK*
- *MONTH_DAY_FORMAT*
- *NUMBER_GROUPING*
- *SHORT_DATE_FORMAT*
- *SHORT_DATETIME_FORMAT*
- *THOUSAND_SEPARATOR*
- *TIME_FORMAT*
- *TIME_INPUT_FORMATS*
- *YEAR_MONTH_FORMAT*

This is useful for maintaining formatting consistency with the Python-rendered values.

gettext_noop This emulates the `gettext` function but does nothing, returning whatever is passed to it:

```
document.write(gettext_noop('this will not be translated'));
```

This is useful for stubbing out portions of the code that will need translation in the future.

pgettext The `pgettext` function behaves like the Python variant (*pgettext()*), providing a contextually translated word:

```
document.write(pgettext('month name', 'May'));
```

npgettext The `npgettext` function also behaves like the Python variant (*npgettext()*), providing a **pluralized** contextually translated word:

```
document.write(npgettext('group', 'party', 1));
// party
document.write(npgettext('group', 'party', 2));
// parties
```

pluralidx The `pluralidx` function works in a similar way to the *pluralize* template filter, determining if a given `count` should use a plural form of a word or not:

```
document.write(pluralidx(0));
// true
document.write(pluralidx(1));
// false
document.write(pluralidx(2));
// true
```

In the simplest case, if no custom pluralization is needed, this returns `false` for the integer `1` and `true` for all other numbers.

However, pluralization is not this simple in all languages. If the language does not support pluralization, an empty value is provided.

Additionally, if there are complex rules around pluralization, the catalog view will render a conditional expression. This will evaluate to either a `true` (should pluralize) or `false` (should **not** pluralize) value.

The `JSONCatalog` view

class JSONCatalog

In order to use another client-side library to handle translations, you may want to take advantage of the `JSONCatalog` view. It's similar to *`JavaScriptCatalog`* but returns a JSON response.

See the documentation for *`JavaScriptCatalog`* to learn about possible values and use of the `domain` and `packages` attributes.

The response format is as follows:

```
{
    "catalog": {
        # Translations catalog
    },
    "formats": {
        # Language formats for date, time, etc.
    },
    "plural": "..."  # Expression for plural forms, or null.
}
```

The `json_catalog` view

json_catalog(*request*, *domain='djangojs'*, *packages=None*)

1.10 : `json_catalog()` is deprecated in favor of *`JSONCatalog`* and will be removed in Django 2.0.

In order to use another client-side library to handle translations, you may want to take advantage of the `json_catalog()` view. It's similar to *`javascript_catalog()`* but returns a JSON response.

The JSON object contains i18n formatting settings (those available for *get_format*), a plural rule (as a `plural` part of a GNU gettext `Plural-Forms` expression), and translation strings. The translation strings are taken from applications or Django's own translations, according to what is specified either via `urlpatterns` arguments or as request parameters. Paths listed in *`LOCALE_PATHS`* are also included.

The view is hooked up to your application and configured in the same fashion as *`javascript_catalog()`* (namely, the `domain` and `packages` arguments behave identically):

```
from django.views.i18n import json_catalog

js_info_dict = {
    'packages': ('your.app.package',),
}

urlpatterns = [
    url(r'^jsoni18n/$', json_catalog, js_info_dict),
]
```

The response format is as follows:

```
{
    "catalog": {
        # Translations catalog
    },
    "formats": {
```

```
        # Language formats for date, time, etc.
    },
    "plural": "..."  # Expression for plural forms, or null.
}
```

Note on performance

The various JavaScript/JSON i18n views generate the catalog from `.mo` files on every request. Since its output is constant, at least for a given version of a site, it's a good candidate for caching.

Server-side caching will reduce CPU load. It's easily implemented with the *cache_page()* decorator. To trigger cache invalidation when your translations change, provide a version-dependent key prefix, as shown in the example below, or map the view at a version-dependent URL:

```
from django.views.decorators.cache import cache_page
from django.views.i18n import JavaScriptCatalog

# The value returned by get_version() must change when translations change.
urlpatterns = [
    url(r'^jsi18n/$',
        cache_page(86400, key_prefix='js18n-%s' % get_version())(JavaScriptCatalog.as_view()),
        name='javascript-catalog'),
]
```

Client-side caching will save bandwidth and make your site load faster. If you're using ETags (*USE_ETAGS = True*), you're already covered. Otherwise, you can apply *conditional decorators*. In the following example, the cache is invalidated whenever you restart your application server:

```
from django.utils import timezone
from django.views.decorators.http import last_modified
from django.views.i18n import JavaScriptCatalog

last_modified_date = timezone.now()

urlpatterns = [
    url(r'^jsi18n/$',
        last_modified(lambda req, **kw: last_modified_date)(JavaScriptCatalog.as_view()),
        name='javascript-catalog'),
]
```

You can even pre-generate the JavaScript catalog as part of your deployment procedure and serve it as a static file. This radical technique is implemented in django-statici18n.

Internationalization: in URL patterns

Django provides two mechanisms to internationalize URL patterns:

- Adding the language prefix to the root of the URL patterns to make it possible for *LocaleMiddleware* to detect the language to activate from the requested URL.
- Making URL patterns themselves translatable via the *django.utils.translation.ugettext_lazy()* function.

: Using either one of these features requires that an active language be set for each request; in other words, you need to have *django.middleware.locale.LocaleMiddleware* in your *MIDDLEWARE* setting.

Language prefix in URL patterns

i18n_patterns(**urls*, *prefix_default_language=True*)

This function can be used in a root URLconf and Django will automatically prepend the current active language code to all URL patterns defined within *i18n_patterns()*.

Setting `prefix_default_language` to `False` removes the prefix from the default language (*LANGUAGE_CODE*). This can be useful when adding translations to existing site so that the current URLs won't change.

Example URL patterns:

```
from django.conf.urls import include, url
from django.conf.urls.i18n import i18n_patterns

from about import views as about_views
from news import views as news_views
from sitemap.views import sitemap

urlpatterns = [
    url(r'^sitemap\.xml$', sitemap, name='sitemap-xml'),
]

news_patterns = ([
    url(r'^$', news_views.index, name='index'),
    url(r'^category/(?P<slug>[\w-]+)/$', news_views.category, name='category'),
    url(r'^(?P<slug>[\w-]+)/$', news_views.details, name='detail'),
], 'news')

urlpatterns += i18n_patterns(
    url(r'^about/$', about_views.main, name='about'),
    url(r'^news/', include(news_patterns, namespace='news')),
)
```

After defining these URL patterns, Django will automatically add the language prefix to the URL patterns that were added by the `i18n_patterns` function. Example:

```
>>> from django.urls import reverse
>>> from django.utils.translation import activate

>>> activate('en')
>>> reverse('sitemap-xml')
'/sitemap.xml'
>>> reverse('news:index')
'/en/news/'

>>> activate('nl')
>>> reverse('news:detail', kwargs={'slug': 'news-slug'})
'/nl/news/news-slug/'
```

With `prefix_default_language=False` and `LANGUAGE_CODE='en'`, the URLs will be:

```
>>> activate('en')
>>> reverse('news:index')
'/news/'

>>> activate('nl')
>>> reverse('news:index')
'/nl/news/'
```

The `prefix_default_language` parameter was added.

: *`i18n_patterns()`* is only allowed in a root URLconf. Using it within an included URLconf will throw an *`ImproperlyConfigured`* exception.

In older version, using `i18n_patterns` in a root URLconf different from *`ROOT_URLCONF`* by setting *`request.urlconf`* wasn't supported.

: Ensure that you don't have non-prefixed URL patterns that might collide with an automatically-added language prefix.

Translating URL patterns

URL patterns can also be marked translatable using the *`ugettext_lazy()`* function. Example:

```
from django.conf.urls import include, url
from django.conf.urls.i18n import i18n_patterns
from django.utils.translation import ugettext_lazy as _

from about import views as about_views
from news import views as news_views
from sitemaps.views import sitemap

urlpatterns = [
    url(r'^sitemap\.xml$', sitemap, name='sitemap-xml'),
]

news_patterns = ([
    url(r'^$', news_views.index, name='index'),
    url(_(r'^category/(?P<slug>[\w-]+)/$'), news_views.category, name='category'),
    url(r'^(?P<slug>[\w-]+)/$', news_views.details, name='detail'),
], 'news')

urlpatterns += i18n_patterns(
    url(_(r'^about/$'), about_views.main, name='about'),
    url(_(r'^news/'), include(news_patterns, namespace='news')),
)
```

After you've created the translations, the *`reverse()`* function will return the URL in the active language. Example:

```
>>> from django.urls import reverse
>>> from django.utils.translation import activate

>>> activate('en')
>>> reverse('news:category', kwargs={'slug': 'recent'})
'/en/news/category/recent/'

>>> activate('nl')
>>> reverse('news:category', kwargs={'slug': 'recent'})
'/nl/nieuws/categorie/recent/'
```

: In most cases, it's best to use translated URLs only within a language-code-prefixed block of patterns (using *`i18n_patterns()`*), to avoid the possibility that a carelessly translated URL causes a collision with a non-translated URL pattern.

Reversing in templates

If localized URLs get reversed in templates they always use the current language. To link to a URL in another language use the *language* template tag. It enables the given language in the enclosed template section:

```
{% load i18n %}

{% get_available_languages as languages %}

{% trans "View this category in:" %}
{% for lang_code, lang_name in languages %}
    {% language lang_code %}
    <a href="{% url 'category' slug=category.slug %}">{{ lang_name }}</a>
    {% endlanguage %}
{% endfor %}
```

The *language* tag expects the language code as the only argument.

Localization: how to create language files

Once the string literals of an application have been tagged for later translation, the translation themselves need to be written (or obtained). Here's how that works.

Message files

The first step is to create a *message file* for a new language. A message file is a plain-text file, representing a single language, that contains all available translation strings and how they should be represented in the given language. Message files have a `.po` file extension.

Django comes with a tool, *django-admin makemessages*, that automates the creation and upkeep of these files.

Gettext utilities

The `makemessages` command (and `compilemessages` discussed later) use commands from the GNU gettext toolset: `xgettext`, `msgfmt`, `msgmerge` and `msguniq`.

The minimum version of the `gettext` utilities supported is 0.15.

To create or update a message file, run this command:

```
django-admin makemessages -l de
```

...where `de` is the *locale name* for the message file you want to create. For example, `pt_BR` for Brazilian Portuguese, `de_AT` for Austrian German or `id` for Indonesian.

The script should be run from one of two places:

- The root directory of your Django project (the one that contains `manage.py`).
- The root directory of one of your Django apps.

The script runs over your project source tree or your application source tree and pulls out all strings marked for translation (see *How Django discovers translations* and be sure *LOCALE_PATHS* is configured correctly). It creates (or updates) a message file in the directory `locale/LANG/LC_MESSAGES`. In the `de` example, the file will be `locale/de/LC_MESSAGES/django.po`.

When you run `makemessages` from the root directory of your project, the extracted strings will be automatically distributed to the proper message files. That is, a string extracted from a file of an app containing a `locale` directory will go in a message file under that directory. A string extracted from a file of an app without any `locale` directory will either go in a message file under the directory listed first in *`LOCALE_PATHS`* or will generate an error if *`LOCALE_PATHS`* is empty.

By default *`django-admin makemessages`* examines every file that has the `.html` or `.txt` file extension. In case you want to override that default, use the `--extension` or `-e` option to specify the file extensions to examine:

```
django-admin makemessages -l de -e txt
```

Separate multiple extensions with commas and/or use `-e` or `--extension` multiple times:

```
django-admin makemessages -l de -e html,txt -e xml
```

: When *creating message files from JavaScript source code* you need to use the special 'djangojs' domain, **not** `-e js`.

Using Jinja2 templates?

`makemessages` doesn't understand the syntax of Jinja2 templates. To extract strings from a project containing Jinja2 templates, use Message Extracting from Babel instead.

Here's an example `babel.cfg` configuration file:

```
# Extraction from Python source files
[python: **.py]

# Extraction from Jinja2 templates
[jinja2: **.jinja]
extensions = jinja2.ext.with_
```

Make sure you list all extensions you're using! Otherwise Babel won't recognize the tags defined by these extensions and will ignore Jinja2 templates containing them entirely.

Babel provides similar features to *`makemessages`*, can replace it in general, and doesn't depend on `gettext`. For more information, read its documentation about working with message catalogs.

No gettext?

If you don't have the `gettext` utilities installed, *`makemessages`* will create empty files. If that's the case, either install the `gettext` utilities or just copy the English message file (`locale/en/LC_MESSAGES/django.po`) if available and use it as a starting point; it's just an empty translation file.

Working on Windows?

If you're using Windows and need to install the GNU gettext utilities so *`makemessages`* works, see *gettext on Windows* for more information.

The format of `.po` files is straightforward. Each `.po` file contains a small bit of metadata, such as the translation maintainer's contact information, but the bulk of the file is a list of **messages** – simple mappings between translation strings and the actual translated text for the particular language.

For example, if your Django app contained a translation string for the text `"Welcome to my site."`, like so:

```
_("Welcome to my site.")
```

...then *django-admin makemessages* will have created a `.po` file containing the following snippet – a message:

```
#: path/to/python/module.py:23
msgid "Welcome to my site."
msgstr ""
```

A quick explanation:

- `msgid` is the translation string, which appears in the source. Don't change it.
- `msgstr` is where you put the language-specific translation. It starts out empty, so it's your responsibility to change it. Make sure you keep the quotes around your translation.
- As a convenience, each message includes, in the form of a comment line prefixed with `#` and located above the `msgid` line, the filename and line number from which the translation string was gleaned.

Long messages are a special case. There, the first string directly after the `msgstr` (or `msgid`) is an empty string. Then the content itself will be written over the next few lines as one string per line. Those strings are directly concatenated. Don't forget trailing spaces within the strings; otherwise, they'll be tacked together without whitespace!

Mind your charset

Due to the way the `gettext` tools work internally and because we want to allow non-ASCII source strings in Django's core and your applications, you **must** use UTF-8 as the encoding for your PO files (the default when PO files are created). This means that everybody will be using the same encoding, which is important when Django processes the PO files.

To reexamine all source code and templates for new translation strings and update all message files for **all** languages, run this:

```
django-admin makemessages -a
```

Compiling message files

After you create your message file – and each time you make changes to it – you'll need to compile it into a more efficient form, for use by `gettext`. Do this with the *django-admin compilemessages* utility.

This tool runs over all available `.po` files and creates `.mo` files, which are binary files optimized for use by `gettext`. In the same directory from which you ran *django-admin makemessages*, run *django-admin compilemessages* like this:

```
django-admin compilemessages
```

That's it. Your translations are ready for use.

`compilemessages` now matches the operation of *makemessages*, scanning the project tree for `.po` files to compile.

Working on Windows?

If you're using Windows and need to install the GNU gettext utilities so *django-admin compilemessages* works see *gettext on Windows* for more information.

.po files: Encoding and BOM usage.

Django only supports `.po` files encoded in UTF-8 and without any BOM (Byte Order Mark) so if your text editor adds such marks to the beginning of files by default then you will need to reconfigure it.

Creating message files from JavaScript source code

You create and update the message files the same way as the other Django message files – with the *`django-admin makemessages`* tool. The only difference is you need to explicitly specify what in gettext parlance is known as a domain in this case the `djangojs` domain, by providing a `-d djangojs` parameter, like this:

```
django-admin makemessages -d djangojs -l de
```

This would create or update the message file for JavaScript for German. After updating message files, just run *`django-admin compilemessages`* the same way as you do with normal Django message files.

`gettext` on Windows

This is only needed for people who either want to extract message IDs or compile message files (`.po`). Translation work itself just involves editing existing files of this type, but if you want to create your own message files, or want to test or compile a changed message file, download a precompiled binary installer.

You may also use `gettext` binaries you have obtained elsewhere, so long as the `xgettext --version` command works properly. Do not attempt to use Django translation utilities with a `gettext` package if the command `xgettext --version` entered at a Windows command prompt causes a popup window saying "xgettext.exe has generated errors and will be closed by Windows".

Customizing the `makemessages` command

If you want to pass additional parameters to `xgettext`, you need to create a custom *`makemessages`* command and override its `xgettext_options` attribute:

```
from django.core.management.commands import makemessages

class Command(makemessages.Command):
    xgettext_options = makemessages.Command.xgettext_options + ['--keyword=mytrans']
```

If you need more flexibility, you could also add a new argument to your custom *`makemessages`* command:

```
from django.core.management.commands import makemessages

class Command(makemessages.Command):

    def add_arguments(self, parser):
        super(Command, self).add_arguments(parser)
        parser.add_argument(
            '--extra-keyword',
            dest='xgettext_keywords',
            action='append',
        )

    def handle(self, *args, **options):
        xgettext_keywords = options.pop('xgettext_keywords')
```

```
        if xgettext_keywords:
            self.xgettext_options = (
                makemessages.Command.xgettext_options[:] +
                ['--keyword=%s' % kwd for kwd in xgettext_keywords]
            )
        super(Command, self).handle(*args, **options)
```

Miscellaneous

The `set_language` redirect view

set_language(*request*)

As a convenience, Django comes with a view, *django.views.i18n.set_language()*, that sets a user's language preference and redirects to a given URL or, by default, back to the previous page.

Activate this view by adding the following line to your URLconf:

```
url(r'^i18n/', include('django.conf.urls.i18n')),
```

(Note that this example makes the view available at `/i18n/setlang/`.)

: Make sure that you don't include the above URL within *i18n_patterns()* - it needs to be language-independent itself to work correctly.

The view expects to be called via the `POST` method, with a `language` parameter set in request. If session support is enabled, the view saves the language choice in the user's session. Otherwise, it saves the language choice in a cookie that is by default named `django_language`. (The name can be changed through the *LANGUAGE_COOKIE_NAME* setting.)

After setting the language choice, Django looks for a `next` parameter in the `POST` or `GET` data. If that is found and Django considers it to be a safe URL (i.e. it doesn't point to a different host and uses a safe scheme), a redirect to that URL will be performed. Otherwise, Django may fall back to redirecting the user to the URL from the `Referer` header or, if it is not set, to `/`, depending on the nature of the request:

- For AJAX requests, the fallback will be performed only if the `next` parameter was set. Otherwise a 204 status code (No Content) will be returned.
- For non-AJAX requests, the fallback will always be performed.

Returning a 204 status code for AJAX requests when no redirect is specified is new.

Here's example HTML template code:

```
{% load i18n %}

<form action="{% url 'set_language' %}" method="post">{% csrf_token %}
    <input name="next" type="hidden" value="{{ redirect_to }}" />
    <select name="language">
        {% get_current_language as LANGUAGE_CODE %}
        {% get_available_languages as LANGUAGES %}
        {% get_language_info_list for LANGUAGES as languages %}
        {% for language in languages %}
            <option value="{{ language.code }}"{% if language.code == LANGUAGE_CODE %} selected="sele
                {{ language.name_local }} ({{ language.code }})
            </option>
        {% endfor %}
    </select>
```

```
    <input type="submit" value="Go" />
</form>
```

In this example, Django looks up the URL of the page to which the user will be redirected in the `redirect_to` context variable.

Explicitly setting the active language

You may want to set the active language for the current session explicitly. Perhaps a user's language preference is retrieved from another system, for example. You've already been introduced to *`django.utils.translation.activate()`*. That applies to the current thread only. To persist the language for the entire session, also modify *`LANGUAGE_SESSION_KEY`* in the session:

```
from django.utils import translation
user_language = 'fr'
translation.activate(user_language)
request.session[translation.LANGUAGE_SESSION_KEY] = user_language
```

You would typically want to use both: *`django.utils.translation.activate()`* will change the language for this thread, and modifying the session makes this preference persist in future requests.

If you are not using sessions, the language will persist in a cookie, whose name is configured in *`LANGUAGE_COOKIE_NAME`*. For example:

```
from django.utils import translation
from django import http
from django.conf import settings
user_language = 'fr'
translation.activate(user_language)
response = http.HttpResponse(...)
response.set_cookie(settings.LANGUAGE_COOKIE_NAME, user_language)
```

Using translations outside views and templates

While Django provides a rich set of i18n tools for use in views and templates, it does not restrict the usage to Django-specific code. The Django translation mechanisms can be used to translate arbitrary texts to any language that is supported by Django (as long as an appropriate translation catalog exists, of course). You can load a translation catalog, activate it and translate text to language of your choice, but remember to switch back to original language, as activating a translation catalog is done on per-thread basis and such change will affect code running in the same thread.

For example:

```
from django.utils import translation

def welcome_translated(language):
    cur_language = translation.get_language()
    try:
        translation.activate(language)
        text = translation.ugettext('welcome')
    finally:
        translation.activate(cur_language)
    return text
```

Calling this function with the value 'de' will give you `"Willkommen"`, regardless of *`LANGUAGE_CODE`* and language set by middleware.

Functions of particular interest are *django.utils.translation.get_language()* which returns the language used in the current thread, *django.utils.translation.activate()* which activates a translation catalog for the current thread, and *django.utils.translation.check_for_language()* which checks if the given language is supported by Django.

To help write more concise code, there is also a context manager *django.utils.translation.override()* that stores the current language on enter and restores it on exit. With it, the above example becomes:

```
from django.utils import translation

def welcome_translated(language):
    with translation.override(language):
        return translation.ugettext('welcome')
```

Language cookie

A number of settings can be used to adjust language cookie options:

- *LANGUAGE_COOKIE_NAME*
- *LANGUAGE_COOKIE_AGE*
- *LANGUAGE_COOKIE_DOMAIN*
- *LANGUAGE_COOKIE_PATH*

Implementation notes

Specialties of Django translation

Django's translation machinery uses the standard `gettext` module that comes with Python. If you know `gettext`, you might note these specialties in the way Django does translation:

- The string domain is `django` or `djangojs`. This string domain is used to differentiate between different programs that store their data in a common message-file library (usually `/usr/share/locale/`). The `django` domain is used for Python and template translation strings and is loaded into the global translation catalogs. The `djangojs` domain is only used for JavaScript translation catalogs to make sure that those are as small as possible.
- Django doesn't use `xgettext` alone. It uses Python wrappers around `xgettext` and `msgfmt`. This is mostly for convenience.

How Django discovers language preference

Once you've prepared your translations – or, if you just want to use the translations that come with Django – you'll just need to activate translation for your app.

Behind the scenes, Django has a very flexible model of deciding which language should be used – installation-wide, for a particular user, or both.

To set an installation-wide language preference, set *LANGUAGE_CODE*. Django uses this language as the default translation – the final attempt if no better matching translation is found through one of the methods employed by the locale middleware (see below).

If all you want is to run Django with your native language all you need to do is set *LANGUAGE_CODE* and make sure the corresponding *message files* and their compiled versions (`.mo`) exist.

If you want to let each individual user specify which language they prefer, then you also need to use the `LocaleMiddleware`. `LocaleMiddleware` enables language selection based on data from the request. It customizes content for each user.

To use `LocaleMiddleware`, add `'django.middleware.locale.LocaleMiddleware'` to your *MIDDLEWARE* setting. Because middleware order matters, follow these guidelines:

- Make sure it's one of the first middlewares installed.
- It should come after `SessionMiddleware`, because `LocaleMiddleware` makes use of session data. And it should come before `CommonMiddleware` because `CommonMiddleware` needs an activated language in order to resolve the requested URL.
- If you use `CacheMiddleware`, put `LocaleMiddleware` after it.

For example, your *MIDDLEWARE* might look like this:

```
MIDDLEWARE = [
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.locale.LocaleMiddleware',
    'django.middleware.common.CommonMiddleware',
]
```

(For more on middleware, see the middleware documentation.)

`LocaleMiddleware` tries to determine the user's language preference by following this algorithm:

- First, it looks for the language prefix in the requested URL. This is only performed when you are using the `i18n_patterns` function in your root URLconf. See *Internationalization: in URL patterns* for more information about the language prefix and how to internationalize URL patterns.
- Failing that, it looks for the *LANGUAGE_SESSION_KEY* key in the current user's session.
- Failing that, it looks for a cookie.

  The name of the cookie used is set by the *LANGUAGE_COOKIE_NAME* setting. (The default name is `django_language`.)
- Failing that, it looks at the `Accept-Language` HTTP header. This header is sent by your browser and tells the server which language(s) you prefer, in order by priority. Django tries each language in the header until it finds one with available translations.
- Failing that, it uses the global *LANGUAGE_CODE* setting.

Notes:

- In each of these places, the language preference is expected to be in the standard *language format*, as a string. For example, Brazilian Portuguese is `pt-br`.
- If a base language is available but the sublanguage specified is not, Django uses the base language. For example, if a user specifies `de-at` (Austrian German) but Django only has `de` available, Django uses `de`.
- Only languages listed in the *LANGUAGES* setting can be selected. If you want to restrict the language selection to a subset of provided languages (because your application doesn't provide all those languages), set *LANGUAGES* to a list of languages. For example:

```
LANGUAGES = [
  ('de', _('German')),
  ('en', _('English')),
]
```

  This example restricts languages that are available for automatic selection to German and English (and any sublanguage, like de-ch or en-us).

- If you define a custom *LANGUAGES* setting, as explained in the previous bullet, you can mark the language names as translation strings – but use *ugettext_lazy()* instead of *ugettext()* to avoid a circular import.

  Here's a sample settings file:

```
from django.utils.translation import ugettext_lazy as _

LANGUAGES = [
    ('de', _('German')),
    ('en', _('English')),
]
```

Once `LocaleMiddleware` determines the user's preference, it makes this preference available as `request.LANGUAGE_CODE` for each *HttpRequest*. Feel free to read this value in your view code. Here's a simple example:

```
from django.http import HttpResponse

def hello_world(request, count):
    if request.LANGUAGE_CODE == 'de-at':
        return HttpResponse("You prefer to read Austrian German.")
    else:
        return HttpResponse("You prefer to read another language.")
```

Note that, with static (middleware-less) translation, the language is in `settings.LANGUAGE_CODE`, while with dynamic (middleware) translation, it's in `request.LANGUAGE_CODE`.

How Django discovers translations

At runtime, Django builds an in-memory unified catalog of literals-translations. To achieve this it looks for translations by following this algorithm regarding the order in which it examines the different file paths to load the compiled *message files* (`.mo`) and the precedence of multiple translations for the same literal:

1. The directories listed in *LOCALE_PATHS* have the highest precedence, with the ones appearing first having higher precedence than the ones appearing later.
2. Then, it looks for and uses if it exists a `locale` directory in each of the installed apps listed in *INSTALLED_APPS*. The ones appearing first have higher precedence than the ones appearing later.
3. Finally, the Django-provided base translation in `django/conf/locale` is used as a fallback.

:

The translations for literals included in JavaScript assets are looked up following a similar but not identical algorithm. See the *javascript_catalog view documentation* for more details.

In all cases the name of the directory containing the translation is expected to be named using *locale name* notation. E.g. `de`, `pt_BR`, `es_AR`, etc.

This way, you can write applications that include their own translations, and you can override base translations in your project. Or, you can just build a big project out of several apps and put all translations into one big common message file specific to the project you are composing. The choice is yours.

All message file repositories are structured the same way. They are:

- All paths listed in *LOCALE_PATHS* in your settings file are searched for `<language>/LC_MESSAGES/django.(po|mo)`
- `$APPPATH/locale/<language>/LC_MESSAGES/django.(po|mo)`
- `$PYTHONPATH/django/conf/locale/<language>/LC_MESSAGES/django.(po|mo)`

To create message files, you use the *django-admin makemessages* tool. And you use *django-admin compilemessages* to produce the binary `.mo` files that are used by `gettext`.

You can also run *django-admin compilemessages --settings=path.to.settings* to make the compiler process all the directories in your *LOCALE_PATHS* setting.

Using a non-English base language

Django makes the general assumption that the original strings in a translatable project are written in English. You can choose another language, but you must be aware of certain limitations:

- `gettext` only provides two plural forms for the original messages, so you will also need to provide a translation for the base language to include all plural forms if the plural rules for the base language are different from English.
- When an English variant is activated and English strings are missing, the fallback language will not be the *LANGUAGE_CODE* of the project, but the original strings. For example, an English user visiting a site with Spanish as the default language and original strings written in Russian will fallback to Russian, not to Spanish.

3.15.2 Format localization

Overview

Django's formatting system is capable of displaying dates, times and numbers in templates using the format specified for the current *locale*. It also handles localized input in forms.

When it's enabled, two users accessing the same content may see dates, times and numbers formatted in different ways, depending on the formats for their current locale.

The formatting system is disabled by default. To enable it, it's necessary to set *USE_L10N = True* in your settings file.

: The default `settings.py` file created by *django-admin startproject* includes *USE_L10N = True* for convenience. Note, however, that to enable number formatting with thousand separators it is necessary to set *USE_THOUSAND_SEPARATOR = True* in your settings file. Alternatively, you could use *intcomma* to format numbers in your template.

: There is also an independent but related *USE_I18N* setting that controls if Django should activate translation. See Translation for more details.

Locale aware input in forms

When formatting is enabled, Django can use localized formats when parsing dates, times and numbers in forms. That means it tries different formats for different locales when guessing the format used by the user when inputting data on forms.

: Django uses different formats for displaying data to those it uses for parsing data. Most notably, the formats for parsing dates can't use the `%a` (abbreviated weekday name), `%A` (full weekday name), `%b` (abbreviated month name), `%B` (full month name), or `%p` (AM/PM).

To enable a form field to localize input and output data simply use its `localize` argument:

```
class CashRegisterForm(forms.Form):
   product = forms.CharField()
   revenue = forms.DecimalField(max_digits=4, decimal_places=2, localize=True)
```

Controlling localization in templates

When you have enabled formatting with *USE_L10N*, Django will try to use a locale specific format whenever it outputs a value in a template.

However, it may not always be appropriate to use localized values – for example, if you're outputting JavaScript or XML that is designed to be machine-readable, you will always want unlocalized values. You may also want to use localization in selected templates, rather than using localization everywhere.

To allow for fine control over the use of localization, Django provides the `l10n` template library that contains the following tags and filters.

Template tags

localize Enables or disables localization of template variables in the contained block.

This tag allows a more fine grained control of localization than *USE_L10N*.

To activate or deactivate localization for a template block, use:

```
{% load l10n %}

{% localize on %}
    {{ value }}
{% endlocalize %}

{% localize off %}
    {{ value }}
{% endlocalize %}
```

: The value of *USE_L10N* isn't respected inside of a `{% localize %}` block.

See *localize* and *unlocalize* for template filters that will do the same job on a per-variable basis.

Template filters

localize Forces localization of a single value.

For example:

```
{% load l10n %}

{{ value|localize }}
```

To disable localization on a single value, use *unlocalize*. To control localization over a large section of a template, use the *localize* template tag.

unlocalize Forces a single value to be printed without localization.

For example:

```
{% load l10n %}

{{ value|unlocalize }}
```

To force localization of a single value, use *localize*. To control localization over a large section of a template, use the *localize* template tag.

Creating custom format files

Django provides format definitions for many locales, but sometimes you might want to create your own, because a format files doesn't exist for your locale, or because you want to overwrite some of the values.

To use custom formats, specify the path where you'll place format files first. To do that, just set your *FORMAT_MODULE_PATH* setting to the package where format files will exist, for instance:

```
FORMAT_MODULE_PATH = [
    'mysite.formats',
    'some_app.formats',
]
```

Files are not placed directly in this directory, but in a directory named as the locale, and must be named `formats.py`. Be careful not to put sensitive information in these files as values inside can be exposed if you pass the string to `django.utils.formats.get_format()` (used by the *date* template filter).

To customize the English formats, a structure like this would be needed:

```
mysite/
    formats/
        __init__.py
        en/
            __init__.py
            formats.py
```

where `formats.py` contains custom format definitions. For example:

```
from __future__ import unicode_literals

THOUSAND_SEPARATOR = '\xa0'
```

to use a non-breaking space (Unicode `00A0`) as a thousand separator, instead of the default for English, a comma.

Limitations of the provided locale formats

Some locales use context-sensitive formats for numbers, which Django's localization system cannot handle automatically.

Switzerland (German)

The Swiss number formatting depends on the type of number that is being formatted. For monetary values, a comma is used as the thousand separator and a decimal point for the decimal separator. For all other numbers, a comma is used as decimal separator and a space as thousand separator. The locale format provided by Django uses the generic separators, a comma for decimal and a space for thousand separators.

3.15.3 Time zones

Overview

When support for time zones is enabled, Django stores datetime information in UTC in the database, uses time-zone-aware datetime objects internally, and translates them to the end user's time zone in templates and forms.

This is handy if your users live in more than one time zone and you want to display datetime information according to each user's wall clock.

Even if your website is available in only one time zone, it's still good practice to store data in UTC in your database. The main reason is Daylight Saving Time (DST). Many countries have a system of DST, where clocks are moved forward in spring and backward in autumn. If you're working in local time, you're likely to encounter errors twice a year, when the transitions happen. (The pytz documentation discusses these issues in greater detail.) This probably doesn't matter for your blog, but it's a problem if you over-bill or under-bill your customers by one hour, twice a year, every year. The solution to this problem is to use UTC in the code and use local time only when interacting with end users.

Time zone support is disabled by default. To enable it, set *USE_TZ = True* in your settings file. Installing pytz is highly recommended, but may not be mandatory depending on your particular database backend, operating system and time zone. If you encounter an exception querying dates or times, please try installing it before filing a bug. It's as simple as:

```
$ pip install pytz
```

: The default `settings.py` file created by *django-admin startproject* includes *USE_TZ = True* for convenience.

: There is also an independent but related *USE_L10N* setting that controls whether Django should activate format localization. See Format localization for more details.

If you're wrestling with a particular problem, start with the *time zone FAQ*.

Concepts

Naive and aware datetime objects

Python's `datetime.datetime` objects have a `tzinfo` attribute that can be used to store time zone information, represented as an instance of a subclass of `datetime.tzinfo`. When this attribute is set and describes an offset, a datetime object is **aware**. Otherwise, it's **naive**.

You can use *is_aware()* and *is_naive()* to determine whether datetimes are aware or naive.

When time zone support is disabled, Django uses naive datetime objects in local time. This is simple and sufficient for many use cases. In this mode, to obtain the current time, you would write:

```
import datetime

now = datetime.datetime.now()
```

When time zone support is enabled (*USE_TZ=True*), Django uses time-zone-aware datetime objects. If your code creates datetime objects, they should be aware too. In this mode, the example above becomes:

```
from django.utils import timezone

now = timezone.now()
```

: Dealing with aware datetime objects isn't always intuitive. For instance, the `tzinfo` argument of the standard datetime constructor doesn't work reliably for time zones with DST. Using UTC is generally safe; if you're using other time zones, you should review the pytz documentation carefully.

: Python's `datetime.time` objects also feature a `tzinfo` attribute, and PostgreSQL has a matching `time with time zone` type. However, as PostgreSQL's docs put it, this type "exhibits properties which lead to questionable usefulness".

Django only supports naive time objects and will raise an exception if you attempt to save an aware time object, as a timezone for a time with no associated date does not make sense.

Interpretation of naive datetime objects

When *`USE_TZ`* is `True`, Django still accepts naive datetime objects, in order to preserve backwards-compatibility. When the database layer receives one, it attempts to make it aware by interpreting it in the *default time zone* and raises a warning.

Unfortunately, during DST transitions, some datetimes don't exist or are ambiguous. In such situations, pytz raises an exception. Other `tzinfo` implementations, such as the local time zone used as a fallback when pytz isn't installed, may raise an exception or return inaccurate results. That's why you should always create aware datetime objects when time zone support is enabled.

In practice, this is rarely an issue. Django gives you aware datetime objects in the models and forms, and most often, new datetime objects are created from existing ones through `timedelta` arithmetic. The only datetime that's often created in application code is the current time, and *`timezone.now()`* automatically does the right thing.

Default time zone and current time zone

The **default time zone** is the time zone defined by the *`TIME_ZONE`* setting.

The **current time zone** is the time zone that's used for rendering.

You should set the current time zone to the end user's actual time zone with *`activate()`*. Otherwise, the default time zone is used.

: As explained in the documentation of *`TIME_ZONE`*, Django sets environment variables so that its process runs in the default time zone. This happens regardless of the value of *`USE_TZ`* and of the current time zone.

When *`USE_TZ`* is `True`, this is useful to preserve backwards-compatibility with applications that still rely on local time. However, *as explained above*, this isn't entirely reliable, and you should always work with aware datetimes in UTC in your own code. For instance, use `fromtimestamp()` and set the `tz` parameter to *`utc`*.

Selecting the current time zone

The current time zone is the equivalent of the current *locale* for translations. However, there's no equivalent of the `Accept-Language` HTTP header that Django could use to determine the user's time zone automatically. Instead,

Django provides *time zone selection functions*. Use them to build the time zone selection logic that makes sense for you.

Most websites that care about time zones just ask users in which time zone they live and store this information in the user's profile. For anonymous users, they use the time zone of their primary audience or UTC. pytz provides helpers, like a list of time zones per country, that you can use to pre-select the most likely choices.

Here's an example that stores the current timezone in the session. (It skips error handling entirely for the sake of simplicity.)

Add the following middleware to *MIDDLEWARE*:

```
import pytz

from django.utils import timezone
from django.utils.deprecation import MiddlewareMixin

class TimezoneMiddleware(MiddlewareMixin):
    def process_request(self, request):
        tzname = request.session.get('django_timezone')
        if tzname:
            timezone.activate(pytz.timezone(tzname))
        else:
            timezone.deactivate()
```

Create a view that can set the current timezone:

```
from django.shortcuts import redirect, render

def set_timezone(request):
    if request.method == 'POST':
        request.session['django_timezone'] = request.POST['timezone']
        return redirect('/')
    else:
        return render(request, 'template.html', {'timezones': pytz.common_timezones})
```

Include a form in `template.html` that will `POST` to this view:

```
{% load tz %}
{% get_current_timezone as TIME_ZONE %}
<form action="{% url 'set_timezone' %}" method="POST">
    {% csrf_token %}
    <label for="timezone">Time zone:</label>
    <select name="timezone">
        {% for tz in timezones %}
        <option value="{{ tz }}"{% if tz == TIME_ZONE %} selected="selected"{% endif %}>{{ tz }}</opt
        {% endfor %}
    </select>
    <input type="submit" value="Set" />
</form>
```

Time zone aware input in forms

When you enable time zone support, Django interprets datetimes entered in forms in the *current time zone* and returns aware datetime objects in `cleaned_data`.

If the current time zone raises an exception for datetimes that don't exist or are ambiguous because they fall in a DST transition (the timezones provided by pytz do this), such datetimes will be reported as invalid values.

Time zone aware output in templates

When you enable time zone support, Django converts aware datetime objects to the *current time zone* when they're rendered in templates. This behaves very much like format localization.

: Django doesn't convert naive datetime objects, because they could be ambiguous, and because your code should never produce naive datetimes when time zone support is enabled. However, you can force conversion with the template filters described below.

Conversion to local time isn't always appropriate – you may be generating output for computers rather than for humans. The following filters and tags, provided by the `tz` template tag library, allow you to control the time zone conversions.

Template tags

`localtime` Enables or disables conversion of aware datetime objects to the current time zone in the contained block.

This tag has exactly the same effects as the *`USE_TZ`* setting as far as the template engine is concerned. It allows a more fine grained control of conversion.

To activate or deactivate conversion for a template block, use:

```
{% load tz %}

{% localtime on %}
    {{ value }}
{% endlocaltime %}

{% localtime off %}
    {{ value }}
{% endlocaltime %}
```

: The value of *`USE_TZ`* isn't respected inside of a `{% localtime %}` block.

`timezone` Sets or unsets the current time zone in the contained block. When the current time zone is unset, the default time zone applies.

```
{% load tz %}

{% timezone "Europe/Paris" %}
    Paris time: {{ value }}
{% endtimezone %}

{% timezone None %}
    Server time: {{ value }}
{% endtimezone %}
```

`get_current_timezone` You can get the name of the current time zone using the `get_current_timezone` tag:

```
{% get_current_timezone as TIME_ZONE %}
```

Alternatively, you can activate the *`tz()`* context processor and use the `TIME_ZONE` context variable.

Template filters

These filters accept both aware and naive datetimes. For conversion purposes, they assume that naive datetimes are in the default time zone. They always return aware datetimes.

localtime Forces conversion of a single value to the current time zone.

For example:

```
{% load tz %}

{{ value|localtime }}
```

utc Forces conversion of a single value to UTC.

For example:

```
{% load tz %}

{{ value|utc }}
```

timezone Forces conversion of a single value to an arbitrary timezone.

The argument must be an instance of a `tzinfo` subclass or a time zone name. If it is a time zone name, pytz is required.

For example:

```
{% load tz %}

{{ value|timezone:"Europe/Paris" }}
```

Migration guide

Here's how to migrate a project that was started before Django supported time zones.

Database

PostgreSQL The PostgreSQL backend stores datetimes as `timestamp with time zone`. In practice, this means it converts datetimes from the connection's time zone to UTC on storage, and from UTC to the connection's time zone on retrieval.

As a consequence, if you're using PostgreSQL, you can switch between `USE_TZ = False` and `USE_TZ = True` freely. The database connection's time zone will be set to *`TIME_ZONE`* or `UTC` respectively, so that Django obtains correct datetimes in all cases. You don't need to perform any data conversions.

Other databases Other backends store datetimes without time zone information. If you switch from `USE_TZ = False` to `USE_TZ = True`, you must convert your data from local time to UTC – which isn't deterministic if your local time has DST.

Code

The first step is to add *USE_TZ = True* to your settings file and install pytz (if possible). At this point, things should mostly work. If you create naive datetime objects in your code, Django makes them aware when necessary.

However, these conversions may fail around DST transitions, which means you aren't getting the full benefits of time zone support yet. Also, you're likely to run into a few problems because it's impossible to compare a naive datetime with an aware datetime. Since Django now gives you aware datetimes, you'll get exceptions wherever you compare a datetime that comes from a model or a form with a naive datetime that you've created in your code.

So the second step is to refactor your code wherever you instantiate datetime objects to make them aware. This can be done incrementally. *django.utils.timezone* defines some handy helpers for compatibility code: *now()*, *is_aware()*, *is_naive()*, *make_aware()*, and *make_naive()*.

Finally, in order to help you locate code that needs upgrading, Django raises a warning when you attempt to save a naive datetime to the database:

```
RuntimeWarning: DateTimeField ModelName.field_name received a naive
datetime (2012-01-01 00:00:00) while time zone support is active.
```

During development, you can turn such warnings into exceptions and get a traceback by adding the following to your settings file:

```
import warnings
warnings.filterwarnings(
    'error', r"DateTimeField .* received a naive datetime",
    RuntimeWarning, r'django\.db\.models\.fields',
)
```

Fixtures

When serializing an aware datetime, the UTC offset is included, like this:

```
"2011-09-01T13:20:30+03:00"
```

For a naive datetime, it obviously isn't:

```
"2011-09-01T13:20:30"
```

For models with *DateTimeField*s, this difference makes it impossible to write a fixture that works both with and without time zone support.

Fixtures generated with `USE_TZ = False`, or before Django 1.4, use the "naive" format. If your project contains such fixtures, after you enable time zone support, you'll see `RuntimeWarning`s when you load them. To get rid of the warnings, you must convert your fixtures to the "aware" format.

You can regenerate fixtures with *loaddata* then *dumpdata*. Or, if they're small enough, you can simply edit them to add the UTC offset that matches your *TIME_ZONE* to each serialized datetime.

FAQ

Setup

1. **I don't need multiple time zones. Should I enable time zone support?**

   Yes. When time zone support is enabled, Django uses a more accurate model of local time. This shields you from subtle and unreproducible bugs around Daylight Saving Time (DST) transitions.

In this regard, time zones are comparable to `unicode` in Python. At first it's hard. You get encoding and decoding errors. Then you learn the rules. And some problems disappear – you never get mangled output again when your application receives non-ASCII input.

When you enable time zone support, you'll encounter some errors because you're using naive datetimes where Django expects aware datetimes. Such errors show up when running tests and they're easy to fix. You'll quickly learn how to avoid invalid operations.

On the other hand, bugs caused by the lack of time zone support are much harder to prevent, diagnose and fix. Anything that involves scheduled tasks or datetime arithmetic is a candidate for subtle bugs that will bite you only once or twice a year.

For these reasons, time zone support is enabled by default in new projects, and you should keep it unless you have a very good reason not to.

2. **I've enabled time zone support. Am I safe?**

   Maybe. You're better protected from DST-related bugs, but you can still shoot yourself in the foot by carelessly turning naive datetimes into aware datetimes, and vice-versa.

   If your application connects to other systems – for instance, if it queries a Web service – make sure datetimes are properly specified. To transmit datetimes safely, their representation should include the UTC offset, or their values should be in UTC (or both!).

   Finally, our calendar system contains interesting traps for computers:

```
>>> import datetime
>>> def one_year_before(value):        # DON'T DO THAT!
...     return value.replace(year=value.year - 1)
>>> one_year_before(datetime.datetime(2012, 3, 1, 10, 0))
datetime.datetime(2011, 3, 1, 10, 0)
>>> one_year_before(datetime.datetime(2012, 2, 29, 10, 0))
Traceback (most recent call last):
...
ValueError: day is out of range for month
```

   (To implement this function, you must decide whether 2012-02-29 minus one year is 2011-02-28 or 2011-03-01, which depends on your business requirements.)

3. **Should I install pytz?**

   Yes. Django has a policy of not requiring external dependencies, and for this reason pytz is optional. However, it's much safer to install it.

   As soon as you activate time zone support, Django needs a definition of the default time zone. When pytz is available, Django loads this definition from the tz database. This is the most accurate solution. Otherwise, it relies on the difference between local time and UTC, as reported by the operating system, to compute conversions. This is less reliable, especially around DST transitions.

   Furthermore, if you want to support users in more than one time zone, pytz is the reference for time zone definitions.

4. **How do I interact with a database that stores datetimes in local time?**

   Set the *`TIME_ZONE`* option to the appropriate time zone for this database in the *`DATABASES`* setting.

   This is useful for connecting to a database that doesn't support time zones and that isn't managed by Django when *`USE_TZ`* is `True`.

Troubleshooting

1. **My application crashes with `TypeError: can't compare offset-naive and offset-aware datetimes` – what's wrong?**

   Let's reproduce this error by comparing a naive and an aware datetime:

```
>>> import datetime
>>> from django.utils import timezone
>>> naive = datetime.datetime.utcnow()
>>> aware = timezone.now()
>>> naive == aware
Traceback (most recent call last):
...
TypeError: can't compare offset-naive and offset-aware datetimes
```

   If you encounter this error, most likely your code is comparing these two things:

   - a datetime provided by Django – for instance, a value read from a form or a model field. Since you enabled time zone support, it's aware.
   - a datetime generated by your code, which is naive (or you wouldn't be reading this).

   Generally, the correct solution is to change your code to use an aware datetime instead.

   If you're writing a pluggable application that's expected to work independently of the value of *`USE_TZ`*, you may find *`django.utils.timezone.now()`* useful. This function returns the current date and time as a naive datetime when `USE_TZ = False` and as an aware datetime when `USE_TZ = True`. You can add or subtract `datetime.timedelta` as needed.

2. **I see lots of `RuntimeWarning: DateTimeField received a naive datetime (YYYY-MM-DD HH:MM:SS) while time zone support is active` – is that bad?**

   When time zone support is enabled, the database layer expects to receive only aware datetimes from your code. This warning occurs when it receives a naive datetime. This indicates that you haven't finished porting your code for time zone support. Please refer to the *migration guide* for tips on this process.

   In the meantime, for backwards compatibility, the datetime is considered to be in the default time zone, which is generally what you expect.

3. **`now.date()` is yesterday! (or tomorrow)**

   If you've always used naive datetimes, you probably believe that you can convert a datetime to a date by calling its `date()` method. You also consider that a `date` is a lot like a `datetime`, except that it's less accurate.

   None of this is true in a time zone aware environment:

```
>>> import datetime
>>> import pytz
>>> paris_tz = pytz.timezone("Europe/Paris")
>>> new_york_tz = pytz.timezone("America/New_York")
>>> paris = paris_tz.localize(datetime.datetime(2012, 3, 3, 1, 30))
# This is the correct way to convert between time zones with pytz.
>>> new_york = new_york_tz.normalize(paris.astimezone(new_york_tz))
>>> paris == new_york, paris.date() == new_york.date()
(True, False)
>>> paris - new_york, paris.date() - new_york.date()
(datetime.timedelta(0), datetime.timedelta(1))
>>> paris
datetime.datetime(2012, 3, 3, 1, 30, tzinfo=<DstTzInfo 'Europe/Paris' CET+1:00:00 STD>)
>>> new_york
datetime.datetime(2012, 3, 2, 19, 30, tzinfo=<DstTzInfo 'America/New_York' EST-1 day, 19:00:00 S
```

As this example shows, the same datetime has a different date, depending on the time zone in which it is represented. But the real problem is more fundamental.

A datetime represents a **point in time**. It's absolute: it doesn't depend on anything. On the contrary, a date is a **calendaring concept**. It's a period of time whose bounds depend on the time zone in which the date is considered. As you can see, these two concepts are fundamentally different, and converting a datetime to a date isn't a deterministic operation.

What does this mean in practice?

Generally, you should avoid converting a `datetime` to `date`. For instance, you can use the *`date`* template filter to only show the date part of a datetime. This filter will convert the datetime into the current time zone before formatting it, ensuring the results appear correctly.

If you really need to do the conversion yourself, you must ensure the datetime is converted to the appropriate time zone first. Usually, this will be the current timezone:

```
>>> from django.utils import timezone
>>> timezone.activate(pytz.timezone("Asia/Singapore"))
# For this example, we just set the time zone to Singapore, but here's how
# you would obtain the current time zone in the general case.
>>> current_tz = timezone.get_current_timezone()
# Again, this is the correct way to convert between time zones with pytz.
>>> local = current_tz.normalize(paris.astimezone(current_tz))
>>> local
datetime.datetime(2012, 3, 3, 8, 30, tzinfo=<DstTzInfo 'Asia/Singapore' SGT+8:00:00 STD>)
>>> local.date()
datetime.date(2012, 3, 3)
```

4. **I get an error** "`Are time zone definitions for your database and pytz installed?`" **pytz is installed, so I guess the problem is my database?**

   If you are using MySQL, see the *Time zone definitions* section of the MySQL notes for instructions on loading time zone definitions.

Usage

1. **I have a string** `"2012-02-21 10:28:45"` **and I know it's in the** `"Europe/Helsinki"` **time zone. How do I turn that into an aware datetime?**

   This is exactly what pytz is for.

```
>>> from django.utils.dateparse import parse_datetime
>>> naive = parse_datetime("2012-02-21 10:28:45")
>>> import pytz
>>> pytz.timezone("Europe/Helsinki").localize(naive, is_dst=None)
datetime.datetime(2012, 2, 21, 10, 28, 45, tzinfo=<DstTzInfo 'Europe/Helsinki' EET+2:00:00 STD>)
```

   Note that `localize` is a pytz extension to the `tzinfo` API. Also, you may want to catch `pytz.InvalidTimeError`. The documentation of pytz contains more examples. You should review it before attempting to manipulate aware datetimes.

2. **How can I obtain the local time in the current time zone?**

   Well, the first question is, do you really need to?

   You should only use local time when you're interacting with humans, and the template layer provides *filters and tags* to convert datetimes to the time zone of your choice.

Furthermore, Python knows how to compare aware datetimes, taking into account UTC offsets when necessary. It's much easier (and possibly faster) to write all your model and view code in UTC. So, in most circumstances, the datetime in UTC returned by *`django.utils.timezone.now()`* will be sufficient.

For the sake of completeness, though, if you really want the local time in the current time zone, here's how you can obtain it:

```
>>> from django.utils import timezone
>>> timezone.localtime(timezone.now())
datetime.datetime(2012, 3, 3, 20, 10, 53, 873365, tzinfo=<DstTzInfo 'Europe/Paris' CET+1:00:00 S
```

In this example, pytz is installed and the current time zone is `"Europe/Paris"`.

3. **How can I see all available time zones?**

   pytz provides helpers, including a list of current time zones and a list of all available time zones – some of which are only of historical interest.

3.15.4 Overview

The goal of internationalization and localization is to allow a single Web application to offer its content in languages and formats tailored to the audience.

Django has full support for translation of text, formatting of dates, times and numbers, and time zones.

Essentially, Django does two things:

- It allows developers and template authors to specify which parts of their apps should be translated or formatted for local languages and cultures.
- It uses these hooks to localize Web apps for particular users according to their preferences.

Obviously, translation depends on the target language, and formatting usually depends on the target country. This information is provided by browsers in the `Accept-Language` header. However, the time zone isn't readily available.

3.15.5 Definitions

The words "internationalization" and "localization" often cause confusion; here's a simplified definition:

internationalization Preparing the software for localization. Usually done by developers.

localization Writing the translations and local formats. Usually done by translators.

More details can be found in the W3C Web Internationalization FAQ, the Wikipedia article or the GNU gettext documentation.

: Translation and formatting are controlled by *`USE_I18N`* and *`USE_L10N`* settings respectively. However, both features involve internationalization and localization. The names of the settings are an unfortunate result of Django's history.

Here are some other terms that will help us to handle a common language:

locale name A locale name, either a language specification of the form `ll` or a combined language and country specification of the form `ll_CC`. Examples: `it`, `de_AT`, `es`, `pt_BR`. The language part is always in lower case and the country part in upper case. The separator is an underscore.

language code Represents the name of a language. Browsers send the names of the languages they accept in the `Accept-Language` HTTP header using this format. Examples: `it`, `de-at`, `es`, `pt-br`. Language codes

are generally represented in lower-case, but the HTTP `Accept-Language` header is case-insensitive. The separator is a dash.

message file A message file is a plain-text file, representing a single language, that contains all available *translation strings* and how they should be represented in the given language. Message files have a `.po` file extension.

translation string A literal that can be translated.

format file A format file is a Python module that defines the data formats for a given locale.

3.16 Logging

3.16.1 A quick logging primer

Django uses Python's builtin `logging` module to perform system logging. The usage of this module is discussed in detail in Python's own documentation. However, if you've never used Python's logging framework (or even if you have), here's a quick primer.

The cast of players

A Python logging configuration consists of four parts:

- *Loggers*
- *Handlers*
- *Filters*
- *Formatters*

Loggers

A logger is the entry point into the logging system. Each logger is a named bucket to which messages can be written for processing.

A logger is configured to have a *log level*. This log level describes the severity of the messages that the logger will handle. Python defines the following log levels:

- `DEBUG`: Low level system information for debugging purposes
- `INFO`: General system information
- `WARNING`: Information describing a minor problem that has occurred.
- `ERROR`: Information describing a major problem that has occurred.
- `CRITICAL`: Information describing a critical problem that has occurred.

Each message that is written to the logger is a *Log Record*. Each log record also has a *log level* indicating the severity of that specific message. A log record can also contain useful metadata that describes the event that is being logged. This can include details such as a stack trace or an error code.

When a message is given to the logger, the log level of the message is compared to the log level of the logger. If the log level of the message meets or exceeds the log level of the logger itself, the message will undergo further processing. If it doesn't, the message will be ignored.

Once a logger has determined that a message needs to be processed, it is passed to a *Handler*.

Handlers

The handler is the engine that determines what happens to each message in a logger. It describes a particular logging behavior, such as writing a message to the screen, to a file, or to a network socket.

Like loggers, handlers also have a log level. If the log level of a log record doesn't meet or exceed the level of the handler, the handler will ignore the message.

A logger can have multiple handlers, and each handler can have a different log level. In this way, it is possible to provide different forms of notification depending on the importance of a message. For example, you could install one handler that forwards `ERROR` and `CRITICAL` messages to a paging service, while a second handler logs all messages (including `ERROR` and `CRITICAL` messages) to a file for later analysis.

Filters

A filter is used to provide additional control over which log records are passed from logger to handler.

By default, any log message that meets log level requirements will be handled. However, by installing a filter, you can place additional criteria on the logging process. For example, you could install a filter that only allows `ERROR` messages from a particular source to be emitted.

Filters can also be used to modify the logging record prior to being emitted. For example, you could write a filter that downgrades `ERROR` log records to `WARNING` records if a particular set of criteria are met.

Filters can be installed on loggers or on handlers; multiple filters can be used in a chain to perform multiple filtering actions.

Formatters

Ultimately, a log record needs to be rendered as text. Formatters describe the exact format of that text. A formatter usually consists of a Python formatting string containing LogRecord attributes; however, you can also write custom formatters to implement specific formatting behavior.

3.16.2 Using logging

Once you have configured your loggers, handlers, filters and formatters, you need to place logging calls into your code. Using the logging framework is very simple. Here's an example:

```
# import the logging library
import logging

# Get an instance of a logger
logger = logging.getLogger(__name__)

def my_view(request, arg1, arg):
    ...
    if bad_mojo:
        # Log an error message
        logger.error('Something went wrong!')
```

And that's it! Every time the `bad_mojo` condition is activated, an error log record will be written.

Naming loggers

The call to `logging.getLogger()` obtains (creating, if necessary) an instance of a logger. The logger instance is identified by a name. This name is used to identify the logger for configuration purposes.

By convention, the logger name is usually `__name__`, the name of the python module that contains the logger. This allows you to filter and handle logging calls on a per-module basis. However, if you have some other way of organizing your logging messages, you can provide any dot-separated name to identify your logger:

```
# Get an instance of a specific named logger
logger = logging.getLogger('project.interesting.stuff')
```

The dotted paths of logger names define a hierarchy. The `project.interesting` logger is considered to be a parent of the `project.interesting.stuff` logger; the `project` logger is a parent of the `project.interesting` logger.

Why is the hierarchy important? Well, because loggers can be set to *propagate* their logging calls to their parents. In this way, you can define a single set of handlers at the root of a logger tree, and capture all logging calls in the subtree of loggers. A logging handler defined in the `project` namespace will catch all logging messages issued on the `project.interesting` and `project.interesting.stuff` loggers.

This propagation can be controlled on a per-logger basis. If you don't want a particular logger to propagate to its parents, you can turn off this behavior.

Making logging calls

The logger instance contains an entry method for each of the default log levels:

- `logger.debug()`
- `logger.info()`
- `logger.warning()`
- `logger.error()`
- `logger.critical()`

There are two other logging calls available:

- `logger.log()`: Manually emits a logging message with a specific log level.
- `logger.exception()`: Creates an `ERROR` level logging message wrapping the current exception stack frame.

3.16.3 Configuring logging

Of course, it isn't enough to just put logging calls into your code. You also need to configure the loggers, handlers, filters and formatters to ensure that logging output is output in a useful way.

Python's logging library provides several techniques to configure logging, ranging from a programmatic interface to configuration files. By default, Django uses the dictConfig format.

In order to configure logging, you use *`LOGGING`* to define a dictionary of logging settings. These settings describes the loggers, handlers, filters and formatters that you want in your logging setup, and the log levels and other properties that you want those components to have.

By default, the *`LOGGING`* setting is merged with *Django's default logging configuration* using the following scheme.

If the `disable_existing_loggers` key in the *`LOGGING`* dictConfig is set to `True` (which is the default) then all loggers from the default configuration will be disabled. Disabled loggers are not the same as removed; the logger

will still exist, but will silently discard anything logged to it, not even propagating entries to a parent logger. Thus you should be very careful using `'disable_existing_loggers': True`; it's probably not what you want. Instead, you can set `disable_existing_loggers` to `False` and redefine some or all of the default loggers; or you can set *`LOGGING_CONFIG`* to `None` and *handle logging config yourself*.

Logging is configured as part of the general Django `setup()` function. Therefore, you can be certain that loggers are always ready for use in your project code.

Examples

The full documentation for dictConfig format is the best source of information about logging configuration dictionaries. However, to give you a taste of what is possible, here are several examples.

First, here's a simple configuration which writes all logging from the *django* logger to a local file:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'file': {
            'level': 'DEBUG',
            'class': 'logging.FileHandler',
            'filename': '/path/to/django/debug.log',
        },
    },
    'loggers': {
        'django': {
            'handlers': ['file'],
            'level': 'DEBUG',
            'propagate': True,
        },
    },
}
```

If you use this example, be sure to change the `'filename'` path to a location that's writable by the user that's running the Django application.

Second, here's an example of how to make the logging system print Django's logging to the console. It may be useful during local development.

By default, this config only sends messages of level `INFO` or higher to the console (same as Django's default logging config, except that the default only displays log records when `DEBUG=True`). Django does not log many such messages. With this config, however, you can also set the environment variable `DJANGO_LOG_LEVEL=DEBUG` to see all of Django's debug logging which is very verbose as it includes all database queries:

```
import os

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
        },
    },
    'loggers': {
        'django': {
            'handlers': ['console'],
            'level': os.getenv('DJANGO_LOG_LEVEL', 'INFO'),
```

```
        },
    },
}
```

Django's default logging configuration changed. See *the release notes* for a description of the changes.

Finally, here's an example of a fairly complex logging setup:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'verbose': {
            'format': '%(levelname)s %(asctime)s %(module)s %(process)d %(thread)d %(message)s'
        },
        'simple': {
            'format': '%(levelname)s %(message)s'
        },
    },
    'filters': {
        'special': {
            '()': 'project.logging.SpecialFilter',
            'foo': 'bar',
        },
        'require_debug_true': {
            '()': 'django.utils.log.RequireDebugTrue',
        },
    },
    'handlers': {
        'console': {
            'level': 'INFO',
            'filters': ['require_debug_true'],
            'class': 'logging.StreamHandler',
            'formatter': 'simple'
        },
        'mail_admins': {
            'level': 'ERROR',
            'class': 'django.utils.log.AdminEmailHandler',
            'filters': ['special']
        }
    },
    'loggers': {
        'django': {
            'handlers': ['console'],
            'propagate': True,
        },
        'django.request': {
            'handlers': ['mail_admins'],
            'level': 'ERROR',
            'propagate': False,
        },
        'myproject.custom': {
            'handlers': ['console', 'mail_admins'],
            'level': 'INFO',
            'filters': ['special']
        }
    }
}
```

This logging configuration does the following things:

- Identifies the configuration as being in 'dictConfig version 1' format. At present, this is the only dictConfig format version.
- Defines two formatters:
  - `simple`, that just outputs the log level name (e.g., `DEBUG`) and the log message.

    The `format` string is a normal Python formatting string describing the details that are to be output on each logging line. The full list of detail that can be output can be found in Formatter Objects.
  - `verbose`, that outputs the log level name, the log message, plus the time, process, thread and module that generate the log message.
- Defines two filters:
  - `project.logging.SpecialFilter`, using the alias `special`. If this filter required additional arguments, they can be provided as additional keys in the filter configuration dictionary. In this case, the argument `foo` will be given a value of `bar` when instantiating `SpecialFilter`.
  - `django.utils.log.RequireDebugTrue`, which passes on records when *DEBUG* is `True`.
- Defines two handlers:
  - `console`, a StreamHandler, which will print any `DEBUG` (or higher) message to stderr. This handler uses the `simple` output format.
  - `mail_admins`, an AdminEmailHandler, which will email any `ERROR` (or higher) message to the site admins. This handler uses the `special` filter.
- Configures three loggers:
  - `django`, which passes all messages to the `console` handler.
  - `django.request`, which passes all `ERROR` messages to the `mail_admins` handler. In addition, this logger is marked to *not* propagate messages. This means that log messages written to `django.request` will not be handled by the `django` logger.
  - `myproject.custom`, which passes all messages at `INFO` or higher that also pass the `special` filter to two handlers – the `console`, and `mail_admins`. This means that all `INFO` level messages (or higher) will be printed to the console; `ERROR` and `CRITICAL` messages will also be output via email.

Custom logging configuration

If you don't want to use Python's dictConfig format to configure your logger, you can specify your own configuration scheme.

The *LOGGING_CONFIG* setting defines the callable that will be used to configure Django's loggers. By default, it points at Python's `logging.config.dictConfig()` function. However, if you want to use a different configuration process, you can use any other callable that takes a single argument. The contents of *LOGGING* will be provided as the value of that argument when logging is configured.

Disabling logging configuration

If you don't want to configure logging at all (or you want to manually configure logging using your own approach), you can set *LOGGING_CONFIG* to `None`. This will disable the configuration process for *Django's default logging*. Here's an example that disables Django's logging configuration and then manually configures logging:

```
settings.py
```

```
LOGGING_CONFIG = None

import logging.config
logging.config.dictConfig(...)
```

Setting *LOGGING_CONFIG* to `None` only means that the automatic configuration process is disabled, not logging itself. If you disable the configuration process, Django will still make logging calls, falling back to whatever default logging behavior is defined.

3.16.4 Django's logging extensions

Django provides a number of utilities to handle the unique requirements of logging in Web server environment.

Loggers

Django provides several built-in loggers.

django

`django` is the catch-all logger. No messages are posted directly to this logger.

django.request

Log messages related to the handling of requests. 5XX responses are raised as `ERROR` messages; 4XX responses are raised as `WARNING` messages.

Messages to this logger have the following extra context:

- `status_code`: The HTTP response code associated with the request.
- `request`: The request object that generated the logging message.

django.server

Log messages related to the handling of requests received by the server invoked by the *runserver* command. HTTP 5XX responses are logged as `ERROR` messages, 4XX responses are logged as `WARNING` messages, and everything else is logged as `INFO`.

Messages to this logger have the following extra context:

- `status_code`: The HTTP response code associated with the request.
- `request`: The request object that generated the logging message.

django.template

Log messages related to the rendering of templates.

- Missing context variables are logged as `DEBUG` messages.
- Uncaught exceptions raised during the rendering of an *{% include %}* are logged as `WARNING` messages when debug mode is off (helpful since `{% include %}` silences the exception and returns an empty string in that case).

django.db.backends

Messages relating to the interaction of code with the database. For example, every application-level SQL statement executed by a request is logged at the DEBUG level to this logger.

Messages to this logger have the following extra context:

- duration: The time taken to execute the SQL statement.
- sql: The SQL statement that was executed.
- params: The parameters that were used in the SQL call.

For performance reasons, SQL logging is only enabled when settings.DEBUG is set to True, regardless of the logging level or handlers that are installed.

This logging does not include framework-level initialization (e.g. SET TIMEZONE) or transaction management queries (e.g. BEGIN, COMMIT, and ROLLBACK). Turn on query logging in your database if you wish to view all database queries.

django.security.*

The security loggers will receive messages on any occurrence of *SuspiciousOperation*. There is a sub-logger for each sub-type of SuspiciousOperation. The level of the log event depends on where the exception is handled. Most occurrences are logged as a warning, while any SuspiciousOperation that reaches the WSGI handler will be logged as an error. For example, when an HTTP Host header is included in a request from a client that does not match *ALLOWED_HOSTS*, Django will return a 400 response, and an error message will be logged to the django.security.DisallowedHost logger.

These log events will reach the 'django' logger by default, which mails error events to admins when DEBUG=False. Requests resulting in a 400 response due to a SuspiciousOperation will not be logged to the django.request logger, but only to the django.security logger.

To silence a particular type of SuspiciousOperation, you can override that specific logger following this example:

```
'handlers': {
    'null': {
        'class': 'logging.NullHandler',
    },
},
'loggers': {
    'django.security.DisallowedHost': {
        'handlers': ['null'],
        'propagate': False,
    },
},
```

django.db.backends.schema

Logs the SQL queries that are executed during schema changes to the database by the migrations framework. Note that it won't log the queries executed by *RunPython*. Messages to this logger have params and sql in their extra context (but unlike django.db.backends, not duration). The values have the same meaning as explained in *django.db.backends*.

The extra context was added.

Handlers

Django provides one log handler in addition to those provided by the Python logging module.

class **AdminEmailHandler**(*include_html=False*, *email_backend=None*)

This handler sends an email to the site admins for each log message it receives.

If the log record contains a `request` attribute, the full details of the request will be included in the email. The email subject will include the phrase "internal IP" if the client's IP address is in the *INTERNAL_IPS* setting; if not, it will include "EXTERNAL IP".

If the log record contains stack trace information, that stack trace will be included in the email.

The `include_html` argument of `AdminEmailHandler` is used to control whether the traceback email includes an HTML attachment containing the full content of the debug Web page that would have been produced if *DEBUG* were `True`. To set this value in your configuration, include it in the handler definition for `django.utils.log.AdminEmailHandler`, like this:

```
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'class': 'django.utils.log.AdminEmailHandler',
        'include_html': True,
    }
},
```

Note that this HTML version of the email contains a full traceback, with names and values of local variables at each level of the stack, plus the values of your Django settings. This information is potentially very sensitive, and you may not want to send it over email. Consider using something such as Sentry to get the best of both worlds – the rich information of full tracebacks plus the security of *not* sending the information over email. You may also explicitly designate certain sensitive information to be filtered out of error reports – learn more on *Filtering error reports*.

By setting the `email_backend` argument of `AdminEmailHandler`, the *email backend* that is being used by the handler can be overridden, like this:

```
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'class': 'django.utils.log.AdminEmailHandler',
        'email_backend': 'django.core.mail.backends.filebased.EmailBackend',
    }
},
```

By default, an instance of the email backend specified in *EMAIL_BACKEND* will be used.

send_mail(*subject*, *message*, **args*, ***kwargs*)

Sends emails to admin users. To customize this behavior, you can subclass the *AdminEmailHandler* class and override this method.

Filters

Django provides two log filters in addition to those provided by the Python logging module.

class **CallbackFilter**(*callback*)

This filter accepts a callback function (which should accept a single argument, the record to be logged), and calls it for each record that passes through the filter. Handling of that record will not proceed if the callback returns False.

For instance, to filter out *UnreadablePostError* (raised when a user cancels an upload) from the admin emails, you would create a filter function:

```
from django.http import UnreadablePostError

def skip_unreadable_post(record):
    if record.exc_info:
        exc_type, exc_value = record.exc_info[:2]
        if isinstance(exc_value, UnreadablePostError):
            return False
    return True
```

and then add it to your logging config:

```
'filters': {
    'skip_unreadable_posts': {
        '()': 'django.utils.log.CallbackFilter',
        'callback': skip_unreadable_post,
    }
},
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'filters': ['skip_unreadable_posts'],
        'class': 'django.utils.log.AdminEmailHandler'
    }
},
```

class RequireDebugFalse

This filter will only pass on records when settings.DEBUG is False.

This filter is used as follows in the default *LOGGING* configuration to ensure that the *AdminEmailHandler* only sends error emails to admins when *DEBUG* is `False`:

```
'filters': {
    'require_debug_false': {
        '()': 'django.utils.log.RequireDebugFalse',
    }
},
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'filters': ['require_debug_false'],
        'class': 'django.utils.log.AdminEmailHandler'
    }
},
```

class RequireDebugTrue

This filter is similar to *RequireDebugFalse*, except that records are passed only when *DEBUG* is `True`.

3.16.5 Django's default logging configuration

By default, Django configures the following logging:

When *DEBUG* is `True`:

- The `django` catch-all logger sends all messages at the `INFO` level or higher to the console.

When *DEBUG* is `False`:

- The `django` logger send messages with `ERROR` or `CRITICAL` level to *`AdminEmailHandler`*.

Independent of the value of *`DEBUG`*:

- The *django.server* logger sends all messages at the `INFO` level or higher to the console.

Django's default logging configuration changed. See *the release notes* for a description of the changes.

See also *Configuring logging* to learn how you can complement or replace this default logging configuration.

3.17 Pagination

Django provides a few classes that help you manage paginated data – that is, data that's split across several pages, with "Previous/Next" links. These classes live in `django/core/paginator.py`.

3.17.1 Example

Give *`Paginator`* a list of objects, plus the number of items you'd like to have on each page, and it gives you methods for accessing the items for each page:

```
>>> from django.core.paginator import Paginator
>>> objects = ['john', 'paul', 'george', 'ringo']
>>> p = Paginator(objects, 2)

>>> p.count
4
>>> p.num_pages
2
>>> type(p.page_range)  # `<type 'rangeiterator'>` in Python 2.
<class 'range_iterator'>
>>> p.page_range
range(1, 3)

>>> page1 = p.page(1)
>>> page1
<Page 1 of 2>
>>> page1.object_list
['john', 'paul']

>>> page2 = p.page(2)
>>> page2.object_list
['george', 'ringo']
>>> page2.has_next()
False
>>> page2.has_previous()
True
>>> page2.has_other_pages()
True
>>> page2.next_page_number()
Traceback (most recent call last):
...
EmptyPage: That page contains no results
>>> page2.previous_page_number()
1
>>> page2.start_index() # The 1-based index of the first item on this page
3
>>> page2.end_index() # The 1-based index of the last item on this page
```

```
4

>>> p.page(0)
Traceback (most recent call last):
...
EmptyPage: That page number is less than 1
>>> p.page(3)
Traceback (most recent call last):
...
EmptyPage: That page contains no results
```

: Note that you can give `Paginator` a list/tuple, a Django `QuerySet`, or any other object with a `count()` or `__len__()` method. When determining the number of objects contained in the passed object, `Paginator` will first try calling `count()`, then fallback to using `len()` if the passed object has no `count()` method. This allows objects such as Django's `QuerySet` to use a more efficient `count()` method when available.

3.17.2 Using `Paginator` in a view

Here's a slightly more complex example using *`Paginator`* in a view to paginate a queryset. We give both the view and the accompanying template to show how you can display the results. This example assumes you have a `Contacts` model that has already been imported.

The view function looks like this:

```
from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger
from django.shortcuts import render

def listing(request):
    contact_list = Contacts.objects.all()
    paginator = Paginator(contact_list, 25) # Show 25 contacts per page

    page = request.GET.get('page')
    try:
        contacts = paginator.page(page)
    except PageNotAnInteger:
        # If page is not an integer, deliver first page.
        contacts = paginator.page(1)
    except EmptyPage:
        # If page is out of range (e.g. 9999), deliver last page of results.
        contacts = paginator.page(paginator.num_pages)

    return render(request, 'list.html', {'contacts': contacts})
```

In the template `list.html`, you'll want to include navigation between pages along with any interesting information from the objects themselves:

```
{% for contact in contacts %}
    {# Each "contact" is a Contact model object. #}
    {{ contact.full_name|upper }}<br />
    ...
{% endfor %}

<div class="pagination">
    <span class="step-links">
        {% if contacts.has_previous %}
```

```
            <a href="?page={{ contacts.previous_page_number }}">previous</a>
        {% endif %}

        <span class="current">
            Page {{ contacts.number }} of {{ contacts.paginator.num_pages }}.
        </span>

        {% if contacts.has_next %}
            <a href="?page={{ contacts.next_page_number }}">next</a>
        {% endif %}
    </span>
</div>
```

3.17.3 `Paginator` objects

The *`Paginator`* class has this constructor:

class `Paginator`(*object_list*, *per_page*, *orphans=0*, *allow_empty_first_page=True*)

Required arguments

`object_list` A list, tuple, `QuerySet`, or other sliceable object with a `count()` or `__len__()` method. For consistent pagination, `QuerySets` should be ordered, e.g. with an *`order_by()`* clause or with a default *`ordering`* on the model.

Performance issues paginating large `QuerySets`

If you're using a `QuerySet` with a very large number of items, requesting high page numbers might be slow on some databases, because the resulting `LIMIT`/`OFFSET` query needs to count the number of `OFFSET` records which takes longer as the page number gets higher.

`per_page` The maximum number of items to include on a page, not including orphans (see the `orphans` optional argument below).

Optional arguments

`orphans` The minimum number of items allowed on the last page, defaults to zero. Use this when you don't want to have a last page with very few items. If the last page would normally have a number of items less than or equal to `orphans`, then those items will be added to the previous page (which becomes the last page) instead of leaving the items on a page by themselves. For example, with 23 items, `per_page=10`, and `orphans=3`, there will be two pages; the first page with 10 items and the second (and last) page with 13 items.

`allow_empty_first_page` Whether or not the first page is allowed to be empty. If `False` and `object_list` is empty, then an `EmptyPage` error will be raised.

Methods

`Paginator.`**`page`**(*number*)

Returns a *`Page`* object with the given 1-based index. Raises *`InvalidPage`* if the given page number doesn't exist.

Attributes

`Paginator.`**`count`**
The total number of objects, across all pages.

: When determining the number of objects contained in `object_list`, `Paginator` will first try calling `object_list.count()`. If `object_list` has no `count()` method, then `Paginator` will fallback to using `len(object_list)`. This allows objects, such as Django's `QuerySet`, to use a more efficient `count()` method when available.

`Paginator.`**`num_pages`**
The total number of pages.

`Paginator.`**`page_range`**
A 1-based range iterator of page numbers, e.g. yielding `[1, 2, 3, 4]`.

In older versions, `page_range` returned a list instead of an iterator.

3.17.4 `InvalidPage` exceptions

exception `InvalidPage`
A base class for exceptions raised when a paginator is passed an invalid page number.

The *`Paginator.page()`* method raises an exception if the requested page is invalid (i.e., not an integer) or contains no objects. Generally, it's enough to catch the `InvalidPage` exception, but if you'd like more granularity, you can catch either of the following exceptions:

exception `PageNotAnInteger`
Raised when `page()` is given a value that isn't an integer.

exception `EmptyPage`
Raised when `page()` is given a valid value but no objects exist on that page.

Both of the exceptions are subclasses of *`InvalidPage`*, so you can handle them both with a simple `except InvalidPage`.

3.17.5 `Page` objects

You usually won't construct `Page` objects by hand – you'll get them using *`Paginator.page()`*.

class `Page`(*object_list*, *number*, *paginator*)
A page acts like a sequence of *`Page.object_list`* when using `len()` or iterating it directly.

Methods

`Page.`**`has_next`**`()`
Returns `True` if there's a next page.

`Page.`**`has_previous`**`()`
Returns `True` if there's a previous page.

`Page.`**`has_other_pages`**`()`
Returns `True` if there's a next *or* previous page.

Page.**next_page_number**()
Returns the next page number. Raises *InvalidPage* if next page doesn't exist.

Page.**previous_page_number**()
Returns the previous page number. Raises *InvalidPage* if previous page doesn't exist.

Page.**start_index**()
Returns the 1-based index of the first object on the page, relative to all of the objects in the paginator's list. For example, when paginating a list of 5 objects with 2 objects per page, the second page's *start_index()* would return 3.

Page.**end_index**()
Returns the 1-based index of the last object on the page, relative to all of the objects in the paginator's list. For example, when paginating a list of 5 objects with 2 objects per page, the second page's *end_index()* would return 4.

Attributes

Page.**object_list**
The list of objects on this page.

Page.**number**
The 1-based page number for this page.

Page.**paginator**
The associated *Paginator* object.

3.18 Porting to Python 3

Django 1.5 is the first version of Django to support Python 3. The same code runs both on Python 2 ($\geq$ 2.6.5) and Python 3 ($\geq$ 3.2), thanks to the six compatibility layer.

This document is primarily targeted at authors of pluggable applications who want to support both Python 2 and 3. It also describes guidelines that apply to Django's code.

3.18.1 Philosophy

This document assumes that you are familiar with the changes between Python 2 and Python 3. If you aren't, read Python's official porting guide first. Refreshing your knowledge of unicode handling on Python 2 and 3 will help; the Pragmatic Unicode presentation is a good resource.

Django uses the *Python 2/3 Compatible Source* strategy. Of course, you're free to chose another strategy for your own code, especially if you don't need to stay compatible with Python 2. But authors of pluggable applications are encouraged to use the same porting strategy as Django itself.

Writing compatible code is much easier if you target Python $\geq$ 2.6. Django 1.5 introduces compatibility tools such as *django.utils.six*, which is a customized version of the six module. For convenience, forwards-compatible aliases were introduced in Django 1.4.2. If your application takes advantage of these tools, it will require Django $\geq$ 1.4.2.

Obviously, writing compatible source code adds some overhead, and that can cause frustration. Django's developers have found that attempting to write Python 3 code that's compatible with Python 2 is much more rewarding than the opposite. Not only does that make your code more future-proof, but Python 3's advantages (like the saner string handling) start shining quickly. Dealing with Python 2 becomes a backwards compatibility requirement, and we as developers are used to dealing with such constraints.

Porting tools provided by Django are inspired by this philosophy, and it's reflected throughout this guide.

3.18.2 Porting tips

Unicode literals

This step consists in:

- Adding `from __future__ import unicode_literals` at the top of your Python modules – it's best to put it in each and every module, otherwise you'll keep checking the top of your files to see which mode is in effect;
- Removing the `u` prefix before unicode strings;
- Adding a `b` prefix before bytestrings.

Performing these changes systematically guarantees backwards compatibility.

However, Django applications generally don't need bytestrings, since Django only exposes unicode interfaces to the programmer. Python 3 discourages using bytestrings, except for binary data or byte-oriented interfaces. Python 2 makes bytestrings and unicode strings effectively interchangeable, as long as they only contain ASCII data. Take advantage of this to use unicode strings wherever possible and avoid the `b` prefixes.

: Python 2's `u` prefix is a syntax error in Python 3.2 but it will be allowed again in Python 3.3 thanks to **PEP 414**. Thus, this transformation is optional if you target Python $\geq$ 3.3. It's still recommended, per the "write Python 3 code" philosophy.

String handling

Python 2's unicode type was renamed `str` in Python 3, `str()` was renamed `bytes`, and basestring disappeared. six provides *tools* to deal with these changes.

Django also contains several string related classes and functions in the *`django.utils.encoding`* and *`django.utils.safestring`* modules. Their names used the words `str`, which doesn't mean the same thing in Python 2 and Python 3, and `unicode`, which doesn't exist in Python 3. In order to avoid ambiguity and confusion these concepts were renamed `bytes` and `text`.

Here are the name changes in *`django.utils.encoding`*:

Old name	New name
`smart_str`	`smart_bytes`
`smart_unicode`	`smart_text`
`force_unicode`	`force_text`

For backwards compatibility, the old names still work on Python 2. Under Python 3, `smart_str` is an alias for `smart_text`.

For forwards compatibility, the new names work as of Django 1.4.2.

: *`django.utils.encoding`* was deeply refactored in Django 1.5 to provide a more consistent API. Check its documentation for more information.

`django.utils.safestring` is mostly used via the *`mark_safe()`* and *`mark_for_escaping()`* functions, which didn't change. In case you're using the internals, here are the name changes:

Old name	New name
EscapeString	EscapeBytes
EscapeUnicode	EscapeText
SafeString	SafeBytes
SafeUnicode	SafeText

For backwards compatibility, the old names still work on Python 2. Under Python 3, EscapeString and SafeString are aliases for EscapeText and SafeText respectively.

For forwards compatibility, the new names work as of Django 1.4.2.

__str__() and __unicode__() methods

In Python 2, the object model specifies __str__() and __unicode__() methods. If these methods exist, they must return str (bytes) and unicode (text) respectively.

The print statement and the str built-in call __str__() to determine the human-readable representation of an object. The unicode built-in calls __unicode__() if it exists, and otherwise falls back to __str__() and decodes the result with the system encoding. Conversely, the *Model* base class automatically derives __str__() from __unicode__() by encoding to UTF-8.

In Python 3, there's simply __str__(), which must return str (text).

(It is also possible to define __bytes__(), but Django applications have little use for that method, because they hardly ever deal with bytes.)

Django provides a simple way to define __str__() and __unicode__() methods that work on Python 2 and 3: you must define a __str__() method returning text and to apply the *python_2_unicode_compatible()* decorator.

On Python 3, the decorator is a no-op. On Python 2, it defines appropriate __unicode__() and __str__() methods (replacing the original __str__() method in the process). Here's an example:

```
from __future__ import unicode_literals
from django.utils.encoding import python_2_unicode_compatible

@python_2_unicode_compatible
class MyClass(object):
    def __str__(self):
        return "Instance of my class"
```

This technique is the best match for Django's porting philosophy.

For forwards compatibility, this decorator is available as of Django 1.4.2.

Finally, note that __repr__() must return a str on all versions of Python.

dict and dict-like classes

dict.keys(), dict.items() and dict.values() return lists in Python 2 and iterators in Python 3. *QueryDict* and the dict-like classes defined in django.utils.datastructures behave likewise in Python 3.

six provides compatibility functions to work around this change: iterkeys(), iteritems(), and itervalues(). It also contains an undocumented iterlists function that works well for django.utils.datastructures.MultiValueDict and its subclasses.

`HttpRequest` and `HttpResponse` objects

According to **PEP 3333**:

- headers are always `str` objects,
- input and output streams are always `bytes` objects.

Specifically, *`HttpResponse.content`* contains `bytes`, which may become an issue if you compare it with a `str` in your tests. The preferred solution is to rely on *`assertContains()`* and *`assertNotContains()`*. These methods accept a response and a unicode string as arguments.

3.18.3 Coding guidelines

The following guidelines are enforced in Django's source code. They're also recommended for third-party applications that follow the same porting strategy.

Syntax requirements

Unicode

In Python 3, all strings are considered Unicode by default. The `unicode` type from Python 2 is called `str` in Python 3, and `str` becomes `bytes`.

You mustn't use the `u` prefix before a unicode string literal because it's a syntax error in Python 3.2. You must prefix byte strings with `b`.

In order to enable the same behavior in Python 2, every module must import `unicode_literals` from `__future__`:

```
from __future__ import unicode_literals

my_string = "This is an unicode literal"
my_bytestring = b"This is a bytestring"
```

If you need a byte string literal under Python 2 and a unicode string literal under Python 3, use the `str` builtin:

```
str('my string')
```

In Python 3, there aren't any automatic conversions between `str` and `bytes`, and the `codecs` module became more strict. `str.encode()` always returns `bytes`, and `bytes.decode` always returns `str`. As a consequence, the following pattern is sometimes necessary:

```
value = value.encode('ascii', 'ignore').decode('ascii')
```

Be cautious if you have to index bytestrings.

Exceptions

When you capture exceptions, use the `as` keyword:

```
try:
    ...
except MyException as exc:
    ...
```

This older syntax was removed in Python 3:

```
try:
    ...
except MyException, exc:    # Don't do that!
    ...
```

The syntax to reraise an exception with a different traceback also changed. Use `six.reraise()`.

Magic methods

Use the patterns below to handle magic methods renamed in Python 3.

Iterators

```
class MyIterator(six.Iterator):
    def __iter__(self):
        return self             # implement some logic here

    def __next__(self):
        raise StopIteration     # implement some logic here
```

Boolean evaluation

```
class MyBoolean(object):

    def __bool__(self):
        return True             # implement some logic here

    def __nonzero__(self):      # Python 2 compatibility
        return type(self).__bool__(self)
```

Division

```
class MyDivisible(object):

    def __truediv__(self, other):
        return self / other     # implement some logic here

    def __div__(self, other):   # Python 2 compatibility
        return type(self).__truediv__(self, other)

    def __itruediv__(self, other):
        return self // other    # implement some logic here

    def __idiv__(self, other):  # Python 2 compatibility
        return type(self).__itruediv__(self, other)
```

Special methods are looked up on the class and not on the instance to reflect the behavior of the Python interpreter.

Writing compatible code with six

six is the canonical compatibility library for supporting Python 2 and 3 in a single codebase. Read its documentation!

A *customized version of six* is bundled with Django as of version 1.4.2. You can import it as `django.utils.six`.

Here are the most common changes required to write compatible code.

String handling

The `basestring` and `unicode` types were removed in Python 3, and the meaning of `str` changed. To test these types, use the following idioms:

```
isinstance(myvalue, six.string_types)       # replacement for basestring
isinstance(myvalue, six.text_type)          # replacement for unicode
isinstance(myvalue, bytes)                  # replacement for str
```

Python ≥ 2.6 provides `bytes` as an alias for `str`, so you don't need `six.binary_type`.

long

The `long` type no longer exists in Python 3. `1L` is a syntax error. Use `six.integer_types` check if a value is an integer or a long:

```
isinstance(myvalue, six.integer_types)      # replacement for (int, long)
```

xrange

If you use `xrange` on Python 2, import `six.moves.range` and use that instead. You can also import `six.moves.xrange` (it's equivalent to `six.moves.range`) but the first technique allows you to simply drop the import when dropping support for Python 2.

Moved modules

Some modules were renamed in Python 3. The `django.utils.six.moves` module (based on the `six.moves` module) provides a compatible location to import them.

PY2

If you need different code in Python 2 and Python 3, check `six.PY2`:

```
if six.PY2:
    # compatibility code for Python 2
```

This is a last resort solution when `six` doesn't provide an appropriate function.

Django customized version of `six`

The version of six bundled with Django (`django.utils.six`) includes a few customizations for internal use only.

3.19 Security in Django

This document is an overview of Django's security features. It includes advice on securing a Django-powered site.

3.19.1 Cross site scripting (XSS) protection

XSS attacks allow a user to inject client side scripts into the browsers of other users. This is usually achieved by storing the malicious scripts in the database where it will be retrieved and displayed to other users, or by getting users to click a link which will cause the attacker's JavaScript to be executed by the user's browser. However, XSS attacks can originate from any untrusted source of data, such as cookies or Web services, whenever the data is not sufficiently sanitized before including in a page.

Using Django templates protects you against the majority of XSS attacks. However, it is important to understand what protections it provides and its limitations.

Django templates *escape specific characters* which are particularly dangerous to HTML. While this protects users from most malicious input, it is not entirely foolproof. For example, it will not protect the following:

```
<style class={{ var }}>...</style>
```

If `var` is set to `'class1 onmouseover=javascript:func()'`, this can result in unauthorized JavaScript execution, depending on how the browser renders imperfect HTML. (Quoting the attribute value would fix this case.)

It is also important to be particularly careful when using `is_safe` with custom template tags, the *`safe`* template tag, *`mark_safe`*, and when autoescape is turned off.

In addition, if you are using the template system to output something other than HTML, there may be entirely separate characters and words which require escaping.

You should also be very careful when storing HTML in the database, especially when that HTML is retrieved and displayed.

3.19.2 Cross site request forgery (CSRF) protection

CSRF attacks allow a malicious user to execute actions using the credentials of another user without that user's knowledge or consent.

Django has built-in protection against most types of CSRF attacks, providing you have *enabled and used it* where appropriate. However, as with any mitigation technique, there are limitations. For example, it is possible to disable the CSRF module globally or for particular views. You should only do this if you know what you are doing. There are other *limitations* if your site has subdomains that are outside of your control.

CSRF protection works by checking for a secret in each POST request. This ensures that a malicious user cannot simply "replay" a form POST to your website and have another logged in user unwittingly submit that form. The malicious user would have to know the secret, which is user specific (using a cookie).

When deployed with *HTTPS*, `CsrfViewMiddleware` will check that the HTTP referer header is set to a URL on the same origin (including subdomain and port). Because HTTPS provides additional security, it is imperative to ensure connections use HTTPS where it is available by forwarding insecure connection requests and using HSTS for supported browsers.

Be very careful with marking views with the `csrf_exempt` decorator unless it is absolutely necessary.

3.19.3 SQL injection protection

SQL injection is a type of attack where a malicious user is able to execute arbitrary SQL code on a database. This can result in records being deleted or data leakage.

By using Django's querysets, the resulting SQL will be properly escaped by the underlying database driver. However, Django also gives developers power to write *raw queries* or execute *custom sql*. These capabilities should be used sparingly and you should always be careful to properly escape any parameters that the user can control. In addition, you should exercise caution when using `extra()` and `RawSQL`.

3.19.4 Clickjacking protection

Clickjacking is a type of attack where a malicious site wraps another site in a frame. This attack can result in an unsuspecting user being tricked into performing unintended actions on the target site.

Django contains *clickjacking protection* in the form of the `X-Frame-Options middleware` which in a supporting browser can prevent a site from being rendered inside a frame. It is possible to disable the protection on a per view basis or to configure the exact header value sent.

The middleware is strongly recommended for any site that does not need to have its pages wrapped in a frame by third party sites, or only needs to allow that for a small section of the site.

3.19.5 SSL/HTTPS

It is always better for security to deploy your site behind HTTPS. Without this, it is possible for malicious network users to sniff authentication credentials or any other information transferred between client and server, and in some cases – **active** network attackers – to alter data that is sent in either direction.

If you want the protection that HTTPS provides, and have enabled it on your server, there are some additional steps you may need:

- If necessary, set `SECURE_PROXY_SSL_HEADER`, ensuring that you have understood the warnings there thoroughly. Failure to do this can result in CSRF vulnerabilities, and failure to do it correctly can also be dangerous!
- Set `SECURE_SSL_REDIRECT` to `True`, so that requests over HTTP are redirected to HTTPS.

  Please note the caveats under `SECURE_PROXY_SSL_HEADER`. For the case of a reverse proxy, it may be easier or more secure to configure the main Web server to do the redirect to HTTPS.
- Use 'secure' cookies.

  If a browser connects initially via HTTP, which is the default for most browsers, it is possible for existing cookies to be leaked. For this reason, you should set your `SESSION_COOKIE_SECURE` and `CSRF_COOKIE_SECURE` settings to `True`. This instructs the browser to only send these cookies over HTTPS connections. Note that this will mean that sessions will not work over HTTP, and the CSRF protection will prevent any POST data being accepted over HTTP (which will be fine if you are redirecting all HTTP traffic to HTTPS).
- Use *HTTP Strict Transport Security* (HSTS)

  HSTS is an HTTP header that informs a browser that all future connections to a particular site should always use HTTPS. Combined with redirecting requests over HTTP to HTTPS, this will ensure that connections always enjoy the added security of SSL provided one successful connection has occurred. HSTS may either be configured with `SECURE_HSTS_SECONDS` and `SECURE_HSTS_INCLUDE_SUBDOMAINS` or on the Web server.

3.19.6 Host header validation

Django uses the `Host` header provided by the client to construct URLs in certain cases. While these values are sanitized to prevent Cross Site Scripting attacks, a fake `Host` value can be used for Cross-Site Request Forgery, cache poisoning attacks, and poisoning links in emails.

Because even seemingly-secure web server configurations are susceptible to fake `Host` headers, Django validates `Host` headers against the *`ALLOWED_HOSTS`* setting in the *`django.http.HttpRequest.get_host()`* method.

This validation only applies via *`get_host()`*; if your code accesses the `Host` header directly from `request.META` you are bypassing this security protection.

For more details see the full *`ALLOWED_HOSTS`* documentation.

: Previous versions of this document recommended configuring your web server to ensure it validates incoming HTTP `Host` headers. While this is still recommended, in many common web servers a configuration that seems to validate the `Host` header may not in fact do so. For instance, even if Apache is configured such that your Django site is served from a non-default virtual host with the `ServerName` set, it is still possible for an HTTP request to match this virtual host and supply a fake `Host` header. Thus, Django now requires that you set *`ALLOWED_HOSTS`* explicitly rather than relying on web server configuration.

Additionally, Django requires you to explicitly enable support for the `X-Forwarded-Host` header (via the *`USE_X_FORWARDED_HOST`* setting) if your configuration requires it.

3.19.7 Session security

Similar to the *CSRF limitations* requiring a site to be deployed such that untrusted users don't have access to any subdomains, *`django.contrib.sessions`* also has limitations. See *the session topic guide section on security* for details.

3.19.8 User-uploaded content

: Consider *serving static files from a cloud service or CDN* to avoid some of these issues.

- If your site accepts file uploads, it is strongly advised that you limit these uploads in your Web server configuration to a reasonable size in order to prevent denial of service (DOS) attacks. In Apache, this can be easily set using the LimitRequestBody directive.
- If you are serving your own static files, be sure that handlers like Apache's `mod_php`, which would execute static files as code, are disabled. You don't want users to be able to execute arbitrary code by uploading and requesting a specially crafted file.
- Django's media upload handling poses some vulnerabilities when that media is served in ways that do not follow security best practices. Specifically, an HTML file can be uploaded as an image if that file contains a valid PNG header followed by malicious HTML. This file will pass verification of the library that Django uses for *`ImageField`* image processing (Pillow). When this file is subsequently displayed to a user, it may be displayed as HTML depending on the type and configuration of your web server.

  No bulletproof technical solution exists at the framework level to safely validate all user uploaded file content, however, there are some other steps you can take to mitigate these attacks:

  1. One class of attacks can be prevented by always serving user uploaded content from a distinct top-level or second-level domain. This prevents any exploit blocked by same-origin policy protections such as cross

site scripting. For example, if your site runs on `example.com`, you would want to serve uploaded content (the *MEDIA_URL* setting) from something like `usercontent-example.com`. It's *not* sufficient to serve content from a subdomain like `usercontent.example.com`.

2. Beyond this, applications may choose to define a whitelist of allowable file extensions for user uploaded files and configure the web server to only serve such files.

3.19.9 Additional security topics

While Django provides good security protection out of the box, it is still important to properly deploy your application and take advantage of the security protection of the Web server, operating system and other components.

- Make sure that your Python code is outside of the Web server's root. This will ensure that your Python code is not accidentally served as plain text (or accidentally executed).
- Take care with any *user uploaded files*.
- Django does not throttle requests to authenticate users. To protect against brute-force attacks against the authentication system, you may consider deploying a Django plugin or Web server module to throttle these requests.
- Keep your *SECRET_KEY* a secret.
- It is a good idea to limit the accessibility of your caching system and database using a firewall.
- Take a look at the Open Web Application Security Project (OWASP) Top 10 list which identifies some common vulnerabilities in web applications. While Django has tools to address some of the issues, other issues must be accounted for in the design of your project.

3.20 Performance and optimization

This document provides an overview of techniques and tools that can help get your Django code running more efficiently - faster, and using fewer system resources.

3.20.1 Introduction

Generally one's first concern is to write code that *works*, whose logic functions as required to produce the expected output. Sometimes, however, this will not be enough to make the code work as *efficiently* as one would like.

In this case, what's needed is something - and in practice, often a collection of things - to improve the code's performance without, or only minimally, affecting its behavior.

3.20.2 General approaches

What are you optimizing *for*?

It's important to have a clear idea what you mean by 'performance'. There is not just one metric of it.

Improved speed might be the most obvious aim for a program, but sometimes other performance improvements might be sought, such as lower memory consumption or fewer demands on the database or network.

Improvements in one area will often bring about improved performance in another, but not always; sometimes one can even be at the expense of another. For example, an improvement in a program's speed might cause it to use more memory. Even worse, it can be self-defeating - if the speed improvement is so memory-hungry that the system starts to run out of memory, you'll have done more harm than good.

There are other trade-offs to bear in mind. Your own time is a valuable resource, more precious than CPU time. Some improvements might be too difficult to be worth implementing, or might affect the portability or maintainability of the code. Not all performance improvements are worth the effort.

So, you need to know what performance improvements you are aiming for, and you also need to know that you have a good reason for aiming in that direction - and for that you need:

Performance benchmarking

It's no good just guessing or assuming where the inefficiencies lie in your code.

Django tools

django-debug-toolbar is a very handy tool that provides insights into what your code is doing and how much time it spends doing it. In particular it can show you all the SQL queries your page is generating, and how long each one has taken.

Third-party panels are also available for the toolbar, that can (for example) report on cache performance and template rendering times.

Third-party services

There are a number of free services that will analyze and report on the performance of your site's pages from the perspective of a remote HTTP client, in effect simulating the experience of an actual user.

These can't report on the internals of your code, but can provide a useful insight into your site's overall performance, including aspects that can't be adequately measured from within Django environment. Examples include:

- Yahoo's Yslow
- Google PageSpeed

There are also several paid-for services that perform a similar analysis, including some that are Django-aware and can integrate with your codebase to profile its performance far more comprehensively.

Get things right from the start

Some work in optimization involves tackling performance shortcomings, but some of the work can simply be built in to what you'd do anyway, as part of the good practices you should adopt even before you start thinking about improving performance.

In this respect Python is an excellent language to work with, because solutions that look elegant and feel right usually are the best performing ones. As with most skills, learning what "looks right" takes practice, but one of the most useful guidelines is:

Work at the appropriate level

Django offers many different ways of approaching things, but just because it's possible to do something in a certain way doesn't mean that it's the most appropriate way to do it. For example, you might find that you could calculate the same thing - the number of items in a collection, perhaps - in a `QuerySet`, in Python, or in a template.

However, it will almost always be faster to do this work at lower rather than higher levels. At higher levels the system has to deal with objects through multiple levels of abstraction and layers of machinery.

That is, the database can typically do things faster than Python can, which can do them faster than the template language can:

```
# QuerySet operation on the database
# fast, because that's what databases are good at
my_bicycles.count()

# counting Python objects
# slower, because it requires a database query anyway, and processing
# of the Python objects
len(my_bicycles)

# Django template filter
# slower still, because it will have to count them in Python anyway,
# and because of template language overheads
{{ my_bicycles|length }}
```

Generally speaking, the most appropriate level for the job is the lowest-level one that it is comfortable to code for.

: The example above is merely illustrative.

Firstly, in a real-life case you need to consider what is happening before and after your count to work out what's an optimal way of doing it *in that particular context*. The database optimization documents describes *a case where counting in the template would be better*.

Secondly, there are other options to consider: in a real-life case, `{{ my_bicycles.count }}`, which invokes the `QuerySet` `count()` method directly from the template, might be the most appropriate choice.

3.20.3 Caching

Often it is expensive (that is, resource-hungry and slow) to compute a value, so there can be huge benefit in saving the value to a quickly accessible cache, ready for the next time it's required.

It's a sufficiently significant and powerful technique that Django includes a comprehensive caching framework, as well as other smaller pieces of caching functionality.

The caching framework

Django's caching framework offers very significant opportunities for performance gains, by saving dynamic content so that it doesn't need to be calculated for each request.

For convenience, Django offers different levels of cache granularity: you can cache the output of specific views, or only the pieces that are difficult to produce, or even an entire site.

Implementing caching should not be regarded as an alternative to improving code that's performing poorly because it has been written badly. It's one of the final steps towards producing well-performing code, not a shortcut.

`cached_property`

It's common to have to call a class instance's method more than once. If that function is expensive, then doing so can be wasteful.

Using the *`cached_property`* decorator saves the value returned by a property; the next time the function is called on that instance, it will return the saved value rather than re-computing it. Note that this only works on methods that take `self` as their only argument and that it changes the method to a property.

Certain Django components also have their own caching functionality; these are discussed below in the sections related to those components.

3.20.4 Understanding laziness

Laziness is a strategy complementary to caching. Caching avoids recomputation by saving results; laziness delays computation until it's actually required.

Laziness allows us to refer to things before they are instantiated, or even before it's possible to instantiate them. This has numerous uses.

For example, *lazy translation* can be used before the target language is even known, because it doesn't take place until the translated string is actually required, such as in a rendered template.

Laziness is also a way to save effort by trying to avoid work in the first place. That is, one aspect of laziness is not doing anything until it has to be done, because it may not turn out to be necessary after all. Laziness can therefore have performance implications, and the more expensive the work concerned, the more there is to gain through laziness.

Python provides a number of tools for lazy evaluation, particularly through the generator and generator expression constructs. It's worth reading up on laziness in Python to discover opportunities for making use of lazy patterns in your code.

Laziness in Django

Django is itself quite lazy. A good example of this can be found in the evaluation of `QuerySets`. *QuerySets are lazy*. Thus a `QuerySet` can be created, passed around and combined with other `QuerySets`, without actually incurring any trips to the database to fetch the items it describes. What gets passed around is the `QuerySet` object, not the collection of items that - eventually - will be required from the database.

On the other hand, *certain operations will force the evaluation of a QuerySet*. Avoiding the premature evaluation of a `QuerySet` can save making an expensive and unnecessary trip to the database.

Django also offers a *`keep_lazy()`* decorator. This allows a function that has been called with a lazy argument to behave lazily itself, only being evaluated when it needs to be. Thus the lazy argument - which could be an expensive one - will not be called upon for evaluation until it's strictly required.

3.20.5 Databases

Database optimization

Django's database layer provides various ways to help developers get the best performance from their databases. The database optimization documentation gathers together links to the relevant documentation and adds various tips that outline the steps to take when attempting to optimize your database usage.

Other database-related tips

Enabling *Persistent connections* can speed up connections to the database accounts for a significant part of the request processing time.

This helps a lot on virtualized hosts with limited network performance, for example.

3.20.6 HTTP performance

Middleware

Django comes with a few helpful pieces of middleware that can help optimize your site's performance. They include:

`ConditionalGetMiddleware`

Adds support for modern browsers to conditionally GET responses based on the `ETag` and `Last-Modified` headers.

`GZipMiddleware`

Compresses responses for all modern browsers, saving bandwidth and transfer time. Note that GZipMiddleware is currently considered a security risk, and is vulnerable to attacks that nullify the protection provided by TLS/SSL. See the warning in *`GZipMiddleware`* for more information.

Sessions

Using cached sessions

Using cached sessions may be a way to increase performance by eliminating the need to load session data from a slower storage source like the database and instead storing frequently used session data in memory.

Static files

Static files, which by definition are not dynamic, make an excellent target for optimization gains.

`CachedStaticFilesStorage`

By taking advantage of web browsers' caching abilities, you can eliminate network hits entirely for a given file after the initial download.

`CachedStaticFilesStorage` appends a content-dependent tag to the filenames of static files to make it safe for browsers to cache them long-term without missing future changes - when a file changes, so will the tag, so browsers will reload the asset automatically.

"Minification"

Several third-party Django tools and packages provide the ability to "minify" HTML, CSS, and JavaScript. They remove unnecessary whitespace, newlines, and comments, and shorten variable names, and thus reduce the size of the documents that your site publishes.

3.20.7 Template performance

Note that:

- using `{% block %}` is faster than using `{% include %}`

- heavily-fragmented templates, assembled from many small pieces, can affect performance

The cached template loader

Enabling the `cached template loader` often improves performance drastically, as it avoids compiling each template every time it needs to be rendered.

3.20.8 Using different versions of available software

It can sometimes be worth checking whether different and better-performing versions of the software that you're using are available.

These techniques are targeted at more advanced users who want to push the boundaries of performance of an already well-optimized Django site.

However, they are not magic solutions to performance problems, and they're unlikely to bring better than marginal gains to sites that don't already do the more basic things the right way.

: It's worth repeating: **reaching for alternatives to software you're already using is never the first answer to performance problems**. When you reach this level of optimization, you need a formal benchmarking solution.

Newer is often - but not always - better

It's fairly rare for a new release of well-maintained software to be less efficient, but the maintainers can't anticipate every possible use-case - so while being aware that newer versions are likely to perform better, don't simply assume that they always will.

This is true of Django itself. Successive releases have offered a number of improvements across the system, but you should still check the real-world performance of your application, because in some cases you may find that changes mean it performs worse rather than better.

Newer versions of Python, and also of Python packages, will often perform better too - but measure, rather than assume.

: Unless you've encountered an unusual performance problem in a particular version, you'll generally find better features, reliability, and security in a new release and that these benefits are far more significant than any performance you might win or lose.

Alternatives to Django's template language

For nearly all cases, Django's built-in template language is perfectly adequate. However, if the bottlenecks in your Django project seem to lie in the template system and you have exhausted other opportunities to remedy this, a third-party alternative may be the answer.

Jinja2 can offer performance improvements, particularly when it comes to speed.

Alternative template systems vary in the extent to which they share Django's templating language.

: *If* you experience performance issues in templates, the first thing to do is to understand exactly why. Using an alternative template system may prove faster, but the same gains may also be available without going to that trouble - for example, expensive processing and logic in your templates could be done more efficiently in your views.

Alternative software implementations

It may be worth checking whether Python software you're using has been provided in a different implementation that can execute the same code faster.

However: most performance problems in well-written Django sites aren't at the Python execution level, but rather in inefficient database querying, caching, and templates. If you're relying on poorly-written Python code, your performance problems are unlikely to be solved by having it execute faster.

Using an alternative implementation may introduce compatibility, deployment, portability, or maintenance issues. It goes without saying that before adopting a non-standard implementation you should ensure it provides sufficient performance gains for your application to outweigh the potential risks.

With these caveats in mind, you should be aware of:

PyPy

PyPy is an implementation of Python in Python itself (the 'standard' Python implementation is in C). PyPy can offer substantial performance gains, typically for heavyweight applications.

A key aim of the PyPy project is compatibility with existing Python APIs and libraries. Django is compatible, but you will need to check the compatibility of other libraries you rely on.

C implementations of Python libraries

Some Python libraries are also implemented in C, and can be much faster. They aim to offer the same APIs. Note that compatibility issues and behavior differences are not unknown (and not always immediately evident).

3.21 Serializing Django objects

Django's serialization framework provides a mechanism for "translating" Django models into other formats. Usually these other formats will be text-based and used for sending Django data over a wire, but it's possible for a serializer to handle any format (text-based or not).

:

If you just want to get some data from your tables into a serialized form, you could use the *dumpdata* management command.

3.21.1 Serializing data

At the highest level, serializing data is a very simple operation:

```
from django.core import serializers
data = serializers.serialize("xml", SomeModel.objects.all())
```

The arguments to the `serialize` function are the format to serialize the data to (see *Serialization formats*) and a *QuerySet* to serialize. (Actually, the second argument can be any iterator that yields Django model instances, but it'll almost always be a QuerySet).

`django.core.serializers.`**`get_serializer`**(*format*)

You can also use a serializer object directly:

```
XMLSerializer = serializers.get_serializer("xml")
xml_serializer = XMLSerializer()
xml_serializer.serialize(queryset)
data = xml_serializer.getvalue()
```

This is useful if you want to serialize data directly to a file-like object (which includes an *HttpResponse*):

```
with open("file.xml", "w") as out:
    xml_serializer.serialize(SomeModel.objects.all(), stream=out)
```

: Calling *get_serializer()* with an unknown *format* will raise a `django.core.serializers.SerializerDoesNotExist` exception.

Subset of fields

If you only want a subset of fields to be serialized, you can specify a `fields` argument to the serializer:

```
from django.core import serializers
data = serializers.serialize('xml', SomeModel.objects.all(), fields=('name','size'))
```

In this example, only the `name` and `size` attributes of each model will be serialized. The primary key is always serialized as the `pk` element in the resulting output; it never appears in the `fields` part.

: Depending on your model, you may find that it is not possible to deserialize a model that only serializes a subset of its fields. If a serialized object doesn't specify all the fields that are required by a model, the deserializer will not be able to save deserialized instances.

Inherited models

If you have a model that is defined using an *abstract base class*, you don't have to do anything special to serialize that model. Just call the serializer on the object (or objects) that you want to serialize, and the output will be a complete representation of the serialized object.

However, if you have a model that uses *multi-table inheritance*, you also need to serialize all of the base classes for the model. This is because only the fields that are locally defined on the model will be serialized. For example, consider the following models:

```
class Place(models.Model):
    name = models.CharField(max_length=50)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField(default=False)
```

If you only serialize the Restaurant model:

```
data = serializers.serialize('xml', Restaurant.objects.all())
```

the fields on the serialized output will only contain the `serves_hot_dogs` attribute. The `name` attribute of the base class will be ignored.

In order to fully serialize your `Restaurant` instances, you will need to serialize the `Place` models as well:

```
all_objects = list(Restaurant.objects.all()) + list(Place.objects.all())
data = serializers.serialize('xml', all_objects)
```

3.21.2 Deserializing data

Deserializing data is also a fairly simple operation:

```
for obj in serializers.deserialize("xml", data):
    do_something_with(obj)
```

As you can see, the `deserialize` function takes the same format argument as `serialize`, a string or stream of data, and returns an iterator.

However, here it gets slightly complicated. The objects returned by the `deserialize` iterator *aren't* simple Django objects. Instead, they are special `DeserializedObject` instances that wrap a created – but unsaved – object and any associated relationship data.

Calling `DeserializedObject.save()` saves the object to the database.

: If the `pk` attribute in the serialized data doesn't exist or is null, a new instance will be saved to the database.

This ensures that deserializing is a non-destructive operation even if the data in your serialized representation doesn't match what's currently in the database. Usually, working with these `DeserializedObject` instances looks something like:

```
for deserialized_object in serializers.deserialize("xml", data):
    if object_should_be_saved(deserialized_object):
        deserialized_object.save()
```

In other words, the usual use is to examine the deserialized objects to make sure that they are "appropriate" for saving before doing so. Of course, if you trust your data source you could just save the object and move on.

The Django object itself can be inspected as `deserialized_object.object`. If fields in the serialized data do not exist on a model, a `DeserializationError` will be raised unless the `ignorenonexistent` argument is passed in as `True`:

```
serializers.deserialize("xml", data, ignorenonexistent=True)
```

3.21.3 Serialization formats

Django supports a number of serialization formats, some of which require you to install third-party Python modules:

Identifier	Information
`xml`	Serializes to and from a simple XML dialect.
`json`	Serializes to and from JSON.
`yaml`	Serializes to YAML (YAML Ain't a Markup Language). This serializer is only available if PyYAML is installed.

XML

The basic XML serialization format is quite simple:

```
<?xml version="1.0" encoding="utf-8"?>
<django-objects version="1.0">
    <object pk="123" model="sessions.session">
        <field type="DateTimeField" name="expire_date">2013-01-16T08:16:59.844560+00:00</field>
        <!-- ... -->
    </object>
</django-objects>
```

The whole collection of objects that is either serialized or de-serialized is represented by a `<django-objects>`-tag which contains multiple `<object>`-elements. Each such object has two attributes: "pk" and "model", the latter being represented by the name of the app ("sessions") and the lowercase name of the model ("session") separated by a dot.

Each field of the object is serialized as a `<field>`-element sporting the fields "type" and "name". The text content of the element represents the value that should be stored.

Foreign keys and other relational fields are treated a little bit differently:

```
<object pk="27" model="auth.permission">
    <!-- ... -->
    <field to="contenttypes.contenttype" name="content_type" rel="ManyToOneRel">9</field>
    <!-- ... -->
</object>
```

In this example we specify that the auth.Permission object with the PK 27 has a foreign key to the contenttypes.ContentType instance with the PK 9.

ManyToMany-relations are exported for the model that binds them. For instance, the auth.User model has such a relation to the auth.Permission model:

```
<object pk="1" model="auth.user">
    <!-- ... -->
    <field to="auth.permission" name="user_permissions" rel="ManyToManyRel">
        <object pk="46"></object>
        <object pk="47"></object>
    </field>
</object>
```

This example links the given user with the permission models with PKs 46 and 47.

Control characters

If the content to be serialized contains control characters that are not accepted in the XML 1.0 standard, the serialization will fail with a `ValueError` exception. Read also the W3C's explanation of HTML, XHTML, XML and Control Codes.

JSON

When staying with the same example data as before it would be serialized as JSON in the following way:

```
[
    {
        "pk": "4b678b301dfd8a4e0dad910de3ae245b",
        "model": "sessions.session",
```

```
        "fields": {
            "expire_date": "2013-01-16T08:16:59.844Z",
            ...
        }
    }
]
```

The formatting here is a bit simpler than with XML. The whole collection is just represented as an array and the objects are represented by JSON objects with three properties: "pk", "model" and "fields". "fields" is again an object containing each field's name and value as property and property-value respectively.

Foreign keys just have the PK of the linked object as property value. ManyToMany-relations are serialized for the model that defines them and are represented as a list of PKs.

Be aware that not all Django output can be passed unmodified to `json`. For example, if you have some custom type in an object to be serialized, you'll have to write a custom `json` encoder for it. Something like this will work:

```
from django.utils.encoding import force_text
from django.core.serializers.json import DjangoJSONEncoder

class LazyEncoder(DjangoJSONEncoder):
    def default(self, obj):
        if isinstance(obj, YourCustomType):
            return force_text(obj)
        return super(LazyEncoder, self).default(obj)
```

Also note that GeoDjango provides a customized GeoJSON serializer.

DjangoJSONEncoder

class `django.core.serializers.json.`**DjangoJSONEncoder**

The JSON serializer uses `DjangoJSONEncoder` for encoding. A subclass of `JSONEncoder`, it handles these additional types:

datetime A string of the form `YYYY-MM-DDTHH:mm:ss.sssZ` or `YYYY-MM-DDTHH:mm:ss.sss+HH:MM` as defined in ECMA-262.

date A string of the form `YYYY-MM-DD` as defined in ECMA-262.

time A string of the form `HH:MM:ss.sss` as defined in ECMA-262.

Decimal, Promise (django.utils.functional.lazy() objects), UUID A string representation of the object.

Support for `Promise` was added.

YAML

YAML serialization looks quite similar to JSON. The object list is serialized as a sequence mappings with the keys "pk", "model" and "fields". Each field is again a mapping with the key being name of the field and the value the value:

```
-   fields: {expire_date: !!timestamp '2013-01-16 08:16:59.844560+00:00'}
    model: sessions.session
    pk: 4b678b301dfd8a4e0dad910de3ae245b
```

Referential fields are again just represented by the PK or sequence of PKs.

3.21.4 Natural keys

The default serialization strategy for foreign keys and many-to-many relations is to serialize the value of the primary key(s) of the objects in the relation. This strategy works well for most objects, but it can cause difficulty in some circumstances.

Consider the case of a list of objects that have a foreign key referencing *ContentType*. If you're going to serialize an object that refers to a content type, then you need to have a way to refer to that content type to begin with. Since `ContentType` objects are automatically created by Django during the database synchronization process, the primary key of a given content type isn't easy to predict; it will depend on how and when *migrate* was executed. This is true for all models which automatically generate objects, notably including *Permission*, *Group*, and *User*.

: You should never include automatically generated objects in a fixture or other serialized data. By chance, the primary keys in the fixture may match those in the database and loading the fixture will have no effect. In the more likely case that they don't match, the fixture loading will fail with an *IntegrityError*.

There is also the matter of convenience. An integer id isn't always the most convenient way to refer to an object; sometimes, a more natural reference would be helpful.

It is for these reasons that Django provides *natural keys*. A natural key is a tuple of values that can be used to uniquely identify an object instance without using the primary key value.

Deserialization of natural keys

Consider the following two models:

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)

    birthdate = models.DateField()

    class Meta:
        unique_together = (('first_name', 'last_name'),)

class Book(models.Model):
    name = models.CharField(max_length=100)
    author = models.ForeignKey(Person, on_delete=models.CASCADE)
```

Ordinarily, serialized data for `Book` would use an integer to refer to the author. For example, in JSON, a Book might be serialized as:

```
...
{
    "pk": 1,
    "model": "store.book",
    "fields": {
        "name": "Mostly Harmless",
        "author": 42
    }
}
...
```

This isn't a particularly natural way to refer to an author. It requires that you know the primary key value for the author; it also requires that this primary key value is stable and predictable.

However, if we add natural key handling to Person, the fixture becomes much more humane. To add natural key handling, you define a default Manager for Person with a `get_by_natural_key()` method. In the case of a Person, a good natural key might be the pair of first and last name:

```
from django.db import models

class PersonManager(models.Manager):
    def get_by_natural_key(self, first_name, last_name):
        return self.get(first_name=first_name, last_name=last_name)

class Person(models.Model):
    objects = PersonManager()

    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)

    birthdate = models.DateField()

    class Meta:
        unique_together = (('first_name', 'last_name'),)
```

Now books can use that natural key to refer to `Person` objects:

```
...
{
    "pk": 1,
    "model": "store.book",
    "fields": {
        "name": "Mostly Harmless",
        "author": ["Douglas", "Adams"]
    }
}
...
```

When you try to load this serialized data, Django will use the `get_by_natural_key()` method to resolve `["Douglas", "Adams"]` into the primary key of an actual `Person` object.

: Whatever fields you use for a natural key must be able to uniquely identify an object. This will usually mean that your model will have a uniqueness clause (either unique=True on a single field, or `unique_together` over multiple fields) for the field or fields in your natural key. However, uniqueness doesn't need to be enforced at the database level. If you are certain that a set of fields will be effectively unique, you can still use those fields as a natural key.

Deserialization of objects with no primary key will always check whether the model's manager has a `get_by_natural_key()` method and if so, use it to populate the deserialized object's primary key.

Serialization of natural keys

So how do you get Django to emit a natural key when serializing an object? Firstly, you need to add another method – this time to the model itself:

```
class Person(models.Model):
    objects = PersonManager()

    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)

    birthdate = models.DateField()
```

```
    def natural_key(self):
        return (self.first_name, self.last_name)

    class Meta:
        unique_together = (('first_name', 'last_name'),)
```

That method should always return a natural key tuple – in this example, `(first name, last name)`. Then, when you call `serializers.serialize()`, you provide `use_natural_foreign_keys=True` or `use_natural_primary_keys=True` arguments:

```
>>> serializers.serialize('json', [book1, book2], indent=2,
...      use_natural_foreign_keys=True, use_natural_primary_keys=True)
```

When `use_natural_foreign_keys=True` is specified, Django will use the `natural_key()` method to serialize any foreign key reference to objects of the type that defines the method.

When `use_natural_primary_keys=True` is specified, Django will not provide the primary key in the serialized data of this object since it can be calculated during deserialization:

```
...
{
    "model": "store.person",
    "fields": {
        "first_name": "Douglas",
        "last_name": "Adams",
        "birth_date": "1952-03-11",
    }
}
...
```

This can be useful when you need to load serialized data into an existing database and you cannot guarantee that the serialized primary key value is not already in use, and do not need to ensure that deserialized objects retain the same primary keys.

If you are using *dumpdata* to generate serialized data, use the *dumpdata --natural-foreign* and *dumpdata --natural-primary* command line flags to generate natural keys.

: You don't need to define both `natural_key()` and `get_by_natural_key()`. If you don't want Django to output natural keys during serialization, but you want to retain the ability to load natural keys, then you can opt to not implement the `natural_key()` method.

Conversely, if (for some strange reason) you want Django to output natural keys during serialization, but *not* be able to load those key values, just don't define the `get_by_natural_key()` method.

Dependencies during serialization

Since natural keys rely on database lookups to resolve references, it is important that the data exists before it is referenced. You can't make a "forward reference" with natural keys – the data you're referencing must exist before you include a natural key reference to that data.

To accommodate this limitation, calls to *dumpdata* that use the *dumpdata --natural-foreign* option will serialize any model with a `natural_key()` method before serializing standard primary key objects.

However, this may not always be enough. If your natural key refers to another object (by using a foreign key or natural key to another object as part of a natural key), then you need to be able to ensure that the objects on which a natural key depends occur in the serialized data before the natural key requires them.

To control this ordering, you can define dependencies on your `natural_key()` methods. You do this by setting a `dependencies` attribute on the `natural_key()` method itself.

For example, let's add a natural key to the `Book` model from the example above:

```
class Book(models.Model):
    name = models.CharField(max_length=100)
    author = models.ForeignKey(Person, on_delete=models.CASCADE)

    def natural_key(self):
        return (self.name,) + self.author.natural_key()
```

The natural key for a `Book` is a combination of its name and its author. This means that `Person` must be serialized before `Book`. To define this dependency, we add one extra line:

```
def natural_key(self):
    return (self.name,) + self.author.natural_key()
natural_key.dependencies = ['example_app.person']
```

This definition ensures that all `Person` objects are serialized before any `Book` objects. In turn, any object referencing `Book` will be serialized after both `Person` and `Book` have been serialized.

3.22 Django settings

A Django settings file contains all the configuration of your Django installation. This document explains how settings work and which settings are available.

3.22.1 The basics

A settings file is just a Python module with module-level variables.

Here are a couple of example settings:

```
ALLOWED_HOSTS = ['www.example.com']
DEBUG = False
DEFAULT_FROM_EMAIL = 'webmaster@example.com'
```

: If you set *DEBUG* to `False`, you also need to properly set the *ALLOWED_HOSTS* setting.

Because a settings file is a Python module, the following apply:

- It doesn't allow for Python syntax errors.
- It can assign settings dynamically using normal Python syntax. For example:

```
    MY_SETTING = [str(i) for i in range(30)]
```

- It can import values from other settings files.

3.22.2 Designating the settings

DJANGO_SETTINGS_MODULE

When you use Django, you have to tell it which settings you're using. Do this by using an environment variable, `DJANGO_SETTINGS_MODULE`.

The value of `DJANGO_SETTINGS_MODULE` should be in Python path syntax, e.g. `mysite.settings`. Note that the settings module should be on the Python import search path.

The `django-admin` utility

When using django-admin, you can either set the environment variable once, or explicitly pass in the settings module each time you run the utility.

Example (Unix Bash shell):

```
export DJANGO_SETTINGS_MODULE=mysite.settings
django-admin runserver
```

Example (Windows shell):

```
set DJANGO_SETTINGS_MODULE=mysite.settings
django-admin runserver
```

Use the `--settings` command-line argument to specify the settings manually:

```
django-admin runserver --settings=mysite.settings
```

On the server (`mod_wsgi`)

In your live server environment, you'll need to tell your WSGI application what settings file to use. Do that with `os.environ`:

```
import os

os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'
```

Read the Django mod_wsgi documentation for more information and other common elements to a Django WSGI application.

3.22.3 Default settings

A Django settings file doesn't have to define any settings if it doesn't need to. Each setting has a sensible default value. These defaults live in the module `django/conf/global_settings.py`.

Here's the algorithm Django uses in compiling settings:

- Load settings from `global_settings.py`.
- Load settings from the specified settings file, overriding the global settings as necessary.

Note that a settings file should *not* import from `global_settings`, because that's redundant.

Seeing which settings you've changed

There's an easy way to view which of your settings deviate from the default settings. The command `python manage.py diffsettings` displays differences between the current settings file and Django's default settings.

For more, see the *`diffsettings`* documentation.

3.22.4 Using settings in Python code

In your Django apps, use settings by importing the object `django.conf.settings`. Example:

```
from django.conf import settings

if settings.DEBUG:
    # Do something
```

Note that `django.conf.settings` isn't a module – it's an object. So importing individual settings is not possible:

```
from django.conf.settings import DEBUG  # This won't work.
```

Also note that your code should *not* import from either `global_settings` or your own settings file. `django.conf.settings` abstracts the concepts of default settings and site-specific settings; it presents a single interface. It also decouples the code that uses settings from the location of your settings.

3.22.5 Altering settings at runtime

You shouldn't alter settings in your applications at runtime. For example, don't do this in a view:

```
from django.conf import settings

settings.DEBUG = True   # Don't do this!
```

The only place you should assign to settings is in a settings file.

3.22.6 Security

Because a settings file contains sensitive information, such as the database password, you should make every attempt to limit access to it. For example, change its file permissions so that only you and your Web server's user can read it. This is especially important in a shared-hosting environment.

3.22.7 Available settings

For a full list of available settings, see the settings reference.

3.22.8 Creating your own settings

There's nothing stopping you from creating your own settings, for your own Django apps. Just follow these guidelines:

- Setting names must be all uppercase.
- Don't reinvent an already-existing setting.

For settings that are sequences, Django itself uses lists, but this is only a convention.

3.22.9 Using settings without setting `DJANGO_SETTINGS_MODULE`

In some cases, you might want to bypass the `DJANGO_SETTINGS_MODULE` environment variable. For example, if you're using the template system by itself, you likely don't want to have to set up an environment variable pointing to a settings module.

In these cases, you can configure Django's settings manually. Do this by calling:

django.conf.settings.**configure**(*default_settings*, ***settings*)

Example:

```
from django.conf import settings

settings.configure(DEBUG=True)
```

Pass `configure()` as many keyword arguments as you'd like, with each keyword argument representing a setting and its value. Each argument name should be all uppercase, with the same name as the settings described above. If a particular setting is not passed to `configure()` and is needed at some later point, Django will use the default setting value.

Configuring Django in this fashion is mostly necessary – and, indeed, recommended – when you're using a piece of the framework inside a larger application.

Consequently, when configured via `settings.configure()`, Django will not make any modifications to the process environment variables (see the documentation of *TIME_ZONE* for why this would normally occur). It's assumed that you're already in full control of your environment in these cases.

Custom default settings

If you'd like default values to come from somewhere other than `django.conf.global_settings`, you can pass in a module or class that provides the default settings as the `default_settings` argument (or as the first positional argument) in the call to `configure()`.

In this example, default settings are taken from `myapp_defaults`, and the *DEBUG* setting is set to `True`, regardless of its value in `myapp_defaults`:

```
from django.conf import settings
from myapp import myapp_defaults

settings.configure(default_settings=myapp_defaults, DEBUG=True)
```

The following example, which uses `myapp_defaults` as a positional argument, is equivalent:

```
settings.configure(myapp_defaults, DEBUG=True)
```

Normally, you will not need to override the defaults in this fashion. The Django defaults are sufficiently tame that you can safely use them. Be aware that if you do pass in a new default module, it entirely *replaces* the Django defaults, so you must specify a value for every possible setting that might be used in that code you are importing. Check in `django.conf.settings.global_settings` for the full list.

Either `configure()` or `DJANGO_SETTINGS_MODULE` is required

If you're not setting the `DJANGO_SETTINGS_MODULE` environment variable, you *must* call `configure()` at some point before using any code that reads settings.

If you don't set `DJANGO_SETTINGS_MODULE` and don't call `configure()`, Django will raise an `ImportError` exception the first time a setting is accessed.

If you set `DJANGO_SETTINGS_MODULE`, access settings values somehow, *then* call `configure()`, Django will raise a `RuntimeError` indicating that settings have already been configured. There is a property just for this purpose:

For example:

```
from django.conf import settings
if not settings.configured:
    settings.configure(myapp_defaults, DEBUG=True)
```

Also, it's an error to call `configure()` more than once, or to call `configure()` after any setting has been accessed.

It boils down to this: Use exactly one of either `configure()` or `DJANGO_SETTINGS_MODULE`. Not both, and not neither.

Calling `django.setup()` is required for "standalone" Django usage

If you're using components of Django "standalone" – for example, writing a Python script which loads some Django templates and renders them, or uses the ORM to fetch some data – there's one more step you'll need in addition to configuring settings.

After you've either set *DJANGO_SETTINGS_MODULE* or called `configure()`, you'll need to call *django.setup()* to load your settings and populate Django's application registry. For example:

```
import django
from django.conf import settings
from myapp import myapp_defaults

settings.configure(default_settings=myapp_defaults, DEBUG=True)
django.setup()

# Now this script or any imported module can use any part of Django it needs.
from myapp import models
```

Note that calling `django.setup()` is only necessary if your code is truly standalone. When invoked by your Web server, or through django-admin, Django will handle this for you.

`django.setup()` may only be called once.

Therefore, avoid putting reusable application logic in standalone scripts so that you have to import from the script elsewhere in your application. If you can't avoid that, put the call to `django.setup()` inside an `if` block:

```
if __name__ == '__main__':
    import django
    django.setup()
```

:

The Settings Reference Contains the complete list of core and contrib app settings.

3.23 Signals

Django includes a "signal dispatcher" which helps allow decoupled applications get notified when actions occur elsewhere in the framework. In a nutshell, signals allow certain *senders* to notify a set of *receivers* that some action has taken place. They're especially useful when many pieces of code may be interested in the same events.

Django provides a set of built-in signals that let user code get notified by Django itself of certain actions. These include some useful notifications:

- *django.db.models.signals.pre_save* & *django.db.models.signals.post_save*

  Sent before or after a model's *save()* method is called.

- *django.db.models.signals.pre_delete* & *django.db.models.signals.post_delete*

  Sent before or after a model's *delete()* method or queryset's *delete()* method is called.

- *django.db.models.signals.m2m_changed*

  Sent when a *ManyToManyField* on a model is changed.

- *django.core.signals.request_started* & *django.core.signals.request_finished*

  Sent when Django starts or finishes an HTTP request.

See the built-in signal documentation for a complete list, and a complete explanation of each signal.

You can also *define and send your own custom signals*; see below.

3.23.1 Listening to signals

To receive a signal, you need to register a *receiver* function that gets called when the signal is sent by using the *Signal.connect()* method:

`Signal.`**`connect`**`(`*receiver*, *sender=None*, *weak=True*, *dispatch_uid=None*`)`

- **receiver** – The callback function which will be connected to this signal. See *Receiver functions* for more information.
- **sender** – Specifies a particular sender to receive signals from. See *Connecting to signals sent by specific senders* for more information.
- **weak** – Django stores signal handlers as weak references by default. Thus, if your receiver is a local function, it may be garbage collected. To prevent this, pass `weak=False` when you call the signal's `connect()` method.
- **dispatch_uid** – A unique identifier for a signal receiver in cases where duplicate signals may be sent. See *Preventing duplicate signals* for more information.

Let's see how this works by registering a signal that gets called after each HTTP request is finished. We'll be connecting to the *request_finished* signal.

Receiver functions

First, we need to define a receiver function. A receiver can be any Python function or method:

```
def my_callback(sender, **kwargs):
    print("Request finished!")
```

Notice that the function takes a `sender` argument, along with wildcard keyword arguments (`**kwargs`); all signal handlers must take these arguments.

We'll look at senders *a bit later*, but right now look at the `**kwargs` argument. All signals send keyword arguments, and may change those keyword arguments at any time. In the case of *request_finished*, it's documented as sending no arguments, which means we might be tempted to write our signal handling as `my_callback(sender)`. This would be wrong – in fact, Django will throw an error if you do so. That's because at any point arguments could get added to the signal and your receiver must be able to handle those new arguments.

Connecting receiver functions

There are two ways you can connect a receiver to a signal. You can take the manual connect route:

```
from django.core.signals import request_finished

request_finished.connect(my_callback)
```

Alternatively, you can use a *receiver()* decorator:

receiver(*signal*)

signal – A signal or a list of signals to connect a function to.

Here's how you connect with the decorator:

```
from django.core.signals import request_finished
from django.dispatch import receiver

@receiver(request_finished)
def my_callback(sender, **kwargs):
    print("Request finished!")
```

Now, our `my_callback` function will be called each time a request finishes.

Where should this code live?

Strictly speaking, signal handling and registration code can live anywhere you like, although it's recommended to avoid the application's root module and its `models` module to minimize side-effects of importing code.

In practice, signal handlers are usually defined in a `signals` submodule of the application they relate to. Signal receivers are connected in the *ready()* method of your application configuration class. If you're using the *receiver()* decorator, simply import the `signals` submodule inside *ready()*.

: The *ready()* method may be executed more than once during testing, so you may want to *guard your signals from duplication*, especially if you're planning to send them within tests.

Connecting to signals sent by specific senders

Some signals get sent many times, but you'll only be interested in receiving a certain subset of those signals. For example, consider the *django.db.models.signals.pre_save* signal sent before a model gets saved. Most of the time, you don't need to know when *any* model gets saved – just when one *specific* model is saved.

In these cases, you can register to receive signals sent only by particular senders. In the case of *django.db.models.signals.pre_save*, the sender will be the model class being saved, so you can indicate that you only want signals sent by some model:

```
from django.db.models.signals import pre_save
from django.dispatch import receiver
from myapp.models import MyModel


@receiver(pre_save, sender=MyModel)
def my_handler(sender, **kwargs):
    ...
```

The `my_handler` function will only be called when an instance of `MyModel` is saved.

Different signals use different objects as their senders; you'll need to consult the built-in signal documentation for details of each particular signal.

Preventing duplicate signals

In some circumstances, the code connecting receivers to signals may run multiple times. This can cause your receiver function to be registered more than once, and thus called multiples times for a single signal event.

If this behavior is problematic (such as when using signals to send an email whenever a model is saved), pass a unique identifier as the `dispatch_uid` argument to identify your receiver function. This identifier will usually be a string, although any hashable object will suffice. The end result is that your receiver function will only be bound to the signal once for each unique `dispatch_uid` value:

```
from django.core.signals import request_finished

request_finished.connect(my_callback, dispatch_uid="my_unique_identifier")
```

3.23.2 Defining and sending signals

Your applications can take advantage of the signal infrastructure and provide its own signals.

Defining signals

class **Signal**(*providing_args=list*)

All signals are *`django.dispatch.Signal`* instances. The `providing_args` is a list of the names of arguments the signal will provide to listeners. This is purely documentational, however, as there is nothing that checks that the signal actually provides these arguments to its listeners.

For example:

```
import django.dispatch

pizza_done = django.dispatch.Signal(providing_args=["toppings", "size"])
```

This declares a `pizza_done` signal that will provide receivers with `toppings` and `size` arguments.

Remember that you're allowed to change this list of arguments at any time, so getting the API right on the first try isn't necessary.

Sending signals

There are two ways to send signals in Django.

`Signal.`**send**(*sender*, ***kwargs*)

`Signal.`**send_robust**(*sender*, ***kwargs*)

To send a signal, call either *`Signal.send()`* (all built-in signals use this) or *`Signal.send_robust()`*. You must provide the `sender` argument (which is a class most of the time) and may provide as many other keyword arguments as you like.

For example, here's how sending our `pizza_done` signal might look:

```
class PizzaStore(object):
    ...

    def send_pizza(self, toppings, size):
        pizza_done.send(sender=self.__class__, toppings=toppings, size=size)
        ...
```

Both `send()` and `send_robust()` return a list of tuple pairs `[(receiver, response), ... ]`, representing the list of called receiver functions and their response values.

`send()` differs from `send_robust()` in how exceptions raised by receiver functions are handled. `send()` does *not* catch any exceptions raised by receivers; it simply allows errors to propagate. Thus not all receivers may be notified of a signal in the face of an error.

`send_robust()` catches all errors derived from Python's `Exception` class, and ensures all receivers are notified of the signal. If an error occurs, the error instance is returned in the tuple pair for the receiver that raised the error.

The tracebacks are present on the `__traceback__` attribute of the errors returned when calling `send_robust()`.

3.23.3 Disconnecting signals

`Signal.`**`disconnect`**(*receiver=None*, *sender=None*, *dispatch_uid=None*)

To disconnect a receiver from a signal, call *`Signal.disconnect()`*. The arguments are as described in *`Signal.connect()`*. The method returns `True` if a receiver was disconnected and `False` if not.

The `receiver` argument indicates the registered receiver to disconnect. It may be `None` if `dispatch_uid` is used to identify the receiver.

1.9 : The `weak` argument is deprecated as it has no effect. It will be removed in Django 2.0.

3.24 System check framework

The system check framework is a set of static checks for validating Django projects. It detects common problems and provides hints for how to fix them. The framework is extensible so you can easily add your own checks.

Checks can be triggered explicitly via the *`check`* command. Checks are triggered implicitly before most commands, including *`runserver`* and *`migrate`*. For performance reasons, checks are not run as part of the WSGI stack that is used in deployment. If you need to run system checks on your deployment server, trigger them explicitly using *`check`*.

Serious errors will prevent Django commands (such as *`runserver`*) from running at all. Minor problems are reported to the console. If you have inspected the cause of a warning and are happy to ignore it, you can hide specific warnings using the *`SILENCED_SYSTEM_CHECKS`* setting in your project settings file.

A full list of all checks that can be raised by Django can be found in the System check reference.

3.24.1 Writing your own checks

The framework is flexible and allows you to write functions that perform any other kind of check you may require. The following is an example stub check function:

```
from django.core.checks import Error, register

@register()
def example_check(app_configs, **kwargs):
    errors = []
    # ... your check logic here
    if check_failed:
        errors.append(
            Error(
                'an error',
                hint='A hint.',
```

```
                obj=checked_object,
                id='myapp.E001',
            )
        )
    return errors
```

The check function *must* accept an `app_configs` argument; this argument is the list of applications that should be inspected. If None, the check must be run on *all* installed apps in the project. The `**kwargs` argument is required for future expansion.

Messages

The function must return a list of messages. If no problems are found as a result of the check, the check function must return an empty list.

The warnings and errors raised by the check method must be instances of *CheckMessage*. An instance of *CheckMessage* encapsulates a single reportable error or warning. It also provides context and hints applicable to the message, and a unique identifier that is used for filtering purposes.

The concept is very similar to messages from the message framework or the logging framework. Messages are tagged with a `level` indicating the severity of the message.

There are also shortcuts to make creating messages with common levels easier. When using these classes you can omit the `level` argument because it is implied by the class name.

- *Debug*
- *Info*
- *Warning*
- *Error*
- *Critical*

Registering and labeling checks

Lastly, your check function must be registered explicitly with system check registry. Checks should be registered in a file that's loaded when your application is loaded; for example, in the *AppConfig.ready()* method.

register(**tags)(function*)

You can pass as many tags to `register` as you want in order to label your check. Tagging checks is useful since it allows you to run only a certain group of checks. For example, to register a compatibility check, you would make the following call:

```
from django.core.checks import register, Tags

@register(Tags.compatibility)
def my_check(app_configs, **kwargs):
    # ... perform compatibility checks and collect errors
    return errors
```

You can register "deployment checks" that are only relevant to a production settings file like this:

```
@register(Tags.security, deploy=True)
def my_check(app_configs, **kwargs):
    ...
```

These checks will only be run if the *check --deploy* option is used.

You can also use `register` as a function rather than a decorator by passing a callable object (usually a function) as the first argument to `register`.

The code below is equivalent to the code above:

```
def my_check(app_configs, **kwargs):
    ...
register(my_check, Tags.security, deploy=True)
```

Field, model, manager, and database checks

In some cases, you won't need to register your check function – you can piggyback on an existing registration.

Fields, models, model managers, and database backends all implement a `check()` method that is already registered with the check framework. If you want to add extra checks, you can extend the implementation on the base class, perform any extra checks you need, and append any messages to those generated by the base class. It's recommended that you delegate each check to separate methods.

Database backend checks were added.

Consider an example where you are implementing a custom field named `RangedIntegerField`. This field adds `min` and `max` arguments to the constructor of `IntegerField`. You may want to add a check to ensure that users provide a min value that is less than or equal to the max value. The following code snippet shows how you can implement this check:

```
from django.core import checks
from django.db import models

class RangedIntegerField(models.IntegerField):
    def __init__(self, min=None, max=None, **kwargs):
        super(RangedIntegerField, self).__init__(**kwargs)
        self.min = min
        self.max = max

    def check(self, **kwargs):
        # Call the superclass
        errors = super(RangedIntegerField, self).check(**kwargs)

        # Do some custom checks and add messages to `errors`:
        errors.extend(self._check_min_max_values(**kwargs))

        # Return all errors and warnings
        return errors

    def _check_min_max_values(self, **kwargs):
        if (self.min is not None and
                self.max is not None and
                self.min > self.max):
            return [
                checks.Error(
                    'min greater than max.',
                    hint='Decrease min or increase max.',
                    obj=self,
                    id='myapp.E001',
                )
            ]
```

```
        # When no error, return an empty list
        return []
```

If you wanted to add checks to a model manager, you would take the same approach on your subclass of *Manager*.

If you want to add a check to a model class, the approach is *almost* the same: the only difference is that the check is a classmethod, not an instance method:

```
class MyModel(models.Model):
    @classmethod
    def check(cls, **kwargs):
        errors = super(MyModel, cls).check(**kwargs)
        # ... your own checks ...
        return errors
```

Writing tests

Messages are comparable. That allows you to easily write tests:

```
from django.core.checks import Error
errors = checked_object.check()
expected_errors = [
    Error(
        'an error',
        hint='A hint.',
        obj=checked_object,
        id='myapp.E001',
    )
]
self.assertEqual(errors, expected_errors)
```

3.25 External packages

Django ships with a variety of extra, optional tools that solve common problems (`contrib.*`). For easier maintenance and to trim the size of the codebase, a few of those applications have been moved out to separate projects.

3.25.1 Localflavor

`django-localflavor` is a collection of utilities for particular countries and cultures.

- GitHub
- Documentation
- PyPI

3.25.2 Comments

`django-contrib-comments` can be used to attach comments to any model, so you can use it for comments on blog entries, photos, book chapters, or anything else. Most users will be better served with a custom solution, or a hosted product like Disqus.

- GitHub

- Documentation
- PyPI

3.25.3 Formtools

`django-formtools` is a collection of assorted utilities to work with forms.

- GitHub
- Documentation
- PyPI

CHAPTER 4

"How-to" guides

Here you'll find short answers to "How do I....?" types of questions. These how-to guides don't cover topics in depth – you'll find that material in the Using Django and the API Reference. However, these guides will help you quickly accomplish common tasks.

4.1 Authentication using `REMOTE_USER`

This document describes how to make use of external authentication sources (where the Web server sets the `REMOTE_USER` environment variable) in your Django applications. This type of authentication solution is typically seen on intranet sites, with single sign-on solutions such as IIS and Integrated Windows Authentication or Apache and mod_authnz_ldap, CAS, Cosign, WebAuth, mod_auth_sspi, etc.

When the Web server takes care of authentication it typically sets the `REMOTE_USER` environment variable for use in the underlying application. In Django, `REMOTE_USER` is made available in the *`request.META`* attribute. Django can be configured to make use of the `REMOTE_USER` value using the `RemoteUserMiddleware` or `PersistentRemoteUserMiddleware`, and *`RemoteUserBackend`* classes found in *`django.contrib.auth`*.

4.1.1 Configuration

First, you must add the *`django.contrib.auth.middleware.RemoteUserMiddleware`* to the *`MIDDLEWARE`* setting **after** the *`django.contrib.auth.middleware.AuthenticationMiddleware`*:

```
MIDDLEWARE = [
    '...',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.auth.middleware.RemoteUserMiddleware',
    '...',
]
```

Next, you must replace the *`ModelBackend`* with *`RemoteUserBackend`* in the *`AUTHENTICATION_BACKENDS`* setting:

```
AUTHENTICATION_BACKENDS = [
    'django.contrib.auth.backends.RemoteUserBackend',
]
```

With this setup, `RemoteUserMiddleware` will detect the username in `request.META['REMOTE_USER']` and will authenticate and auto-login that user using the *`RemoteUserBackend`*.

Be aware that this particular setup disables authentication with the default `ModelBackend`. This means that if the `REMOTE_USER` value is not set then the user is unable to log in, even using Django's admin interface. Adding `'django.contrib.auth.backends.ModelBackend'` to the `AUTHENTICATION_BACKENDS` list will use `ModelBackend` as a fallback if `REMOTE_USER` is absent, which will solve these issues.

Django's user management, such as the views in `contrib.admin` and the *createsuperuser* management command, doesn't integrate with remote users. These interfaces work with users stored in the database regardless of `AUTHENTICATION_BACKENDS`.

: Since the `RemoteUserBackend` inherits from `ModelBackend`, you will still have all of the same permissions checking that is implemented in `ModelBackend`.

Users with *is_active=False* won't be allowed to authenticate. Use *AllowAllUsersRemoteUserBackend* if you want to allow them to.

In older versions, inactive users weren't rejected as described above.

If your authentication mechanism uses a custom HTTP header and not `REMOTE_USER`, you can subclass `RemoteUserMiddleware` and set the `header` attribute to the desired `request.META` key. For example:

```
from django.contrib.auth.middleware import RemoteUserMiddleware

class CustomHeaderMiddleware(RemoteUserMiddleware):
    header = 'HTTP_AUTHUSER'
```

: Be very careful if using a `RemoteUserMiddleware` subclass with a custom HTTP header. You must be sure that your front-end web server always sets or strips that header based on the appropriate authentication checks, never permitting an end-user to submit a fake (or "spoofed") header value. Since the HTTP headers `X-Auth-User` and `X-Auth_User` (for example) both normalize to the `HTTP_X_AUTH_USER` key in `request.META`, you must also check that your web server doesn't allow a spoofed header using underscores in place of dashes.
This warning doesn't apply to `RemoteUserMiddleware` in its default configuration with `header = 'REMOTE_USER'`, since a key that doesn't start with `HTTP_` in `request.META` can only be set by your WSGI server, not directly from an HTTP request header.

If you need more control, you can create your own authentication backend that inherits from *RemoteUserBackend* and override one or more of its attributes and methods.

4.1.2 Using `REMOTE_USER` on login pages only

The `RemoteUserMiddleware` authentication middleware assumes that the HTTP request header `REMOTE_USER` is present with all authenticated requests. That might be expected and practical when Basic HTTP Auth with `htpasswd` or other simple mechanisms are used, but with Negotiate (GSSAPI/Kerberos) or other resource intensive authentication methods, the authentication in the front-end HTTP server is usually only set up for one or a few login URLs, and after successful authentication, the application is supposed to maintain the authenticated session itself.

PersistentRemoteUserMiddleware provides support for this use case. It will maintain the authenticated session until explicit logout by the user. The class can be used as a drop-in replacement of *RemoteUserMiddleware* in the documentation above.

4.2 Writing custom `django-admin` commands

Applications can register their own actions with `manage.py`. For example, you might want to add a `manage.py` action for a Django app that you're distributing. In this document, we will be building a custom `closepoll` command for the `polls` application from the tutorial.

To do this, just add a `management/commands` directory to the application. Django will register a `manage.py` command for each Python module in that directory whose name doesn't begin with an underscore. For example:

```
polls/
    __init__.py
    models.py
    management/
        __init__.py
        commands/
            __init__.py
            _private.py
            closepoll.py
    tests.py
    views.py
```

On Python 2, be sure to include `__init__.py` files in both the `management` and `management/commands` directories as done above or your command will not be detected.

In this example, the `closepoll` command will be made available to any project that includes the `polls` application in *`INSTALLED_APPS`*.

The `_private.py` module will not be available as a management command.

The `closepoll.py` module has only one requirement – it must define a class `Command` that extends *`BaseCommand`* or one of its *subclasses*.

Standalone scripts

Custom management commands are especially useful for running standalone scripts or for scripts that are periodically executed from the UNIX crontab or from Windows scheduled tasks control panel.

To implement the command, edit `polls/management/commands/closepoll.py` to look like this:

```
from django.core.management.base import BaseCommand, CommandError
from polls.models import Question as Poll

class Command(BaseCommand):
    help = 'Closes the specified poll for voting'

    def add_arguments(self, parser):
        parser.add_argument('poll_id', nargs='+', type=int)

    def handle(self, *args, **options):
        for poll_id in options['poll_id']:
            try:
                poll = Poll.objects.get(pk=poll_id)
            except Poll.DoesNotExist:
                raise CommandError('Poll "%s" does not exist' % poll_id)

            poll.opened = False
            poll.save()
```

```
            self.stdout.write(self.style.SUCCESS('Successfully closed poll "%s"' % poll_id))
```

: When you are using management commands and wish to provide console output, you should write to `self.stdout` and `self.stderr`, instead of printing to `stdout` and `stderr` directly. By using these proxies, it becomes much easier to test your custom command. Note also that you don't need to end messages with a newline character, it will be added automatically, unless you specify the `ending` parameter:

```
self.stdout.write("Unterminated line", ending='')
```

The new custom command can be called using `python manage.py closepoll <poll_id>`.

The `handle()` method takes one or more `poll_ids` and sets `poll.opened` to `False` for each one. If the user referenced any nonexistent polls, a *`CommandError`* is raised. The `poll.opened` attribute does not exist in the tutorial and was added to `polls.models.Question` for this example.

4.2.1 Accepting optional arguments

The same `closepoll` could be easily modified to delete a given poll instead of closing it by accepting additional command line options. These custom options can be added in the *`add_arguments()`* method like this:

```
class Command(BaseCommand):
    def add_arguments(self, parser):
        # Positional arguments
        parser.add_argument('poll_id', nargs='+', type=int)

        # Named (optional) arguments
        parser.add_argument(
            '--delete',
            action='store_true',
            dest='delete',
            default=False,
            help='Delete poll instead of closing it',
        )

    def handle(self, *args, **options):
        # ...
        if options['delete']:
            poll.delete()
        # ...
```

The option (`delete` in our example) is available in the options dict parameter of the handle method. See the `argparse` Python documentation for more about `add_argument` usage.

In addition to being able to add custom command line options, all management commands can accept some default options such as *`--verbosity`* and *`--traceback`*.

4.2.2 Management commands and locales

By default, the *`BaseCommand.execute()`* method deactivates translations because some commands shipped with Django perform several tasks (for example, user-facing content rendering and database population) that require a project-neutral string language.

If, for some reason, your custom management command needs to use a fixed locale, you should manually activate and deactivate it in your *`handle()`* method using the functions provided by the I18N support code:

```
from django.core.management.base import BaseCommand, CommandError
from django.utils import translation

class Command(BaseCommand):
    ...
    can_import_settings = True

    def handle(self, *args, **options):

        # Activate a fixed locale, e.g. Russian
        translation.activate('ru')

        # Or you can activate the LANGUAGE_CODE # chosen in the settings:
        from django.conf import settings
        translation.activate(settings.LANGUAGE_CODE)

        # Your command logic here
        ...

        translation.deactivate()
```

Another need might be that your command simply should use the locale set in settings and Django should be kept from deactivating it. You can achieve it by using the *BaseCommand.leave_locale_alone* option.

When working on the scenarios described above though, take into account that system management commands typically have to be very careful about running in non-uniform locales, so you might need to:

- Make sure the *USE_I18N* setting is always `True` when running the command (this is a good example of the potential problems stemming from a dynamic runtime environment that Django commands avoid offhand by deactivating translations).
- Review the code of your command and the code it calls for behavioral differences when locales are changed and evaluate its impact on predictable behavior of your command.

4.2.3 Testing

Information on how to test custom management commands can be found in the *testing docs*.

4.2.4 Command objects

class BaseCommand

The base class from which all management commands ultimately derive.

Use this class if you want access to all of the mechanisms which parse the command-line arguments and work out what code to call in response; if you don't need to change any of that behavior, consider using one of its *subclasses*.

Subclassing the *BaseCommand* class requires that you implement the *handle()* method.

Attributes

All attributes can be set in your derived class and can be used in *BaseCommand*'s *subclasses*.

`BaseCommand.`**can_import_settings**

A boolean indicating whether the command needs to be able to import Django settings; if `True`, `execute()` will verify that this is possible before proceeding. Default value is `True`.

BaseCommand.**help**
: A short description of the command, which will be printed in the help message when the user runs the command `python manage.py help <command>`.

BaseCommand.**missing_args_message**
: If your command defines mandatory positional arguments, you can customize the message error returned in the case of missing arguments. The default is output by `argparse` ("too few arguments").

BaseCommand.**output_transaction**
: A boolean indicating whether the command outputs SQL statements; if `True`, the output will automatically be wrapped with `BEGIN;` and `COMMIT;`. Default value is `False`.

BaseCommand.**requires_migrations_checks**
: A boolean; if `True`, the command prints a warning if the set of migrations on disk don't match the migrations in the database. A warning doesn't prevent the command from executing. Default value is `False`.

BaseCommand.**requires_system_checks**
: A boolean; if `True`, the entire Django project will be checked for potential problems prior to executing the command. Default value is `True`.

BaseCommand.**leave_locale_alone**
: A boolean indicating whether the locale set in settings should be preserved during the execution of the command instead of being forcibly set to 'en-us'.

  Default value is `False`.

  Make sure you know what you are doing if you decide to change the value of this option in your custom command if it creates database content that is locale-sensitive and such content shouldn't contain any translations (like it happens e.g. with django.contrib.auth permissions) as making the locale differ from the de facto default 'en-us' might cause unintended effects. Seethe *Management commands and locales* section above for further details.

  This option can't be `False` when the *can_import_settings* option is set to `False` too because attempting to set the locale needs access to settings. This condition will generate a *CommandError*.

BaseCommand.**style**
: An instance attribute that helps create colored output when writing to `stdout` or `stderr`. For example:

```
self.stdout.write(self.style.SUCCESS('...'))
```

  See *Syntax coloring* to learn how to modify the color palette and to see the available styles (use uppercased versions of the "roles" described in that section).

  If you pass the *--no-color* option when running your command, all `self.style()` calls will return the original string uncolored.

Methods

BaseCommand has a few methods that can be overridden but only the *handle()* method must be implemented.

Implementing a constructor in a subclass

If you implement `__init__` in your subclass of *BaseCommand*, you must call *BaseCommand*'s `__init__`:

```
class Command(BaseCommand):
    def __init__(self, *args, **kwargs):
        super(Command, self).__init__(*args, **kwargs)
        # ...
```

`BaseCommand.`**`add_arguments`**`(`*parser*`)`
Entry point to add parser arguments to handle command line arguments passed to the command. Custom commands should override this method to add both positional and optional arguments accepted by the command. Calling `super()` is not needed when directly subclassing `BaseCommand`.

`BaseCommand.`**`get_version`**`()`
Returns the Django version, which should be correct for all built-in Django commands. User-supplied commands can override this method to return their own version.

`BaseCommand.`**`execute`**`(`**args*, ***options*`)`
Tries to execute this command, performing system checks if needed (as controlled by the *`requires_system_checks`* attribute). If the command raises a *`CommandError`*, it's intercepted and printed to stderr.

Calling a management command in your code

`execute()` should not be called directly from your code to execute a command. Use *`call_command()`* instead.

`BaseCommand.`**`handle`**`(`**args*, ***options*`)`
The actual logic of the command. Subclasses must implement this method.

It may return a Unicode string which will be printed to `stdout` (wrapped by `BEGIN;` and `COMMIT;` if *`output_transaction`* is `True`).

`BaseCommand.`**`check`**`(`*app_configs=None*, *tags=None*, *display_num_errors=False*`)`
Uses the system check framework to inspect the entire Django project for potential problems. Serious problems are raised as a *`CommandError`*; warnings are output to stderr; minor notifications are output to stdout.

If `app_configs` and `tags` are both `None`, all system checks are performed. `tags` can be a list of check tags, like `compatibility` or `models`.

`BaseCommand` subclasses

class `AppCommand`

A management command which takes one or more installed application labels as arguments, and does something with each of them.

Rather than implementing *`handle()`*, subclasses must implement *`handle_app_config()`*, which will be called once for each application.

`AppCommand.`**`handle_app_config`**`(`*app_config*, ***options*`)`
Perform the command's actions for `app_config`, which will be an *`AppConfig`* instance corresponding to an application label given on the command line.

class `LabelCommand`

A management command which takes one or more arbitrary arguments (labels) on the command line, and does something with each of them.

Rather than implementing *`handle()`*, subclasses must implement *`handle_label()`*, which will be called once for each label.

`LabelCommand.`**`handle_label`**`(`*label*, ***options*`)`
Perform the command's actions for `label`, which will be the string as given on the command line.

Command exceptions

exception `CommandError`

Exception class indicating a problem while executing a management command.

If this exception is raised during the execution of a management command from a command line console, it will be caught and turned into a nicely-printed error message to the appropriate output stream (i.e., stderr); as a result, raising this exception (with a sensible description of the error) is the preferred way to indicate that something has gone wrong in the execution of a command.

If a management command is called from code through *`call_command()`*, it's up to you to catch the exception when needed.

4.3 Writing custom model fields

4.3.1 Introduction

The model reference documentation explains how to use Django's standard field classes – *`CharField`*, *`DateField`*, etc. For many purposes, those classes are all you'll need. Sometimes, though, the Django version won't meet your precise requirements, or you'll want to use a field that is entirely different from those shipped with Django.

Django's built-in field types don't cover every possible database column type – only the common types, such as `VARCHAR` and `INTEGER`. For more obscure column types, such as geographic polygons or even user-created types such as PostgreSQL custom types, you can define your own Django `Field` subclasses.

Alternatively, you may have a complex Python object that can somehow be serialized to fit into a standard database column type. This is another case where a `Field` subclass will help you use your object with your models.

Our example object

Creating custom fields requires a bit of attention to detail. To make things easier to follow, we'll use a consistent example throughout this document: wrapping a Python object representing the deal of cards in a hand of Bridge. Don't worry, you don't have to know how to play Bridge to follow this example. You only need to know that 52 cards are dealt out equally to four players, who are traditionally called *north*, *east*, *south* and *west*. Our class looks something like this:

```
class Hand(object):
    """A hand of cards (bridge style)"""

    def __init__(self, north, east, south, west):
        # Input parameters are lists of cards ('Ah', '9s', etc.)
        self.north = north
        self.east = east
        self.south = south
        self.west = west

    # ... (other possibly useful methods omitted) ...
```

This is just an ordinary Python class, with nothing Django-specific about it. We'd like to be able to do things like this in our models (we assume the `hand` attribute on the model is an instance of `Hand`):

```
example = MyModel.objects.get(pk=1)
print(example.hand.north)

new_hand = Hand(north, east, south, west)
```

```
example.hand = new_hand
example.save()
```

We assign to and retrieve from the `hand` attribute in our model just like any other Python class. The trick is to tell Django how to handle saving and loading such an object.

In order to use the `Hand` class in our models, we **do not** have to change this class at all. This is ideal, because it means you can easily write model support for existing classes where you cannot change the source code.

: You might only be wanting to take advantage of custom database column types and deal with the data as standard Python types in your models; strings, or floats, for example. This case is similar to our `Hand` example and we'll note any differences as we go along.

4.3.2 Background theory

Database storage

The simplest way to think of a model field is that it provides a way to take a normal Python object – string, boolean, `datetime`, or something more complex like `Hand` – and convert it to and from a format that is useful when dealing with the database (and serialization, but, as we'll see later, that falls out fairly naturally once you have the database side under control).

Fields in a model must somehow be converted to fit into an existing database column type. Different databases provide different sets of valid column types, but the rule is still the same: those are the only types you have to work with. Anything you want to store in the database must fit into one of those types.

Normally, you're either writing a Django field to match a particular database column type, or there's a fairly straightforward way to convert your data to, say, a string.

For our `Hand` example, we could convert the card data to a string of 104 characters by concatenating all the cards together in a pre-determined order – say, all the *north* cards first, then the *east*, *south* and *west* cards. So `Hand` objects can be saved to text or character columns in the database.

What does a field class do?

All of Django's fields (and when we say *fields* in this document, we always mean model fields and not form fields) are subclasses of *`django.db.models.Field`*. Most of the information that Django records about a field is common to all fields – name, help text, uniqueness and so forth. Storing all that information is handled by `Field`. We'll get into the precise details of what `Field` can do later on; for now, suffice it to say that everything descends from `Field` and then customizes key pieces of the class behavior.

It's important to realize that a Django field class is not what is stored in your model attributes. The model attributes contain normal Python objects. The field classes you define in a model are actually stored in the `Meta` class when the model class is created (the precise details of how this is done are unimportant here). This is because the field classes aren't necessary when you're just creating and modifying attributes. Instead, they provide the machinery for converting between the attribute value and what is stored in the database or sent to the serializer.

Keep this in mind when creating your own custom fields. The Django `Field` subclass you write provides the machinery for converting between your Python instances and the database/serializer values in various ways (there are differences between storing a value and using a value for lookups, for example). If this sounds a bit tricky, don't worry – it will become clearer in the examples below. Just remember that you will often end up creating two classes when you want a custom field:

- The first class is the Python object that your users will manipulate. They will assign it to the model attribute, they will read from it for displaying purposes, things like that. This is the `Hand` class in our example.
- The second class is the `Field` subclass. This is the class that knows how to convert your first class back and forth between its permanent storage form and the Python form.

4.3.3 Writing a field subclass

When planning your *Field* subclass, first give some thought to which existing *Field* class your new field is most similar to. Can you subclass an existing Django field and save yourself some work? If not, you should subclass the *Field* class, from which everything is descended.

Initializing your new field is a matter of separating out any arguments that are specific to your case from the common arguments and passing the latter to the `__init__()` method of *Field* (or your parent class).

In our example, we'll call our field `HandField`. (It's a good idea to call your *Field* subclass `<Something>Field`, so it's easily identifiable as a *Field* subclass.) It doesn't behave like any existing field, so we'll subclass directly from *Field*:

```
from django.db import models

class HandField(models.Field):

    description = "A hand of cards (bridge style)"

    def __init__(self, *args, **kwargs):
        kwargs['max_length'] = 104
        super(HandField, self).__init__(*args, **kwargs)
```

Our `HandField` accepts most of the standard field options (see the list below), but we ensure it has a fixed length, since it only needs to hold 52 card values plus their suits; 104 characters in total.

: Many of Django's model fields accept options that they don't do anything with. For example, you can pass both *editable* and *auto_now* to a *django.db.models.DateField* and it will simply ignore the *editable* parameter (*auto_now* being set implies `editable=False`). No error is raised in this case.

This behavior simplifies the field classes, because they don't need to check for options that aren't necessary. They just pass all the options to the parent class and then don't use them later on. It's up to you whether you want your fields to be more strict about the options they select, or to use the simpler, more permissive behavior of the current fields.

The `Field.__init__()` method takes the following parameters:

- *verbose_name*
- `name`
- *primary_key*
- *max_length*
- *unique*
- *blank*
- *null*
- *db_index*
- `rel`: Used for related fields (like *ForeignKey*). For advanced use only.

- *default*
- *editable*
- serialize: If False, the field will not be serialized when the model is passed to Django's serializers. Defaults to True.
- *unique_for_date*
- *unique_for_month*
- *unique_for_year*
- *choices*
- *help_text*
- *db_column*
- *db_tablespace*: Only for index creation, if the backend supports tablespaces. You can usually ignore this option.
- *auto_created*: True if the field was automatically created, as for the *OneToOneField* used by model inheritance. For advanced use only.

All of the options without an explanation in the above list have the same meaning they do for normal Django fields. See the field documentation for examples and details.

Field deconstruction

The counterpoint to writing your __init__() method is writing the deconstruct() method. This method tells Django how to take an instance of your new field and reduce it to a serialized form - in particular, what arguments to pass to __init__() to re-create it.

If you haven't added any extra options on top of the field you inherited from, then there's no need to write a new deconstruct() method. If, however, you're changing the arguments passed in __init__() (like we are in HandField), you'll need to supplement the values being passed.

The contract of deconstruct() is simple; it returns a tuple of four items: the field's attribute name, the full import path of the field class, the positional arguments (as a list), and the keyword arguments (as a dict). Note this is different from the deconstruct() method *for custom classes* which returns a tuple of three things.

As a custom field author, you don't need to care about the first two values; the base Field class has all the code to work out the field's attribute name and import path. You do, however, have to care about the positional and keyword arguments, as these are likely the things you are changing.

For example, in our HandField class we're always forcibly setting max_length in __init__(). The deconstruct() method on the base Field class will see this and try to return it in the keyword arguments; thus, we can drop it from the keyword arguments for readability:

```
from django.db import models

class HandField(models.Field):

    def __init__(self, *args, **kwargs):
        kwargs['max_length'] = 104
        super(HandField, self).__init__(*args, **kwargs)

    def deconstruct(self):
        name, path, args, kwargs = super(HandField, self).deconstruct()
        del kwargs["max_length"]
        return name, path, args, kwargs
```

If you add a new keyword argument, you need to write code to put its value into `kwargs` yourself:

```
from django.db import models

class CommaSepField(models.Field):
    "Implements comma-separated storage of lists"

    def __init__(self, separator=",", *args, **kwargs):
        self.separator = separator
        super(CommaSepField, self).__init__(*args, **kwargs)

    def deconstruct(self):
        name, path, args, kwargs = super(CommaSepField, self).deconstruct()
        # Only include kwarg if it's not the default
        if self.separator != ",":
            kwargs['separator'] = self.separator
        return name, path, args, kwargs
```

More complex examples are beyond the scope of this document, but remember - for any configuration of your Field instance, `deconstruct()` must return arguments that you can pass to `__init__` to reconstruct that state.

Pay extra attention if you set new default values for arguments in the `Field` superclass; you want to make sure they're always included, rather than disappearing if they take on the old default value.

In addition, try to avoid returning values as positional arguments; where possible, return values as keyword arguments for maximum future compatibility. Of course, if you change the names of things more often than their position in the constructor's argument list, you might prefer positional, but bear in mind that people will be reconstructing your field from the serialized version for quite a while (possibly years), depending how long your migrations live for.

You can see the results of deconstruction by looking in migrations that include the field, and you can test deconstruction in unit tests by just deconstructing and reconstructing the field:

```
name, path, args, kwargs = my_field_instance.deconstruct()
new_instance = MyField(*args, **kwargs)
self.assertEqual(my_field_instance.some_attribute, new_instance.some_attribute)
```

Changing a custom field's base class

You can't change the base class of a custom field because Django won't detect the change and make a migration for it. For example, if you start with:

```
class CustomCharField(models.CharField):
    ...
```

and then decide that you want to use `TextField` instead, you can't change the subclass like this:

```
class CustomCharField(models.TextField):
    ...
```

Instead, you must create a new custom field class and update your models to reference it:

```
class CustomCharField(models.CharField):
    ...

class CustomTextField(models.TextField):
    ...
```

As discussed in *removing fields*, you must retain the original `CustomCharField` class as long as you have migrations that reference it.

Documenting your custom field

As always, you should document your field type, so users will know what it is. In addition to providing a docstring for it, which is useful for developers, you can also allow users of the admin app to see a short description of the field type via the django.contrib.admindocs application. To do this simply provide descriptive text in a *description* class attribute of your custom field. In the above example, the description displayed by the admindocs application for a HandField will be 'A hand of cards (bridge style)'.

In the *django.contrib.admindocs* display, the field description is interpolated with field.__dict__ which allows the description to incorporate arguments of the field. For example, the description for *CharField* is:

```
description = _("String (up to %(max_length)s)")
```

Useful methods

Once you've created your *Field* subclass, you might consider overriding a few standard methods, depending on your field's behavior. The list of methods below is in approximately decreasing order of importance, so start from the top.

Custom database types

Say you've created a PostgreSQL custom type called mytype. You can subclass Field and implement the *db_type()* method, like so:

```
from django.db import models

class MytypeField(models.Field):
    def db_type(self, connection):
        return 'mytype'
```

Once you have MytypeField, you can use it in any model, just like any other Field type:

```
class Person(models.Model):
    name = models.CharField(max_length=80)
    something_else = MytypeField()
```

If you aim to build a database-agnostic application, you should account for differences in database column types. For example, the date/time column type in PostgreSQL is called timestamp, while the same column in MySQL is called datetime. The simplest way to handle this in a *db_type()* method is to check the connection.settings_dict['ENGINE'] attribute.

For example:

```
class MyDateField(models.Field):
    def db_type(self, connection):
        if connection.settings_dict['ENGINE'] == 'django.db.backends.mysql':
            return 'datetime'
        else:
            return 'timestamp'
```

The *db_type()* and *rel_db_type()* methods are called by Django when the framework constructs the CREATE TABLE statements for your application – that is, when you first create your tables. The methods are also called when constructing a WHERE clause that includes the model field – that is, when you retrieve data using QuerySet methods like get(), filter(), and exclude() and have the model field as an argument. They are not called at any other time, so it can afford to execute slightly complex code, such as the connection.settings_dict check in the above example.

Some database column types accept parameters, such as `CHAR(25)`, where the parameter `25` represents the maximum column length. In cases like these, it's more flexible if the parameter is specified in the model rather than being hard-coded in the `db_type()` method. For example, it wouldn't make much sense to have a `CharMaxlength25Field`, shown here:

```
# This is a silly example of hard-coded parameters.
class CharMaxlength25Field(models.Field):
    def db_type(self, connection):
        return 'char(25)'

# In the model:
class MyModel(models.Model):
    # ...
    my_field = CharMaxlength25Field()
```

The better way of doing this would be to make the parameter specifiable at run time – i.e., when the class is instantiated. To do that, just implement `Field.__init__()`, like so:

```
# This is a much more flexible example.
class BetterCharField(models.Field):
    def __init__(self, max_length, *args, **kwargs):
        self.max_length = max_length
        super(BetterCharField, self).__init__(*args, **kwargs)

    def db_type(self, connection):
        return 'char(%s)' % self.max_length

# In the model:
class MyModel(models.Model):
    # ...
    my_field = BetterCharField(25)
```

Finally, if your column requires truly complex SQL setup, return `None` from *`db_type()`*. This will cause Django's SQL creation code to skip over this field. You are then responsible for creating the column in the right table in some other way, of course, but this gives you a way to tell Django to get out of the way.

The *`rel_db_type()`* method is called by fields such as `ForeignKey` and `OneToOneField` that point to another field to determine their database column data types. For example, if you have an `UnsignedAutoField`, you also need the foreign keys that point to that field to use the same data type:

```
# MySQL unsigned integer (range 0 to 4294967295).
class UnsignedAutoField(models.AutoField):
    def db_type(self, connection):
        return 'integer UNSIGNED AUTO_INCREMENT'

    def rel_db_type(self, connection):
        return 'integer UNSIGNED'
```

The *`rel_db_type()`* method was added.

Converting values to Python objects

If your custom *`Field`* class deals with data structures that are more complex than strings, dates, integers, or floats, then you may need to override *`from_db_value()`* and *`to_python()`*.

If present for the field subclass, `from_db_value()` will be called in all circumstances when the data is loaded from the database, including in aggregates and *`values()`* calls.

`to_python()` is called by deserialization and during the *`clean()`* method used from forms.

As a general rule, `to_python()` should deal gracefully with any of the following arguments:

- An instance of the correct type (e.g., `Hand` in our ongoing example).
- A string
- `None` (if the field allows `null=True`)

In our `HandField` class, we're storing the data as a VARCHAR field in the database, so we need to be able to process strings and `None` in the `from_db_value()`. In `to_python()`, we need to also handle `Hand` instances:

```
import re

from django.core.exceptions import ValidationError
from django.db import models
from django.utils.translation import ugettext_lazy as _

def parse_hand(hand_string):
    """Takes a string of cards and splits into a full hand."""
    p1 = re.compile('.{26}')
    p2 = re.compile('..')
    args = [p2.findall(x) for x in p1.findall(hand_string)]
    if len(args) != 4:
        raise ValidationError(_("Invalid input for a Hand instance"))
    return Hand(*args)

class HandField(models.Field):
    # ...

    def from_db_value(self, value, expression, connection, context):
        if value is None:
            return value
        return parse_hand(value)

    def to_python(self, value):
        if isinstance(value, Hand):
            return value

        if value is None:
            return value

        return parse_hand(value)
```

Notice that we always return a `Hand` instance from these methods. That's the Python object type we want to store in the model's attribute.

For `to_python()`, if anything goes wrong during value conversion, you should raise a *`ValidationError`* exception.

Converting Python objects to query values

Since using a database requires conversion in both ways, if you override *`to_python()`* you also have to override *`get_prep_value()`* to convert Python objects back to query values.

For example:

```
class HandField(models.Field):
    # ...

    def get_prep_value(self, value):
```

```
        return ''.join([''.join(l) for l in (value.north,
                value.east, value.south, value.west)])
```

: If your custom field uses the `CHAR`, `VARCHAR` or `TEXT` types for MySQL, you must make sure that *`get_prep_value()`* always returns a string type. MySQL performs flexible and unexpected matching when a query is performed on these types and the provided value is an integer, which can cause queries to include unexpected objects in their results. This problem cannot occur if you always return a string type from *`get_prep_value()`*.

Converting query values to database values

Some data types (for example, dates) need to be in a specific format before they can be used by a database backend. *`get_db_prep_value()`* is the method where those conversions should be made. The specific connection that will be used for the query is passed as the `connection` parameter. This allows you to use backend-specific conversion logic if it is required.

For example, Django uses the following method for its *`BinaryField`*:

```
def get_db_prep_value(self, value, connection, prepared=False):
    value = super(BinaryField, self).get_db_prep_value(value, connection, prepared)
    if value is not None:
        return connection.Database.Binary(value)
    return value
```

In case your custom field needs a special conversion when being saved that is not the same as the conversion used for normal query parameters, you can override *`get_db_prep_save()`*.

Preprocessing values before saving

If you want to preprocess the value just before saving, you can use *`pre_save()`*. For example, Django's *`DateTimeField`* uses this method to set the attribute correctly in the case of *`auto_now`* or *`auto_now_add`*.

If you do override this method, you must return the value of the attribute at the end. You should also update the model's attribute if you make any changes to the value so that code holding references to the model will always see the correct value.

Specifying the form field for a model field

To customize the form field used by *`ModelForm`*, you can override *`formfield()`*.

The form field class can be specified via the `form_class` and `choices_form_class` arguments; the latter is used if the field has choices specified, the former otherwise. If these arguments are not provided, *`CharField`* or *`TypedChoiceField`* will be used.

All of the `kwargs` dictionary is passed directly to the form field's `__init__()` method. Normally, all you need to do is set up a good default for the `form_class` (and maybe `choices_form_class`) argument and then delegate further handling to the parent class. This might require you to write a custom form field (and even a form widget). See the forms documentation for information about this.

Continuing our ongoing example, we can write the *`formfield()`* method as:

```
class HandField(models.Field):
    # ...
```

```
    def formfield(self, **kwargs):
        # This is a fairly standard way to set up some defaults
        # while letting the caller override them.
        defaults = {'form_class': MyFormField}
        defaults.update(kwargs)
        return super(HandField, self).formfield(**defaults)
```

This assumes we've imported a `MyFormField` field class (which has its own default widget). This document doesn't cover the details of writing custom form fields.

Emulating built-in field types

If you have created a *db_type()* method, you don't need to worry about *get_internal_type()* – it won't be used much. Sometimes, though, your database storage is similar in type to some other field, so you can use that other field's logic to create the right column.

For example:

```
class HandField(models.Field):
    # ...

    def get_internal_type(self):
        return 'CharField'
```

No matter which database backend we are using, this will mean that *migrate* and other SQL commands create the right column type for storing a string.

If *get_internal_type()* returns a string that is not known to Django for the database backend you are using – that is, it doesn't appear in `django.db.backends.<db_name>.base.DatabaseWrapper.data_types` – the string will still be used by the serializer, but the default *db_type()* method will return `None`. See the documentation of *db_type()* for reasons why this might be useful. Putting a descriptive string in as the type of the field for the serializer is a useful idea if you're ever going to be using the serializer output in some other place, outside of Django.

Converting field data for serialization

To customize how the values are serialized by a serializer, you can override *value_to_string()*. Using `value_from_object()` is the best way to get the field's value prior to serialization. For example, since our `HandField` uses strings for its data storage anyway, we can reuse some existing conversion code:

```
class HandField(models.Field):
    # ...

    def value_to_string(self, obj):
        value = self.value_from_object(obj)
        return self.get_prep_value(value)
```

Some general advice

Writing a custom field can be a tricky process, particularly if you're doing complex conversions between your Python types and your database and serialization formats. Here are a couple of tips to make things go more smoothly:

1. Look at the existing Django fields (in `django/db/models/fields/__init__.py`) for inspiration. Try to find a field that's similar to what you want and extend it a little bit, instead of creating an entirely new field from scratch.

2. Put a `__str__()` (`__unicode__()` on Python 2) method on the class you're wrapping up as a field. There are a lot of places where the default behavior of the field code is to call *`force_text()`* on the value. (In our examples in this document, `value` would be a `Hand` instance, not a `HandField`). So if your `__str__()` method (`__unicode__()` on Python 2) automatically converts to the string form of your Python object, you can save yourself a lot of work.

4.3.4 Writing a `FileField` subclass

In addition to the above methods, fields that deal with files have a few other special requirements which must be taken into account. The majority of the mechanics provided by `FileField`, such as controlling database storage and retrieval, can remain unchanged, leaving subclasses to deal with the challenge of supporting a particular type of file.

Django provides a `File` class, which is used as a proxy to the file's contents and operations. This can be subclassed to customize how the file is accessed, and what methods are available. It lives at `django.db.models.fields.files`, and its default behavior is explained in the file documentation.

Once a subclass of `File` is created, the new `FileField` subclass must be told to use it. To do so, simply assign the new `File` subclass to the special `attr_class` attribute of the `FileField` subclass.

A few suggestions

In addition to the above details, there are a few guidelines which can greatly improve the efficiency and readability of the field's code.

1. The source for Django's own `ImageField` (in `django/db/models/fields/files.py`) is a great example of how to subclass `FileField` to support a particular type of file, as it incorporates all of the techniques described above.
2. Cache file attributes wherever possible. Since files may be stored in remote storage systems, retrieving them may cost extra time, or even money, that isn't always necessary. Once a file is retrieved to obtain some data about its content, cache as much of that data as possible to reduce the number of times the file must be retrieved on subsequent calls for that information.

4.4 Custom Lookups

Django offers a wide variety of *built-in lookups* for filtering (for example, `exact` and `icontains`). This documentation explains how to write custom lookups and how to alter the working of existing lookups. For the API references of lookups, see the Lookup API reference.

4.4.1 A simple lookup example

Let's start with a simple custom lookup. We will write a custom lookup `ne` which works opposite to `exact`. `Author.objects.filter(name__ne='Jack')` will translate to the SQL:

```
"author"."name" <> 'Jack'
```

This SQL is backend independent, so we don't need to worry about different databases.

There are two steps to making this work. Firstly we need to implement the lookup, then we need to tell Django about it. The implementation is quite straightforward:

```
from django.db.models import Lookup

class NotEqual(Lookup):
    lookup_name = 'ne'

    def as_sql(self, compiler, connection):
        lhs, lhs_params = self.process_lhs(compiler, connection)
        rhs, rhs_params = self.process_rhs(compiler, connection)
        params = lhs_params + rhs_params
        return '%s <> %s' % (lhs, rhs), params
```

To register the `NotEqual` lookup we will just need to call `register_lookup` on the field class we want the lookup to be available. In this case, the lookup makes sense on all `Field` subclasses, so we register it with `Field` directly:

```
from django.db.models.fields import Field
Field.register_lookup(NotEqual)
```

Lookup registration can also be done using a decorator pattern:

```
from django.db.models.fields import Field

@Field.register_lookup
class NotEqualLookup(Lookup):
    # ...
```

We can now use `foo__ne` for any field `foo`. You will need to ensure that this registration happens before you try to create any querysets using it. You could place the implementation in a `models.py` file, or register the lookup in the `ready()` method of an `AppConfig`.

Taking a closer look at the implementation, the first required attribute is `lookup_name`. This allows the ORM to understand how to interpret `name__ne` and use `NotEqual` to generate the SQL. By convention, these names are always lowercase strings containing only letters, but the only hard requirement is that it must not contain the string `__`.

We then need to define the `as_sql` method. This takes a `SQLCompiler` object, called `compiler`, and the active database connection. `SQLCompiler` objects are not documented, but the only thing we need to know about them is that they have a `compile()` method which returns a tuple containing an SQL string, and the parameters to be interpolated into that string. In most cases, you don't need to use it directly and can pass it on to `process_lhs()` and `process_rhs()`.

A `Lookup` works against two values, `lhs` and `rhs`, standing for left-hand side and right-hand side. The left-hand side is usually a field reference, but it can be anything implementing the *query expression API*. The right-hand is the value given by the user. In the example `Author.objects.filter(name__ne='Jack')`, the left-hand side is a reference to the `name` field of the `Author` model, and `'Jack'` is the right-hand side.

We call `process_lhs` and `process_rhs` to convert them into the values we need for SQL using the `compiler` object described before. These methods return tuples containing some SQL and the parameters to be interpolated into that SQL, just as we need to return from our `as_sql` method. In the above example, `process_lhs` returns `('"author"."name"', [])` and `process_rhs` returns `('%s"', ['Jack'])`. In this example there were no parameters for the left hand side, but this would depend on the object we have, so we still need to include them in the parameters we return.

Finally we combine the parts into an SQL expression with `<>`, and supply all the parameters for the query. We then return a tuple containing the generated SQL string and the parameters.

4.4.2 A simple transformer example

The custom lookup above is great, but in some cases you may want to be able to chain lookups together. For example, let's suppose we are building an application where we want to make use of the `abs()` operator. We have an `Experiment` model which records a start value, end value, and the change (start - end). We would like to find all experiments where the change was equal to a certain amount (`Experiment.objects.filter(change__abs=27)`), or where it did not exceed a certain amount (`Experiment.objects.filter(change__abs__lt=27)`).

: This example is somewhat contrived, but it nicely demonstrates the range of functionality which is possible in a database backend independent manner, and without duplicating functionality already in Django.

We will start by writing a `AbsoluteValue` transformer. This will use the SQL function `ABS()` to transform the value before comparison:

```
from django.db.models import Transform

class AbsoluteValue(Transform):
    lookup_name = 'abs'
    function = 'ABS'
```

Next, let's register it for `IntegerField`:

```
from django.db.models import IntegerField
IntegerField.register_lookup(AbsoluteValue)
```

We can now run the queries we had before. `Experiment.objects.filter(change__abs=27)` will generate the following SQL:

```
SELECT ... WHERE ABS("experiments"."change") = 27
```

By using `Transform` instead of `Lookup` it means we are able to chain further lookups afterwards. So `Experiment.objects.filter(change__abs__lt=27)` will generate the following SQL:

```
SELECT ... WHERE ABS("experiments"."change") < 27
```

Note that in case there is no other lookup specified, Django interprets `change__abs=27` as `change__abs__exact=27`.

When looking for which lookups are allowable after the `Transform` has been applied, Django uses the `output_field` attribute. We didn't need to specify this here as it didn't change, but supposing we were applying `AbsoluteValue` to some field which represents a more complex type (for example a point relative to an origin, or a complex number) then we may have wanted to specify that the transform returns a `FloatField` type for further lookups. This can be done by adding an `output_field` attribute to the transform:

```
from django.db.models import FloatField, Transform

class AbsoluteValue(Transform):
    lookup_name = 'abs'
    function = 'ABS'

    @property
    def output_field(self):
        return FloatField()
```

This ensures that further lookups like `abs__lte` behave as they would for a `FloatField`.

4.4.3 Writing an efficient `abs__lt` lookup

When using the above written `abs` lookup, the SQL produced will not use indexes efficiently in some cases. In particular, when we use `change__abs__lt=27`, this is equivalent to `change__gt=-27` AND `change__lt=27`. (For the `lte` case we could use the SQL `BETWEEN`).

So we would like `Experiment.objects.filter(change__abs__lt=27)` to generate the following SQL:

```
SELECT .. WHERE "experiments"."change" < 27 AND "experiments"."change" > -27
```

The implementation is:

```
from django.db.models import Lookup

class AbsoluteValueLessThan(Lookup):
    lookup_name = 'lt'

    def as_sql(self, compiler, connection):
        lhs, lhs_params = compiler.compile(self.lhs.lhs)
        rhs, rhs_params = self.process_rhs(compiler, connection)
        params = lhs_params + rhs_params + lhs_params + rhs_params
        return '%s < %s AND %s > -%s' % (lhs, rhs, lhs, rhs), params

AbsoluteValue.register_lookup(AbsoluteValueLessThan)
```

There are a couple of notable things going on. First, `AbsoluteValueLessThan` isn't calling `process_lhs()`. Instead it skips the transformation of the `lhs` done by `AbsoluteValue` and uses the original `lhs`. That is, we want to get `"experiments"."change"` not `ABS("experiments"."change")`. Referring directly to `self.lhs.lhs` is safe as `AbsoluteValueLessThan` can be accessed only from the `AbsoluteValue` lookup, that is the `lhs` is always an instance of `AbsoluteValue`.

Notice also that as both sides are used multiple times in the query the params need to contain `lhs_params` and `rhs_params` multiple times.

The final query does the inversion (`27` to `-27`) directly in the database. The reason for doing this is that if the `self.rhs` is something else than a plain integer value (for example an `F()` reference) we can't do the transformations in Python.

: In fact, most lookups with `__abs` could be implemented as range queries like this, and on most database backends it is likely to be more sensible to do so as you can make use of the indexes. However with PostgreSQL you may want to add an index on `abs(change)` which would allow these queries to be very efficient.

4.4.4 A bilateral transformer example

The `AbsoluteValue` example we discussed previously is a transformation which applies to the left-hand side of the lookup. There may be some cases where you want the transformation to be applied to both the left-hand side and the right-hand side. For instance, if you want to filter a queryset based on the equality of the left and right-hand side insensitively to some SQL function.

Let's examine the simple example of case-insensitive transformation here. This transformation isn't very useful in practice as Django already comes with a bunch of built-in case-insensitive lookups, but it will be a nice demonstration of bilateral transformations in a database-agnostic way.

We define an `UpperCase` transformer which uses the SQL function `UPPER()` to transform the values before comparison. We define *`bilateral = True`* to indicate that this transformation should apply to both `lhs` and `rhs`:

```
from django.db.models import Transform

class UpperCase(Transform):
    lookup_name = 'upper'
    function = 'UPPER'
    bilateral = True
```

Next, let's register it:

```
from django.db.models import CharField, TextField
CharField.register_lookup(UpperCase)
TextField.register_lookup(UpperCase)
```

Now, the queryset `Author.objects.filter(name__upper="doe")` will generate a case insensitive query like this:

```
SELECT ... WHERE UPPER("author"."name") = UPPER('doe')
```

4.4.5 Writing alternative implementations for existing lookups

Sometimes different database vendors require different SQL for the same operation. For this example we will rewrite a custom implementation for MySQL for the NotEqual operator. Instead of <> we will be using != operator. (Note that in reality almost all databases support both, including all the official databases supported by Django).

We can change the behavior on a specific backend by creating a subclass of `NotEqual` with a `as_mysql` method:

```
class MySQLNotEqual(NotEqual):
    def as_mysql(self, compiler, connection):
        lhs, lhs_params = self.process_lhs(compiler, connection)
        rhs, rhs_params = self.process_rhs(compiler, connection)
        params = lhs_params + rhs_params
        return '%s != %s' % (lhs, rhs), params

Field.register_lookup(MySQLNotEqual)
```

We can then register it with `Field`. It takes the place of the original `NotEqual` class as it has the same `lookup_name`.

When compiling a query, Django first looks for `as_%s % connection.vendor` methods, and then falls back to `as_sql`. The vendor names for the in-built backends are `sqlite`, `postgresql`, `oracle` and `mysql`.

4.4.6 How Django determines the lookups and transforms which are used

In some cases you may wish to dynamically change which `Transform` or `Lookup` is returned based on the name passed in, rather than fixing it. As an example, you could have a field which stores coordinates or an arbitrary dimension, and wish to allow a syntax like `.filter(coords__x7=4)` to return the objects where the 7th coordinate has value 4. In order to do this, you would override `get_lookup` with something like:

```
class CoordinatesField(Field):
    def get_lookup(self, lookup_name):
        if lookup_name.startswith('x'):
            try:
                dimension = int(lookup_name[1:])
            except ValueError:
                pass
            else:
```

```
                return get_coordinate_lookup(dimension)
        return super(CoordinatesField, self).get_lookup(lookup_name)
```

You would then define `get_coordinate_lookup` appropriately to return a `Lookup` subclass which handles the relevant value of `dimension`.

There is a similarly named method called `get_transform()`. `get_lookup()` should always return a `Lookup` subclass, and `get_transform()` a `Transform` subclass. It is important to remember that `Transform` objects can be further filtered on, and `Lookup` objects cannot.

When filtering, if there is only one lookup name remaining to be resolved, we will look for a `Lookup`. If there are multiple names, it will look for a `Transform`. In the situation where there is only one name and a `Lookup` is not found, we look for a `Transform` and then the `exact` lookup on that `Transform`. All call sequences always end with a `Lookup`. To clarify:

- `.filter(myfield__mylookup)` will call `myfield.get_lookup('mylookup')`.
- `.filter(myfield__mytransform__mylookup)` will call `myfield.get_transform('mytransform')`, and then `mytransform.get_lookup('mylookup')`.
- `.filter(myfield__mytransform)` will first call `myfield.get_lookup('mytransform')`, which will fail, so it will fall back to calling `myfield.get_transform('mytransform')` and then `mytransform.get_lookup('exact')`.

4.5 Custom template tags and filters

Django's template language comes with a wide variety of built-in tags and filters designed to address the presentation logic needs of your application. Nevertheless, you may find yourself needing functionality that is not covered by the core set of template primitives. You can extend the template engine by defining custom tags and filters using Python, and then make them available to your templates using the *{% load %}* tag.

4.5.1 Code layout

The most common place to specify custom template tags and filters is inside a Django app. If they relate to an existing app, it makes sense to bundle them there; otherwise, they can be added to a new app. When a Django app is added to *INSTALLED_APPS*, any tags it defines in the conventional location described below are automatically made available to load within templates.

The app should contain a `templatetags` directory, at the same level as `models.py`, `views.py`, etc. If this doesn't already exist, create it - don't forget the `__init__.py` file to ensure the directory is treated as a Python package.

Development server won't automatically restart

After adding the `templatetags` module, you will need to restart your server before you can use the tags or filters in templates.

Your custom tags and filters will live in a module inside the `templatetags` directory. The name of the module file is the name you'll use to load the tags later, so be careful to pick a name that won't clash with custom tags and filters in another app.

For example, if your custom tags/filters are in a file called `poll_extras.py`, your app layout might look like this:

```
polls/
    __init__.py
    models.py
    templatetags/
        __init__.py
        poll_extras.py
    views.py
```

And in your template you would use the following:

```
{% load poll_extras %}
```

The app that contains the custom tags must be in *INSTALLED_APPS* in order for the *{% load %}* tag to work. This is a security feature: It allows you to host Python code for many template libraries on a single host machine without enabling access to all of them for every Django installation.

There's no limit on how many modules you put in the `templatetags` package. Just keep in mind that a *{% load %}* statement will load tags/filters for the given Python module name, not the name of the app.

To be a valid tag library, the module must contain a module-level variable named `register` that is a `template.Library` instance, in which all the tags and filters are registered. So, near the top of your module, put the following:

```
from django import template

register = template.Library()
```

Alternatively, template tag modules can be registered through the `'libraries'` argument to *DjangoTemplates*. This is useful if you want to use a different label from the template tag module name when loading template tags. It also enables you to register tags without installing an application.

Behind the scenes

For a ton of examples, read the source code for Django's default filters and tags. They're in `django/template/defaultfilters.py` and `django/template/defaulttags.py`, respectively.

For more information on the *load* tag, read its documentation.

4.5.2 Writing custom template filters

Custom filters are just Python functions that take one or two arguments:

- The value of the variable (input) – not necessarily a string.
- The value of the argument – this can have a default value, or be left out altogether.

For example, in the filter `{{ var|foo:"bar" }}`, the filter `foo` would be passed the variable `var` and the argument `"bar"`.

Since the template language doesn't provide exception handling, any exception raised from a template filter will be exposed as a server error. Thus, filter functions should avoid raising exceptions if there is a reasonable fallback value to return. In case of input that represents a clear bug in a template, raising an exception may still be better than silent failure which hides the bug.

Here's an example filter definition:

```
def cut(value, arg):
    """Removes all values of arg from the given string"""
    return value.replace(arg, '')
```

And here's an example of how that filter would be used:

```
{{ somevariable|cut:"0" }}
```

Most filters don't take arguments. In this case, just leave the argument out of your function. Example:

```
def lower(value): # Only one argument.
    """Converts a string into all lowercase"""
    return value.lower()
```

Registering custom filters

`django.template.Library.filter()`

Once you've written your filter definition, you need to register it with your `Library` instance, to make it available to Django's template language:

```
register.filter('cut', cut)
register.filter('lower', lower)
```

The `Library.filter()` method takes two arguments:

1. The name of the filter – a string.
2. The compilation function – a Python function (not the name of the function as a string).

You can use `register.filter()` as a decorator instead:

```
@register.filter(name='cut')
def cut(value, arg):
    return value.replace(arg, '')

@register.filter
def lower(value):
    return value.lower()
```

If you leave off the `name` argument, as in the second example above, Django will use the function's name as the filter name.

Finally, `register.filter()` also accepts three keyword arguments, `is_safe`, `needs_autoescape`, and `expects_localtime`. These arguments are described in *filters and auto-escaping* and *filters and time zones* below.

Template filters that expect strings

`django.template.defaultfilters.stringfilter()`

If you're writing a template filter that only expects a string as the first argument, you should use the decorator `stringfilter`. This will convert an object to its string value before being passed to your function:

```
from django import template
from django.template.defaultfilters import stringfilter

register = template.Library()

@register.filter
```

```
@stringfilter
def lower(value):
    return value.lower()
```

This way, you'll be able to pass, say, an integer to this filter, and it won't cause an `AttributeError` (because integers don't have `lower()` methods).

Filters and auto-escaping

When writing a custom filter, give some thought to how the filter will interact with Django's auto-escaping behavior. Note that three types of strings can be passed around inside the template code:

- **Raw strings** are the native Python `str` or `unicode` types. On output, they're escaped if auto-escaping is in effect and presented unchanged, otherwise.
- **Safe strings** are strings that have been marked safe from further escaping at output time. Any necessary escaping has already been done. They're commonly used for output that contains raw HTML that is intended to be interpreted as-is on the client side.

  Internally, these strings are of type `SafeBytes` or `SafeText`. They share a common base class of `SafeData`, so you can test for them using code like:

```
    if isinstance(value, SafeData):
        # Do something with the "safe" string.
        ...
```

Template filter code falls into one of two situations:

1. Your filter does not introduce any HTML-unsafe characters (<, >, ', " or &) into the result that were not already present. In this case, you can let Django take care of all the auto-escaping handling for you. All you need to do is set the `is_safe` flag to `True` when you register your filter function, like so:

```
    @register.filter(is_safe=True)
    def myfilter(value):
        return value
```

   This flag tells Django that if a "safe" string is passed into your filter, the result will still be "safe" and if a non-safe string is passed in, Django will automatically escape it, if necessary.

   You can think of this as meaning "this filter is safe – it doesn't introduce any possibility of unsafe HTML."

   The reason `is_safe` is necessary is because there are plenty of normal string operations that will turn a `SafeData` object back into a normal `str` or `unicode` object and, rather than try to catch them all, which would be very difficult, Django repairs the damage after the filter has completed.

   For example, suppose you have a filter that adds the string `xx` to the end of any input. Since this introduces no dangerous HTML characters to the result (aside from any that were already present), you should mark your filter with `is_safe`:

```
    @register.filter(is_safe=True)
    def add_xx(value):
        return '%sxx' % value
```

   When this filter is used in a template where auto-escaping is enabled, Django will escape the output whenever the input is not already marked as "safe".

   By default, `is_safe` is `False`, and you can omit it from any filters where it isn't required.

   Be careful when deciding if your filter really does leave safe strings as safe. If you're *removing* characters, you might inadvertently leave unbalanced HTML tags or entities in the result. For example, removing a > from the

input might turn `<a>` into `<a`, which would need to be escaped on output to avoid causing problems. Similarly, removing a semicolon (`;`) can turn `&` into `&amp`, which is no longer a valid entity and thus needs further escaping. Most cases won't be nearly this tricky, but keep an eye out for any problems like that when reviewing your code.

Marking a filter `is_safe` will coerce the filter's return value to a string. If your filter should return a boolean or other non-string value, marking it `is_safe` will probably have unintended consequences (such as converting a boolean False to the string 'False').

2. Alternatively, your filter code can manually take care of any necessary escaping. This is necessary when you're introducing new HTML markup into the result. You want to mark the output as safe from further escaping so that your HTML markup isn't escaped further, so you'll need to handle the input yourself.

   To mark the output as a safe string, use *`django.utils.safestring.mark_safe()`*.

   Be careful, though. You need to do more than just mark the output as safe. You need to ensure it really *is* safe, and what you do depends on whether auto-escaping is in effect. The idea is to write filters that can operate in templates where auto-escaping is either on or off in order to make things easier for your template authors.

   In order for your filter to know the current auto-escaping state, set the `needs_autoescape` flag to `True` when you register your filter function. (If you don't specify this flag, it defaults to `False`). This flag tells Django that your filter function wants to be passed an extra keyword argument, called `autoescape`, that is `True` if auto-escaping is in effect and `False` otherwise. It is recommended to set the default of the `autoescape` parameter to `True`, so that if you call the function from Python code it will have escaping enabled by default.

   For example, let's write a filter that emphasizes the first character of a string:

```
from django import template
from django.utils.html import conditional_escape
from django.utils.safestring import mark_safe

register = template.Library()

@register.filter(needs_autoescape=True)
def initial_letter_filter(text, autoescape=True):
    first, other = text[0], text[1:]
    if autoescape:
        esc = conditional_escape
    else:
        esc = lambda x: x
    result = '<strong>%s</strong>%s' % (esc(first), esc(other))
    return mark_safe(result)
```

   The `needs_autoescape` flag and the `autoescape` keyword argument mean that our function will know whether automatic escaping is in effect when the filter is called. We use `autoescape` to decide whether the input data needs to be passed through `django.utils.html.conditional_escape` or not. (In the latter case, we just use the identity function as the "escape" function.) The `conditional_escape()` function is like `escape()` except it only escapes input that is **not** a `SafeData` instance. If a `SafeData` instance is passed to `conditional_escape()`, the data is returned unchanged.

   Finally, in the above example, we remember to mark the result as safe so that our HTML is inserted directly into the template without further escaping.

   There's no need to worry about the `is_safe` flag in this case (although including it wouldn't hurt anything). Whenever you manually handle the auto-escaping issues and return a safe string, the `is_safe` flag won't change anything either way.

: Avoiding XSS vulnerabilities when reusing built-in filters
Django's built-in filters have `autoescape=True` by default in order to get the proper autoescaping behavior and avoid a cross-site script vulnerability.
In older versions of Django, be careful when reusing Django's built-in filters as `autoescape` defaults to `None`. You'll need to pass `autoescape=True` to get autoescaping.
For example, if you wanted to write a custom filter called `urlize_and_linebreaks` that combined the *`urlize`* and *`linebreaksbr`* filters, the filter would look like:

```
from django.template.defaultfilters import linebreaksbr, urlize

@register.filter(needs_autoescape=True)
def urlize_and_linebreaks(text, autoescape=True):
    return linebreaksbr(
        urlize(text, autoescape=autoescape),
        autoescape=autoescape
    )
```

Then:

```
{{ comment|urlize_and_linebreaks }}
```

would be equivalent to:

```
{{ comment|urlize|linebreaksbr }}
```

Filters and time zones

If you write a custom filter that operates on `datetime` objects, you'll usually register it with the `expects_localtime` flag set to `True`:

```
@register.filter(expects_localtime=True)
def businesshours(value):
    try:
        return 9 <= value.hour < 17
    except AttributeError:
        return ''
```

When this flag is set, if the first argument to your filter is a time zone aware datetime, Django will convert it to the current time zone before passing it to your filter when appropriate, according to *rules for time zones conversions in templates*.

4.5.3 Writing custom template tags

Tags are more complex than filters, because tags can do anything. Django provides a number of shortcuts that make writing most types of tags easier. First we'll explore those shortcuts, then explain how to write a tag from scratch for those cases when the shortcuts aren't powerful enough.

Simple tags

`django.template.Library.`**`simple_tag`**`()`

Many template tags take a number of arguments – strings or template variables – and return a result after doing some processing based solely on the input arguments and some external information. For example, a `current_time` tag might accept a format string and return the time as a string formatted accordingly.

To ease the creation of these types of tags, Django provides a helper function, `simple_tag`. This function, which is a method of `django.template.Library`, takes a function that accepts any number of arguments, wraps it in a `render` function and the other necessary bits mentioned above and registers it with the template system.

Our `current_time` function could thus be written like this:

```
import datetime
from django import template

register = template.Library()

@register.simple_tag
def current_time(format_string):
    return datetime.datetime.now().strftime(format_string)
```

A few things to note about the `simple_tag` helper function:

- Checking for the required number of arguments, etc., has already been done by the time our function is called, so we don't need to do that.
- The quotes around the argument (if any) have already been stripped away, so we just receive a plain string.
- If the argument was a template variable, our function is passed the current value of the variable, not the variable itself.

Unlike other tag utilities, `simple_tag` passes its output through *`conditional_escape()`* if the template context is in autoescape mode, to ensure correct HTML and protect you from XSS vulnerabilities.

If additional escaping is not desired, you will need to use *`mark_safe()`* if you are absolutely sure that your code does not contain XSS vulnerabilities. For building small HTML snippets, use of *`format_html()`* instead of `mark_safe()` is strongly recommended.

Auto-escaping for `simple_tag` as described in the previous two paragraphs was added.

If your template tag needs to access the current context, you can use the `takes_context` argument when registering your tag:

```
@register.simple_tag(takes_context=True)
def current_time(context, format_string):
    timezone = context['timezone']
    return your_get_current_time_method(timezone, format_string)
```

Note that the first argument *must* be called `context`.

For more information on how the `takes_context` option works, see the section on *inclusion tags*.

If you need to rename your tag, you can provide a custom name for it:

```
register.simple_tag(lambda x: x - 1, name='minusone')

@register.simple_tag(name='minustwo')
def some_function(value):
    return value - 2
```

`simple_tag` functions may accept any number of positional or keyword arguments. For example:

```
@register.simple_tag
def my_tag(a, b, *args, **kwargs):
    warning = kwargs['warning']
```

```
    profile = kwargs['profile']
    ...
    return ...
```

Then in the template any number of arguments, separated by spaces, may be passed to the template tag. Like in Python, the values for keyword arguments are set using the equal sign ("=") and must be provided after the positional arguments. For example:

```
{% my_tag 123 "abcd" book.title warning=message|lower profile=user.profile %}
```

It's possible to store the tag results in a template variable rather than directly outputting it. This is done by using the `as` argument followed by the variable name. Doing so enables you to output the content yourself where you see fit:

```
{% get_current_time "%Y-%m-%d %I:%M %p" as the_time %}
<p>The time is {{ the_time }}.</p>
```

Inclusion tags

`django.template.Library.`**`inclusion_tag`**`()`

Another common type of template tag is the type that displays some data by rendering *another* template. For example, Django's admin interface uses custom template tags to display the buttons along the bottom of the "add/change" form pages. Those buttons always look the same, but the link targets change depending on the object being edited – so they're a perfect case for using a small template that is filled with details from the current object. (In the admin's case, this is the `submit_row` tag.)

These sorts of tags are called "inclusion tags".

Writing inclusion tags is probably best demonstrated by example. Let's write a tag that outputs a list of choices for a given `Poll` object, such as was created in the *tutorials*. We'll use the tag like this:

```
{% show_results poll %}
```

...and the output will be something like this:

```
<ul>
  <li>First choice</li>
  <li>Second choice</li>
  <li>Third choice</li>
</ul>
```

First, define the function that takes the argument and produces a dictionary of data for the result. The important point here is we only need to return a dictionary, not anything more complex. This will be used as a template context for the template fragment. Example:

```
def show_results(poll):
    choices = poll.choice_set.all()
    return {'choices': choices}
```

Next, create the template used to render the tag's output. This template is a fixed feature of the tag: the tag writer specifies it, not the template designer. Following our example, the template is very simple:

```
<ul>
{% for choice in choices %}
    <li> {{ choice }} </li>
{% endfor %}
</ul>
```

Now, create and register the inclusion tag by calling the `inclusion_tag()` method on a `Library` object. Following our example, if the above template is in a file called `results.html` in a directory that's searched by the template loader, we'd register the tag like this:

```
# Here, register is a django.template.Library instance, as before
@register.inclusion_tag('results.html')
def show_results(poll):
    ...
```

Alternatively it is possible to register the inclusion tag using a *`django.template.Template`* instance:

```
from django.template.loader import get_template
t = get_template('results.html')
register.inclusion_tag(t)(show_results)
```

...when first creating the function.

Sometimes, your inclusion tags might require a large number of arguments, making it a pain for template authors to pass in all the arguments and remember their order. To solve this, Django provides a `takes_context` option for inclusion tags. If you specify `takes_context` in creating a template tag, the tag will have no required arguments, and the underlying Python function will have one argument – the template context as of when the tag was called.

For example, say you're writing an inclusion tag that will always be used in a context that contains `home_link` and `home_title` variables that point back to the main page. Here's what the Python function would look like:

```
@register.inclusion_tag('link.html', takes_context=True)
def jump_link(context):
    return {
        'link': context['home_link'],
        'title': context['home_title'],
    }
```

Note that the first parameter to the function *must* be called `context`.

In that `register.inclusion_tag()` line, we specified `takes_context=True` and the name of the template. Here's what the template `link.html` might look like:

```
Jump directly to <a href="{{ link }}">{{ title }}</a>.
```

Then, any time you want to use that custom tag, load its library and call it without any arguments, like so:

```
{% jump_link %}
```

Note that when you're using `takes_context=True`, there's no need to pass arguments to the template tag. It automatically gets access to the context.

The `takes_context` parameter defaults to `False`. When it's set to `True`, the tag is passed the context object, as in this example. That's the only difference between this case and the previous `inclusion_tag` example.

`inclusion_tag` functions may accept any number of positional or keyword arguments. For example:

```
@register.inclusion_tag('my_template.html')
def my_tag(a, b, *args, **kwargs):
    warning = kwargs['warning']
    profile = kwargs['profile']
    ...
    return ...
```

Then in the template any number of arguments, separated by spaces, may be passed to the template tag. Like in Python, the values for keyword arguments are set using the equal sign ("=") and must be provided after the positional arguments. For example:

```
{% my_tag 123 "abcd" book.title warning=message|lower profile=user.profile %}
```

Assignment tags

`django.template.Library.`**`assignment_tag`**`()`

1.9 : `simple_tag` can now store results in a template variable and should be used instead.

To ease the creation of tags setting a variable in the context, Django provides a helper function, `assignment_tag`. This function works the same way as *`simple_tag()`* except that it stores the tag's result in a specified context variable instead of directly outputting it.

Our earlier `current_time` function could thus be written like this:

```
@register.assignment_tag
def get_current_time(format_string):
    return datetime.datetime.now().strftime(format_string)
```

You may then store the result in a template variable using the `as` argument followed by the variable name, and output it yourself where you see fit:

```
{% get_current_time "%Y-%m-%d %I:%M %p" as the_time %}
<p>The time is {{ the_time }}.</p>
```

Advanced custom template tags

Sometimes the basic features for custom template tag creation aren't enough. Don't worry, Django gives you complete access to the internals required to build a template tag from the ground up.

A quick overview

The template system works in a two-step process: compiling and rendering. To define a custom template tag, you specify how the compilation works and how the rendering works.

When Django compiles a template, it splits the raw template text into ''nodes''. Each node is an instance of `django.template.Node` and has a `render()` method. A compiled template is, simply, a list of `Node` objects. When you call `render()` on a compiled template object, the template calls `render()` on each `Node` in its node list, with the given context. The results are all concatenated together to form the output of the template.

Thus, to define a custom template tag, you specify how the raw template tag is converted into a `Node` (the compilation function), and what the node's `render()` method does.

Writing the compilation function

For each template tag the template parser encounters, it calls a Python function with the tag contents and the parser object itself. This function is responsible for returning a `Node` instance based on the contents of the tag.

For example, let's write a full implementation of our simple template tag, `{% current_time %}`, that displays the current date/time, formatted according to a parameter given in the tag, in `strftime()` syntax. It's a good idea to decide the tag syntax before anything else. In our case, let's say the tag should be used like this:

```
<p>The time is {% current_time "%Y-%m-%d %I:%M %p" %}.</p>
```

The parser for this function should grab the parameter and create a `Node` object:

```
from django import template

def do_current_time(parser, token):
    try:
        # split_contents() knows not to split quoted strings.
        tag_name, format_string = token.split_contents()
    except ValueError:
        raise template.TemplateSyntaxError(
            "%r tag requires a single argument" % token.contents.split()[0]
        )
    if not (format_string[0] == format_string[-1] and format_string[0] in ('"', "'")):
        raise template.TemplateSyntaxError(
            "%r tag's argument should be in quotes" % tag_name
        )
    return CurrentTimeNode(format_string[1:-1])
```

Notes:

- `parser` is the template parser object. We don't need it in this example.
- `token.contents` is a string of the raw contents of the tag. In our example, it's `'current_time "%Y-%m-%d %I:%M %p"'`.
- The `token.split_contents()` method separates the arguments on spaces while keeping quoted strings together. The more straightforward `token.contents.split()` wouldn't be as robust, as it would naively split on *all* spaces, including those within quoted strings. It's a good idea to always use `token.split_contents()`.
- This function is responsible for raising `django.template.TemplateSyntaxError`, with helpful messages, for any syntax error.
- The `TemplateSyntaxError` exceptions use the `tag_name` variable. Don't hard-code the tag's name in your error messages, because that couples the tag's name to your function. `token.contents.split()[0]` will ''always'' be the name of your tag – even when the tag has no arguments.
- The function returns a `CurrentTimeNode` with everything the node needs to know about this tag. In this case, it just passes the argument – `"%Y-%m-%d %I:%M %p"`. The leading and trailing quotes from the template tag are removed in `format_string[1:-1]`.
- The parsing is very low-level. The Django developers have experimented with writing small frameworks on top of this parsing system, using techniques such as EBNF grammars, but those experiments made the template engine too slow. It's low-level because that's fastest.

Writing the renderer

The second step in writing custom tags is to define a `Node` subclass that has a `render()` method.

Continuing the above example, we need to define `CurrentTimeNode`:

```
import datetime
from django import template

class CurrentTimeNode(template.Node):
    def __init__(self, format_string):
        self.format_string = format_string

    def render(self, context):
        return datetime.datetime.now().strftime(self.format_string)
```

Notes:

- `__init__()` gets the `format_string` from `do_current_time()`. Always pass any options/parameters/arguments to a `Node` via its `__init__()`.
- The `render()` method is where the work actually happens.
- `render()` should generally fail silently, particularly in a production environment. In some cases however, particularly if `context.template.engine.debug` is `True`, this method may raise an exception to make debugging easier. For example, several core tags raise `django.template.TemplateSyntaxError` if they receive the wrong number or type of arguments.

Ultimately, this decoupling of compilation and rendering results in an efficient template system, because a template can render multiple contexts without having to be parsed multiple times.

Auto-escaping considerations

The output from template tags is **not** automatically run through the auto-escaping filters (with the exception of *`simple_tag()`* as described above). However, there are still a couple of things you should keep in mind when writing a template tag.

If the `render()` function of your template stores the result in a context variable (rather than returning the result in a string), it should take care to call `mark_safe()` if appropriate. When the variable is ultimately rendered, it will be affected by the auto-escape setting in effect at the time, so content that should be safe from further escaping needs to be marked as such.

Also, if your template tag creates a new context for performing some sub-rendering, set the auto-escape attribute to the current context's value. The `__init__` method for the `Context` class takes a parameter called `autoescape` that you can use for this purpose. For example:

```
from django.template import Context

def render(self, context):
    # ...
    new_context = Context({'var': obj}, autoescape=context.autoescape)
    # ... Do something with new_context ...
```

This is not a very common situation, but it's useful if you're rendering a template yourself. For example:

```
def render(self, context):
    t = context.template.engine.get_template('small_fragment.html')
    return t.render(Context({'var': obj}, autoescape=context.autoescape))
```

If we had neglected to pass in the current `context.autoescape` value to our new `Context` in this example, the results would have *always* been automatically escaped, which may not be the desired behavior if the template tag is used inside a *`{% autoescape off %}`* block.

Thread-safety considerations

Once a node is parsed, its `render` method may be called any number of times. Since Django is sometimes run in multi-threaded environments, a single node may be simultaneously rendering with different contexts in response to two separate requests. Therefore, it's important to make sure your template tags are thread safe.

To make sure your template tags are thread safe, you should never store state information on the node itself. For example, Django provides a builtin *`cycle`* template tag that cycles among a list of given strings each time it's rendered:

```
{% for o in some_list %}
    <tr class="{% cycle 'row1' 'row2' %}">
        ...
    </tr>
{% endfor %}
```

A naive implementation of `CycleNode` might look something like this:

```
import itertools
from django import template

class CycleNode(template.Node):
    def __init__(self, cyclevars):
        self.cycle_iter = itertools.cycle(cyclevars)

    def render(self, context):
        return next(self.cycle_iter)
```

But, suppose we have two templates rendering the template snippet from above at the same time:

1. Thread 1 performs its first loop iteration, `CycleNode.render()` returns 'row1'
2. Thread 2 performs its first loop iteration, `CycleNode.render()` returns 'row2'
3. Thread 1 performs its second loop iteration, `CycleNode.render()` returns 'row1'
4. Thread 2 performs its second loop iteration, `CycleNode.render()` returns 'row2'

The CycleNode is iterating, but it's iterating globally. As far as Thread 1 and Thread 2 are concerned, it's always returning the same value. This is obviously not what we want!

To address this problem, Django provides a `render_context` that's associated with the `context` of the template that is currently being rendered. The `render_context` behaves like a Python dictionary, and should be used to store `Node` state between invocations of the `render` method.

Let's refactor our `CycleNode` implementation to use the `render_context`:

```
class CycleNode(template.Node):
    def __init__(self, cyclevars):
        self.cyclevars = cyclevars

    def render(self, context):
        if self not in context.render_context:
            context.render_context[self] = itertools.cycle(self.cyclevars)
        cycle_iter = context.render_context[self]
        return next(cycle_iter)
```

Note that it's perfectly safe to store global information that will not change throughout the life of the `Node` as an attribute. In the case of `CycleNode`, the `cyclevars` argument doesn't change after the `Node` is instantiated, so we don't need to put it in the `render_context`. But state information that is specific to the template that is currently being rendered, like the current iteration of the `CycleNode`, should be stored in the `render_context`.

: Notice how we used `self` to scope the `CycleNode` specific information within the `render_context`. There may be multiple `CycleNodes` in a given template, so we need to be careful not to clobber another node's state information. The easiest way to do this is to always use `self` as the key into `render_context`. If you're keeping track of several state variables, make `render_context[self]` a dictionary.

Registering the tag

Finally, register the tag with your module's `Library` instance, as explained in *writing custom template filters* above. Example:

```
register.tag('current_time', do_current_time)
```

The `tag()` method takes two arguments:

1. The name of the template tag – a string. If this is left out, the name of the compilation function will be used.
2. The compilation function – a Python function (not the name of the function as a string).

As with filter registration, it is also possible to use this as a decorator:

```
@register.tag(name="current_time")
def do_current_time(parser, token):
    ...

@register.tag
def shout(parser, token):
    ...
```

If you leave off the `name` argument, as in the second example above, Django will use the function's name as the tag name.

Passing template variables to the tag

Although you can pass any number of arguments to a template tag using `token.split_contents()`, the arguments are all unpacked as string literals. A little more work is required in order to pass dynamic content (a template variable) to a template tag as an argument.

While the previous examples have formatted the current time into a string and returned the string, suppose you wanted to pass in a *`DateTimeField`* from an object and have the template tag format that date-time:

```
<p>This post was last updated at {% format_time blog_entry.date_updated "%Y-%m-%d %I:%M %p" %}.</p>
```

Initially, `token.split_contents()` will return three values:

1. The tag name `format_time`.
2. The string `'blog_entry.date_updated'` (without the surrounding quotes).
3. The formatting string `'"%Y-%m-%d %I:%M %p"'`. The return value from `split_contents()` will include the leading and trailing quotes for string literals like this.

Now your tag should begin to look like this:

```
from django import template

def do_format_time(parser, token):
    try:
        # split_contents() knows not to split quoted strings.
        tag_name, date_to_be_formatted, format_string = token.split_contents()
    except ValueError:
        raise template.TemplateSyntaxError(
            "%r tag requires exactly two arguments" % token.contents.split()[0]
        )
    if not (format_string[0] == format_string[-1] and format_string[0] in ('"', "'")):
        raise template.TemplateSyntaxError(
            "%r tag's argument should be in quotes" % tag_name
```

```
        )
    return FormatTimeNode(date_to_be_formatted, format_string[1:-1])
```

You also have to change the renderer to retrieve the actual contents of the `date_updated` property of the `blog_entry` object. This can be accomplished by using the `Variable()` class in `django.template`.

To use the `Variable` class, simply instantiate it with the name of the variable to be resolved, and then call `variable.resolve(context)`. So, for example:

```
class FormatTimeNode(template.Node):
    def __init__(self, date_to_be_formatted, format_string):
        self.date_to_be_formatted = template.Variable(date_to_be_formatted)
        self.format_string = format_string

    def render(self, context):
        try:
            actual_date = self.date_to_be_formatted.resolve(context)
            return actual_date.strftime(self.format_string)
        except template.VariableDoesNotExist:
            return ''
```

Variable resolution will throw a `VariableDoesNotExist` exception if it cannot resolve the string passed to it in the current context of the page.

Setting a variable in the context

The above examples simply output a value. Generally, it's more flexible if your template tags set template variables instead of outputting values. That way, template authors can reuse the values that your template tags create.

To set a variable in the context, just use dictionary assignment on the context object in the `render()` method. Here's an updated version of `CurrentTimeNode` that sets a template variable `current_time` instead of outputting it:

```
import datetime
from django import template

class CurrentTimeNode2(template.Node):
    def __init__(self, format_string):
        self.format_string = format_string
    def render(self, context):
        context['current_time'] = datetime.datetime.now().strftime(self.format_string)
        return ''
```

Note that `render()` returns the empty string. `render()` should always return string output. If all the template tag does is set a variable, `render()` should return the empty string.

Here's how you'd use this new version of the tag:

```
{% current_time "%Y-%M-%d %I:%M %p" %}<p>The time is {{ current_time }}.</p>
```

Variable scope in context

Any variable set in the context will only be available in the same `block` of the template in which it was assigned. This behavior is intentional; it provides a scope for variables so that they don't conflict with context in other blocks.

But, there's a problem with `CurrentTimeNode2`: The variable name `current_time` is hard-coded. This means you'll need to make sure your template doesn't use `{{ current_time }}` anywhere else, because the

`{% current_time %}` will blindly overwrite that variable's value. A cleaner solution is to make the template tag specify the name of the output variable, like so:

```
{% current_time "%Y-%M-%d %I:%M %p" as my_current_time %}
<p>The current time is {{ my_current_time }}.</p>
```

To do that, you'll need to refactor both the compilation function and `Node` class, like so:

```
import re

class CurrentTimeNode3(template.Node):
    def __init__(self, format_string, var_name):
        self.format_string = format_string
        self.var_name = var_name
    def render(self, context):
        context[self.var_name] = datetime.datetime.now().strftime(self.format_string)
        return ''

def do_current_time(parser, token):
    # This version uses a regular expression to parse tag contents.
    try:
        # Splitting by None == splitting by spaces.
        tag_name, arg = token.contents.split(None, 1)
    except ValueError:
        raise template.TemplateSyntaxError(
            "%r tag requires arguments" % token.contents.split()[0]
        )
    m = re.search(r'(.*?) as (\w+)', arg)
    if not m:
        raise template.TemplateSyntaxError("%r tag had invalid arguments" % tag_name)
    format_string, var_name = m.groups()
    if not (format_string[0] == format_string[-1] and format_string[0] in ('"', "'")):
        raise template.TemplateSyntaxError(
            "%r tag's argument should be in quotes" % tag_name
        )
    return CurrentTimeNode3(format_string[1:-1], var_name)
```

The difference here is that `do_current_time()` grabs the format string and the variable name, passing both to `CurrentTimeNode3`.

Finally, if you only need to have a simple syntax for your custom context-updating template tag, consider using the *`simple_tag()`* shortcut, which supports assigning the tag results to a template variable.

Parsing until another block tag

Template tags can work in tandem. For instance, the standard *`{% comment %}`* tag hides everything until `{% endcomment %}`. To create a template tag such as this, use `parser.parse()` in your compilation function.

Here's how a simplified `{% comment %}` tag might be implemented:

```
def do_comment(parser, token):
    nodelist = parser.parse(('endcomment',))
    parser.delete_first_token()
    return CommentNode()

class CommentNode(template.Node):
    def render(self, context):
        return ''
```

: The actual implementation of *{% comment %}* is slightly different in that it allows broken template tags to appear between `{% comment %}` and `{% endcomment %}`. It does so by calling `parser.skip_past('endcomment')` instead of `parser.parse(('endcomment',))` followed by `parser.delete_first_token()`, thus avoiding the generation of a node list.

`parser.parse()` takes a tuple of names of block tags ''to parse until''. It returns an instance of `django.template.NodeList`, which is a list of all `Node` objects that the parser encountered ''before'' it encountered any of the tags named in the tuple.

In "`nodelist = parser.parse(('endcomment',))`" in the above example, `nodelist` is a list of all nodes between the `{% comment %}` and `{% endcomment %}`, not counting `{% comment %}` and `{% endcomment %}` themselves.

After `parser.parse()` is called, the parser hasn't yet "consumed" the `{% endcomment %}` tag, so the code needs to explicitly call `parser.delete_first_token()`.

`CommentNode.render()` simply returns an empty string. Anything between `{% comment %}` and `{% endcomment %}` is ignored.

Parsing until another block tag, and saving contents

In the previous example, `do_comment()` discarded everything between `{% comment %}` and `{% endcomment %}`. Instead of doing that, it's possible to do something with the code between block tags.

For example, here's a custom template tag, `{% upper %}`, that capitalizes everything between itself and `{% endupper %}`.

Usage:

```
{% upper %}This will appear in uppercase, {{ your_name }}.{% endupper %}
```

As in the previous example, we'll use `parser.parse()`. But this time, we pass the resulting `nodelist` to the `Node`:

```
def do_upper(parser, token):
    nodelist = parser.parse(('endupper',))
    parser.delete_first_token()
    return UpperNode(nodelist)

class UpperNode(template.Node):
    def __init__(self, nodelist):
        self.nodelist = nodelist
    def render(self, context):
        output = self.nodelist.render(context)
        return output.upper()
```

The only new concept here is the `self.nodelist.render(context)` in `UpperNode.render()`.

For more examples of complex rendering, see the source code of *{% for %}* in `django/template/defaulttags.py` and *{% if %}* in `django/template/smartif.py`.

4.6 Writing a custom storage system

If you need to provide custom file storage – a common example is storing files on some remote system – you can do so by defining a custom storage class. You'll need to follow these steps:

1. Your custom storage system must be a subclass of `django.core.files.storage.Storage`:

```
from django.core.files.storage import Storage

class MyStorage(Storage):
    ...
```

2. Django must be able to instantiate your storage system without any arguments. This means that any settings should be taken from `django.conf.settings`:

```
from django.conf import settings
from django.core.files.storage import Storage

class MyStorage(Storage):
    def __init__(self, option=None):
        if not option:
            option = settings.CUSTOM_STORAGE_OPTIONS
        ...
```

3. Your storage class must implement the *`_open()`* and *`_save()`* methods, along with any other methods appropriate to your storage class. See below for more on these methods.

   In addition, if your class provides local file storage, it must override the `path()` method.

4. Your storage class must be *deconstructible* so it can be serialized when it's used on a field in a migration. As long as your field has arguments that are themselves *serializable*, you can use the `django.utils.deconstruct.deconstructible` class decorator for this (that's what Django uses on FileSystemStorage).

By default, the following methods raise *NotImplementedError* and will typically have to be overridden:

- *`Storage.delete()`*
- *`Storage.exists()`*
- *`Storage.listdir()`*
- *`Storage.size()`*
- *`Storage.url()`*

Note however that not all these methods are required and may be deliberately omitted. As it happens, it is possible to leave each method unimplemented and still have a working Storage.

By way of example, if listing the contents of certain storage backends turns out to be expensive, you might decide not to implement *Storage.listdir*.

Another example would be a backend that only handles writing to files. In this case, you would not need to implement any of the above methods.

Ultimately, which of these methods are implemented is up to you. Leaving some methods unimplemented will result in a partial (possibly broken) interface.

You'll also usually want to use hooks specifically designed for custom storage objects. These are:

`_open`(*name*, *mode='rb'*)

Required.

Called by `Storage.open()`, this is the actual mechanism the storage class uses to open the file. This must return a `File` object, though in most cases, you'll want to return some subclass here that implements logic specific to the backend storage system.

`_save`(*name*, *content*)

Called by `Storage.save()`. The `name` will already have gone through `get_valid_name()` and `get_available_name()`, and the `content` will be a `File` object itself.

Should return the actual name of name of the file saved (usually the `name` passed in, but if the storage needs to change the file name return the new name instead).

`get_valid_name`(*name*)

Returns a filename suitable for use with the underlying storage system. The `name` argument passed to this method is either the original filename sent to the server or, if `upload_to` is a callable, the filename returned by that method after any path information is removed. Override this to customize how non-standard characters are converted to safe filenames.

In older versions, this method was not called when `upload_to` was a callable.

The code provided on `Storage` retains only alpha-numeric characters, periods and underscores from the original filename, removing everything else.

`get_available_name`(*name*, *max_length=None*)

Returns a filename that is available in the storage mechanism, possibly taking the provided filename into account. The `name` argument passed to this method will have already cleaned to a filename valid for the storage system, according to the `get_valid_name()` method described above.

The length of the filename will not exceed `max_length`, if provided. If a free unique filename cannot be found, a *`SuspiciousFileOperation`* exception is raised.

If a file with `name` already exists, an underscore plus a random 7 character alphanumeric string is appended to the filename before the extension.

4.7 Deploying Django

Django's chock-full of shortcuts to make Web developer's lives easier, but all those tools are of no use if you can't easily deploy your sites. Since Django's inception, ease of deployment has been a major goal.

4.7.1 How to deploy with WSGI

Django's primary deployment platform is WSGI, the Python standard for web servers and applications.

Django's *`startproject`* management command sets up a simple default WSGI configuration for you, which you can tweak as needed for your project, and direct any WSGI-compliant application server to use.

Django includes getting-started documentation for the following WSGI servers:

How to use Django with Apache and `mod_wsgi`

Deploying Django with Apache and mod_wsgi is a tried and tested way to get Django into production.

mod_wsgi is an Apache module which can host any Python WSGI application, including Django. Django will work with any version of Apache which supports mod_wsgi.

The official mod_wsgi documentation is fantastic; it's your source for all the details about how to use mod_wsgi. You'll probably want to start with the installation and configuration documentation.

Basic configuration

Once you've got mod_wsgi installed and activated, edit your Apache server's httpd.conf file and add the following. If you are using a version of Apache older than 2.4, replace `Require all granted` with `Allow from all` and also add the line `Order deny,allow` above it.

```
WSGIScriptAlias / /path/to/mysite.com/mysite/wsgi.py
WSGIPythonPath /path/to/mysite.com

<Directory /path/to/mysite.com/mysite>
<Files wsgi.py>
Require all granted
</Files>
</Directory>
```

The first bit in the `WSGIScriptAlias` line is the base URL path you want to serve your application at (`/` indicates the root url), and the second is the location of a "WSGI file" – see below – on your system, usually inside of your project package (`mysite` in this example). This tells Apache to serve any request below the given URL using the WSGI application defined in that file.

The `WSGIPythonPath` line ensures that your project package is available for import on the Python path; in other words, that `import mysite` works.

The `<Directory>` piece just ensures that Apache can access your `wsgi.py` file.

Next we'll need to ensure this `wsgi.py` with a WSGI application object exists. As of Django version 1.4, *`startproject`* will have created one for you; otherwise, you'll need to create it. See the WSGI overview documentation for the default contents you should put in this file, and what else you can add to it.

: If multiple Django sites are run in a single mod_wsgi process, all of them will use the settings of whichever one happens to run first. This can be solved by changing:
`os.environ.setdefault("DJANGO_SETTINGS_MODULE", "{{ project_name }}.settings")`
in `wsgi.py`, to:
`os.environ["DJANGO_SETTINGS_MODULE"] = "{{ project_name }}.settings"`
or by *using mod_wsgi daemon mode* and ensuring that each site runs in its own daemon process.

Fixing `UnicodeEncodeError` for file uploads

If you get a `UnicodeEncodeError` when uploading files with file names that contain non-ASCII characters, make sure Apache is configured to accept non-ASCII file names:

```
export LANG='en_US.UTF-8'
export LC_ALL='en_US.UTF-8'
```

A common location to put this configuration is `/etc/apache2/envvars`.

See the *Files* section of the Unicode reference guide for details.

Using a `virtualenv`

If you install your project's Python dependencies inside a virtualenv, you'll need to add the path to this virtualenv's `site-packages` directory to your Python path as well. To do this, add an additional path to your

`WSGIPythonPath` directive, with multiple paths separated by a colon (`:`) if using a UNIX-like system, or a semi-colon (`;`) if using Windows. If any part of a directory path contains a space character, the complete argument string to `WSGIPythonPath` must be quoted:

```
WSGIPythonPath /path/to/mysite.com:/path/to/your/venv/lib/python3.X/site-packages
```

Make sure you give the correct path to your virtualenv, and replace `python3.X` with the correct Python version (e.g. `python3.4`).

Using `mod_wsgi` daemon mode

"Daemon mode" is the recommended mode for running mod_wsgi (on non-Windows platforms). To create the required daemon process group and delegate the Django instance to run in it, you will need to add appropriate `WSGIDaemonProcess` and `WSGIProcessGroup` directives. A further change required to the above configuration if you use daemon mode is that you can't use `WSGIPythonPath`; instead you should use the `python-path` option to `WSGIDaemonProcess`, for example:

```
WSGIDaemonProcess example.com python-path=/path/to/mysite.com:/path/to/venv/lib/python2.7/site-packa
WSGIProcessGroup example.com
```

If you want to serve your project in a subdirectory (`https://example.com/mysite` in this example), you can add `WSGIScriptAlias` to the configuration above:

```
WSGIScriptAlias /mysite /path/to/mysite.com/mysite/wsgi.py process-group=example.com
```

See the official mod_wsgi documentation for details on setting up daemon mode.

Serving files

Django doesn't serve files itself; it leaves that job to whichever Web server you choose.

We recommend using a separate Web server – i.e., one that's not also running Django – for serving media. Here are some good choices:

- Nginx
- A stripped-down version of Apache

If, however, you have no option but to serve media files on the same Apache `VirtualHost` as Django, you can set up Apache to serve some URLs as static media, and others using the mod_wsgi interface to Django.

This example sets up Django at the site root, but serves `robots.txt`, `favicon.ico`, and anything in the `/static/` and `/media/` URL space as a static file. All other URLs will be served using mod_wsgi:

```
Alias /robots.txt /path/to/mysite.com/static/robots.txt
Alias /favicon.ico /path/to/mysite.com/static/favicon.ico

Alias /media/ /path/to/mysite.com/media/
Alias /static/ /path/to/mysite.com/static/

<Directory /path/to/mysite.com/static>
Require all granted
</Directory>

<Directory /path/to/mysite.com/media>
Require all granted
</Directory>
```

```
WSGIScriptAlias / /path/to/mysite.com/mysite/wsgi.py

<Directory /path/to/mysite.com/mysite>
<Files wsgi.py>
Require all granted
</Files>
</Directory>
```

If you are using a version of Apache older than 2.4, replace `Require all granted` with `Allow from all` and also add the line `Order deny,allow` above it.

Serving the admin files

When *django.contrib.staticfiles* is in *INSTALLED_APPS*, the Django development server automatically serves the static files of the admin app (and any other installed apps). This is however not the case when you use any other server arrangement. You're responsible for setting up Apache, or whichever Web server you're using, to serve the admin files.

The admin files live in (`django/contrib/admin/static/admin`) of the Django distribution.

We **strongly** recommend using *django.contrib.staticfiles* to handle the admin files (along with a Web server as outlined in the previous section; this means using the *collectstatic* management command to collect the static files in *STATIC_ROOT*, and then configuring your Web server to serve *STATIC_ROOT* at *STATIC_URL*), but here are three other approaches:

1. Create a symbolic link to the admin static files from within your document root (this may require `+FollowSymLinks` in your Apache configuration).
2. Use an `Alias` directive, as demonstrated above, to alias the appropriate URL (probably *STATIC_URL* + `admin/`) to the actual location of the admin files.
3. Copy the admin static files so that they live within your Apache document root.

Authenticating against Django's user database from Apache

Django provides a handler to allow Apache to authenticate users directly against Django's authentication backends. See the mod_wsgi authentication documentation.

Authenticating against Django's user database from Apache

Since keeping multiple authentication databases in sync is a common problem when dealing with Apache, you can configure Apache to authenticate against Django's authentication system directly. This requires Apache version >= 2.2 and mod_wsgi >= 2.0. For example, you could:

- Serve static/media files directly from Apache only to authenticated users.
- Authenticate access to a Subversion repository against Django users with a certain permission.
- Allow certain users to connect to a WebDAV share created with mod_dav.

: If you have installed a *custom User model* and want to use this default auth handler, it must support an `is_active` attribute. If you want to use group based authorization, your custom user must have a relation named 'groups', referring to a related object that has a 'name' field. You can also specify your own custom mod_wsgi auth handler if your custom cannot conform to these requirements.

Authentication with `mod_wsgi`

: The use of `WSGIApplicationGroup %{GLOBAL}` in the configurations below presumes that your Apache instance is running only one Django application. If you are running more than one Django application, please refer to the Defining Application Groups section of the mod_wsgi docs for more information about this setting.

Make sure that mod_wsgi is installed and activated and that you have followed the steps to setup Apache with mod_wsgi.

Next, edit your Apache configuration to add a location that you want only authenticated users to be able to view:

```
WSGIScriptAlias / /path/to/mysite.com/mysite/wsgi.py
WSGIPythonPath /path/to/mysite.com

WSGIProcessGroup %{GLOBAL}
WSGIApplicationGroup %{GLOBAL}

<Location "/secret">
    AuthType Basic
    AuthName "Top Secret"
    Require valid-user
    AuthBasicProvider wsgi
    WSGIAuthUserScript /path/to/mysite.com/mysite/wsgi.py
</Location>
```

The `WSGIAuthUserScript` directive tells mod_wsgi to execute the `check_password` function in specified wsgi script, passing the user name and password that it receives from the prompt. In this example, the `WSGIAuthUserScript` is the same as the `WSGIScriptAlias` that defines your application that is created by django-admin startproject.

Using Apache 2.2 with authentication

Make sure that `mod_auth_basic` and `mod_authz_user` are loaded.

These might be compiled statically into Apache, or you might need to use LoadModule to load them dynamically in your `httpd.conf`:

```
LoadModule auth_basic_module modules/mod_auth_basic.so
LoadModule authz_user_module modules/mod_authz_user.so
```

Finally, edit your WSGI script `mysite.wsgi` to tie Apache's authentication to your site's authentication mechanisms by importing the `check_password` function:

```
import os

os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'

from django.contrib.auth.handlers.modwsgi import check_password

from django.core.handlers.wsgi import WSGIHandler
application = WSGIHandler()
```

Requests beginning with `/secret/` will now require a user to authenticate.

The mod_wsgi access control mechanisms documentation provides additional details and information about alternative methods of authentication.

Authorization with `mod_wsgi` and Django groups mod_wsgi also provides functionality to restrict a particular location to members of a group.

In this case, the Apache configuration should look like this:

```
WSGIScriptAlias / /path/to/mysite.com/mysite/wsgi.py

WSGIProcessGroup %{GLOBAL}
WSGIApplicationGroup %{GLOBAL}

<Location "/secret">
    AuthType Basic
    AuthName "Top Secret"
    AuthBasicProvider wsgi
    WSGIAuthUserScript /path/to/mysite.com/mysite/wsgi.py
    WSGIAuthGroupScript /path/to/mysite.com/mysite/wsgi.py
    Require group secret-agents
    Require valid-user
</Location>
```

To support the `WSGIAuthGroupScript` directive, the same WSGI script `mysite.wsgi` must also import the `groups_for_user` function which returns a list groups the given user belongs to.

```
from django.contrib.auth.handlers.modwsgi import check_password, groups_for_user
```

Requests for `/secret/` will now also require user to be a member of the "secret-agents" group.

How to use Django with Gunicorn

Gunicorn ('Green Unicorn') is a pure-Python WSGI server for UNIX. It has no dependencies and is easy to install and use.

Installing Gunicorn

Installing gunicorn is as easy as `pip install gunicorn`. For more details, see the gunicorn documentation.

Running Django in Gunicorn as a generic WSGI application

When Gunicorn is installed, a `gunicorn` command is available which starts the Gunicorn server process. At its simplest, gunicorn just needs to be called with the location of a module containing a WSGI application object named *application*. So for a typical Django project, invoking gunicorn would look like:

```
gunicorn myproject.wsgi
```

This will start one process running one thread listening on `127.0.0.1:8000`. It requires that your project be on the Python path; the simplest way to ensure that is to run this command from the same directory as your `manage.py` file.

See Gunicorn's deployment documentation for additional tips.

How to use Django with uWSGI

uWSGI is a fast, self-healing and developer/sysadmin-friendly application container server coded in pure C.

:

The uWSGI docs offer a tutorial covering Django, nginx, and uWSGI (one possible deployment setup of many). The docs below are focused on how to integrate Django with uWSGI.

Prerequisite: uWSGI

The uWSGI wiki describes several installation procedures. Using pip, the Python package manager, you can install any uWSGI version with a single command. For example:

```
# Install current stable version.
$ pip install uwsgi

# Or install LTS (long term support).
$ pip install https://projects.unbit.it/downloads/uwsgi-lts.tar.gz
```

: Some distributions, including Debian and Ubuntu, ship an outdated version of uWSGI that does not conform to the WSGI specification. Versions prior to 1.2.6 do not call `close` on the response object after handling a request. In those cases the *`request_finished`* signal isn't sent. This can result in idle connections to database and memcache servers.

uWSGI model uWSGI operates on a client-server model. Your Web server (e.g., nginx, Apache) communicates with a django-uwsgi "worker" process to serve dynamic content. See uWSGI's background documentation for more detail.

Configuring and starting the uWSGI server for Django uWSGI supports multiple ways to configure the process. See uWSGI's configuration documentation and examples.

Here's an example command to start a uWSGI server:

```
uwsgi --chdir=/path/to/your/project \
    --module=mysite.wsgi:application \
    --env DJANGO_SETTINGS_MODULE=mysite.settings \
    --master --pidfile=/tmp/project-master.pid \
    --socket=127.0.0.1:49152 \      # can also be a file
    --processes=5 \                 # number of worker processes
    --uid=1000 --gid=2000 \         # if root, uwsgi can drop privileges
    --harakiri=20 \                 # respawn processes taking more than 20 seconds
    --max-requests=5000 \           # respawn processes after serving 5000 requests
    --vacuum \                      # clear environment on exit
    --home=/path/to/virtual/env \   # optional path to a virtualenv
    --daemonize=/var/log/uwsgi/yourproject.log      # background the process
```

This assumes you have a top-level project package named `mysite`, and within it a module `mysite/wsgi.py` that contains a WSGI `application` object. This is the layout you'll have if you ran `django-admin startproject mysite` (using your own project name in place of `mysite`) with a recent version of Django. If this file doesn't exist, you'll need to create it. See the How to deploy with WSGI documentation for the default contents you should put in this file and what else you can add to it.

The Django-specific options here are:

- `chdir`: The path to the directory that needs to be on Python's import path – i.e., the directory containing the `mysite` package.
- `module`: The WSGI module to use – probably the `mysite.wsgi` module that *`startproject`* creates.
- `env`: Should probably contain at least `DJANGO_SETTINGS_MODULE`.

- `home`: Optional path to your project virtualenv.

Example ini configuration file:

```
[uwsgi]
chdir=/path/to/your/project
module=mysite.wsgi:application
master=True
pidfile=/tmp/project-master.pid
vacuum=True
max-requests=5000
daemonize=/var/log/uwsgi/yourproject.log
```

Example ini configuration file usage:

```
uwsgi --ini uwsgi.ini
```

Fixing `UnicodeEncodeError` for file uploads

If you get a `UnicodeEncodeError` when uploading files with file names that contain non-ASCII characters, make sure uWSGI is configured to accept non-ASCII file names by adding this to your `uwsgi.ini`:

```
env = LANG=en_US.UTF-8
```

See the *Files* section of the Unicode reference guide for details.

See the uWSGI docs on managing the uWSGI process for information on starting, stopping and reloading the uWSGI workers.

The `application` object

The key concept of deploying with WSGI is the `application` callable which the application server uses to communicate with your code. It's commonly provided as an object named `application` in a Python module accessible to the server.

The *`startproject`* command creates a file `<project_name>/wsgi.py` that contains such an `application` callable.

It's used both by Django's development server and in production WSGI deployments.

WSGI servers obtain the path to the `application` callable from their configuration. Django's built-in server, namely the *`runserver`* command, reads it from the *`WSGI_APPLICATION`* setting. By default, it's set to `<project_name>.wsgi.application`, which points to the `application` callable in `<project_name>/wsgi.py`.

Configuring the settings module

When the WSGI server loads your application, Django needs to import the settings module — that's where your entire application is defined.

Django uses the *`DJANGO_SETTINGS_MODULE`* environment variable to locate the appropriate settings module. It must contain the dotted path to the settings module. You can use a different value for development and production; it all depends on how you organize your settings.

If this variable isn't set, the default `wsgi.py` sets it to `mysite.settings`, where `mysite` is the name of your project. That's how *`runserver`* discovers the default settings file by default.

: Since environment variables are process-wide, this doesn't work when you run multiple Django sites in the same process. This happens with mod_wsgi.

To avoid this problem, use mod_wsgi's daemon mode with each site in its own daemon process, or override the value from the environment by enforcing `os.environ["DJANGO_SETTINGS_MODULE"] = "mysite.settings"` in your `wsgi.py`.

Applying WSGI middleware

To apply WSGI middleware you can simply wrap the application object. For instance you could add these lines at the bottom of `wsgi.py`:

```
from helloworld.wsgi import HelloWorldApplication
application = HelloWorldApplication(application)
```

You could also replace the Django WSGI application with a custom WSGI application that later delegates to the Django WSGI application, if you want to combine a Django application with a WSGI application of another framework.

: Some third-party WSGI middleware do not call `close` on the response object after handling a request. In those cases the *`request_finished`* signal isn't sent. This can result in idle connections to database and memcache servers.

4.7.2 Deployment checklist

The Internet is a hostile environment. Before deploying your Django project, you should take some time to review your settings, with security, performance, and operations in mind.

Django includes many security features. Some are built-in and always enabled. Others are optional because they aren't always appropriate, or because they're inconvenient for development. For example, forcing HTTPS may not be suitable for all websites, and it's impractical for local development.

Performance optimizations are another category of trade-offs with convenience. For instance, caching is useful in production, less so for local development. Error reporting needs are also widely different.

The following checklist includes settings that:

- must be set properly for Django to provide the expected level of security;
- are expected to be different in each environment;
- enable optional security features;
- enable performance optimizations;
- provide error reporting.

Many of these settings are sensitive and should be treated as confidential. If you're releasing the source code for your project, a common practice is to publish suitable settings for development, and to use a private settings module for production.

Run `manage.py check --deploy`

Some of the checks described below can be automated using the *`check --deploy`* option. Be sure to run it against your production settings file as described in the option's documentation.

Critical settings

SECRET_KEY

The secret key must be a large random value and it must be kept secret.

Make sure that the key used in production isn't used anywhere else and avoid committing it to source control. This reduces the number of vectors from which an attacker may acquire the key.

Instead of hardcoding the secret key in your settings module, consider loading it from an environment variable:

```
import os
SECRET_KEY = os.environ['SECRET_KEY']
```

or from a file:

```
with open('/etc/secret_key.txt') as f:
    SECRET_KEY = f.read().strip()
```

DEBUG

You must never enable debug in production.

You're certainly developing your project with *DEBUG = True*, since this enables handy features like full tracebacks in your browser.

For a production environment, though, this is a really bad idea, because it leaks lots of information about your project: excerpts of your source code, local variables, settings, libraries used, etc.

Environment-specific settings

ALLOWED_HOSTS

When *DEBUG = False*, Django doesn't work at all without a suitable value for *ALLOWED_HOSTS*.

This setting is required to protect your site against some CSRF attacks. If you use a wildcard, you must perform your own validation of the `Host` HTTP header, or otherwise ensure that you aren't vulnerable to this category of attacks.

You should also configure the Web server that sits in front of Django to validate the host. It should respond with a static error page or ignore requests for incorrect hosts instead of forwarding the request to Django. This way you'll avoid spurious errors in your Django logs (or emails if you have error reporting configured that way). For example, on nginx you might setup a default server to return "444 No Response" on an unrecognized host:

```
server {
    listen 80 default_server;
    return 444;
}
```

CACHES

If you're using a cache, connection parameters may be different in development and in production.

Cache servers often have weak authentication. Make sure they only accept connections from your application servers.

If you're using Memcached, consider using *cached sessions* to improve performance.

DATABASES

Database connection parameters are probably different in development and in production.

Database passwords are very sensitive. You should protect them exactly like *SECRET_KEY*.

For maximum security, make sure database servers only accept connections from your application servers.

If you haven't set up backups for your database, do it right now!

EMAIL_BACKEND and related settings

If your site sends emails, these values need to be set correctly.

By default, Django sends email from webmaster@localhost and root@localhost. However, some mail providers reject email from these addresses. To use different sender addresses, modify the *DEFAULT_FROM_EMAIL* and *SERVER_EMAIL* settings.

STATIC_ROOT and STATIC_URL

Static files are automatically served by the development server. In production, you must define a *STATIC_ROOT* directory where *collectstatic* will copy them.

See Managing static files (e.g. images, JavaScript, CSS) for more information.

MEDIA_ROOT and MEDIA_URL

Media files are uploaded by your users. They're untrusted! Make sure your web server never attempts to interpret them. For instance, if a user uploads a `.php` file, the web server shouldn't execute it.

Now is a good time to check your backup strategy for these files.

HTTPS

Any website which allows users to log in should enforce site-wide HTTPS to avoid transmitting access tokens in clear. In Django, access tokens include the login/password, the session cookie, and password reset tokens. (You can't do much to protect password reset tokens if you're sending them by email.)

Protecting sensitive areas such as the user account or the admin isn't sufficient, because the same session cookie is used for HTTP and HTTPS. Your web server must redirect all HTTP traffic to HTTPS, and only transmit HTTPS requests to Django.

Once you've set up HTTPS, enable the following settings.

CSRF_COOKIE_SECURE

Set this to `True` to avoid transmitting the CSRF cookie over HTTP accidentally.

SESSION_COOKIE_SECURE

Set this to `True` to avoid transmitting the session cookie over HTTP accidentally.

Performance optimizations

Setting *DEBUG = False* disables several features that are only useful in development. In addition, you can tune the following settings.

CONN_MAX_AGE

Enabling *persistent database connections* can result in a nice speed-up when connecting to the database accounts for a significant part of the request processing time.

This helps a lot on virtualized hosts with limited network performance.

TEMPLATES

Enabling the cached template loader often improves performance drastically, as it avoids compiling each template every time it needs to be rendered. See the *template loaders docs* for more information.

Error reporting

By the time you push your code to production, it's hopefully robust, but you can't rule out unexpected errors. Thankfully, Django can capture errors and notify you accordingly.

LOGGING

Review your logging configuration before putting your website in production, and check that it works as expected as soon as you have received some traffic.

See Logging for details on logging.

ADMINS and MANAGERS

ADMINS will be notified of 500 errors by email.

MANAGERS will be notified of 404 errors. *IGNORABLE_404_URLS* can help filter out spurious reports.

See Error reporting for details on error reporting by email.

Error reporting by email doesn't scale very well

Consider using an error monitoring system such as Sentry before your inbox is flooded by reports. Sentry can also aggregate logs.

Customize the default error views

Django includes default views and templates for several HTTP error codes. You may want to override the default templates by creating the following templates in your root template directory: `404.html`, `500.html`, `403.html`, and `400.html`. The default views should suffice for 99% of Web applications, but if you desire to customize them, see these instructions which also contain details about the default templates:

- *The 404 (page not found) view*

- *The 500 (server error) view*
- *The 403 (HTTP Forbidden) view*
- *The 400 (bad request) view*

Python Options

It's strongly recommended that you invoke the Python process running your Django application using the -R option or with the `PYTHONHASHSEED` environment variable set to `random`. This option is enabled by default starting with Python 3.3.

These options help protect your site from denial-of-service (DoS) attacks triggered by carefully crafted inputs. Such an attack can drastically increase CPU usage by causing worst-case performance when creating `dict` instances. See oCERT advisory #2011-003 for more information.

If you're new to deploying Django and/or Python, we'd recommend you try mod_wsgi first. In most cases it'll be the easiest, fastest, and most stable deployment choice.

4.8 Upgrading Django to a newer version

While it can be a complex process at times, upgrading to the latest Django version has several benefits:

- New features and improvements are added.
- Bugs are fixed.
- Older version of Django will eventually no longer receive security updates. (see *Supported versions*).
- Upgrading as each new Django release is available makes future upgrades less painful by keeping your code base up to date.

Here are some things to consider to help make your upgrade process as smooth as possible.

4.8.1 Required Reading

If it's your first time doing an upgrade, it is useful to read the guide on the different release processes.

Afterwards, you should familiarize yourself with the changes that were made in the new Django version(s):

- Read the release notes for each 'final' release from the one after your current Django version, up to and including the version to which you plan to upgrade.
- Look at the deprecation timeline for the relevant versions.

Pay particular attention to backwards incompatible changes to get a clear idea of what will be needed for a successful upgrade.

4.8.2 Dependencies

In most cases it will be necessary to upgrade to the latest version of your Django-related dependencies as well. If the Django version was recently released or if some of your dependencies are not well-maintained, some of your dependencies may not yet support the new Django version. In these cases you may have to wait until new versions of your dependencies are released.

4.8.3 Resolving deprecation warnings

Before upgrading, it's a good idea to resolve any deprecation warnings raised by your project while using your current version of Django. Fixing these warnings before upgrading ensures that you're informed about areas of the code that need altering.

In Python, deprecation warnings are silenced by default. You must turn them on using the `-Wall` Python command line option or the `PYTHONWARNINGS` environment variable. For example, to show warnings while running tests:

```
$ python -Wall manage.py test
```

If you're not using the Django test runner, you may need to also ensure that any console output is not captured which would hide deprecation warnings. For example, if you use *py.test*:

```
$ PYTHONWARNINGS=all py.test tests --capture=no
```

Resolve any deprecation warnings with your current version of Django before continuing the upgrade process.

Third party applications might use deprecated APIs in order to support multiple versions of Django, so deprecation warnings in packages you've installed don't necessarily indicate a problem. If a package doesn't support the latest version of Django, consider raising an issue or sending a pull request for it.

4.8.4 Installation

Once you're ready, it is time to install the new Django version. If you are using virtualenv and it is a major upgrade, you might want to set up a new environment with all the dependencies first.

Exactly which steps you will need to take depends on your installation process. The most convenient way is to use pip with the `--upgrade` or `-U` flag:

```
$ pip install -U Django
```

pip also automatically uninstalls the previous version of Django.

If you use some other installation process, you might have to manually *uninstall the old Django version* and should look at the complete installation instructions.

4.8.5 Testing

When the new environment is set up, run the full test suite for your application. Again, it's useful to turn on deprecation warnings on so they're shown in the test output (you can also use the flag if you test your app manually using `manage.py runserver`):

```
$ python -Wall manage.py test
```

After you have run the tests, fix any failures. While you have the release notes fresh in your mind, it may also be a good time to take advantage of new features in Django by refactoring your code to eliminate any deprecation warnings.

4.8.6 Deployment

When you are sufficiently confident your app works with the new version of Django, you're ready to go ahead and deploy your upgraded Django project.

If you are using caching provided by Django, you should consider clearing your cache after upgrading. Otherwise you may run into problems, for example, if you are caching pickled objects as these objects are not guaranteed to be pickle-compatible across Django versions. A past instance of incompatibility was caching pickled *`HttpResponse`* objects, either directly or indirectly via the *`cache_page()`* decorator.

4.9 Error reporting

When you're running a public site you should always turn off the *DEBUG* setting. That will make your server run much faster, and will also prevent malicious users from seeing details of your application that can be revealed by the error pages.

However, running with *DEBUG* set to `False` means you'll never see errors generated by your site – everyone will just see your public error pages. You need to keep track of errors that occur in deployed sites, so Django can be configured to create reports with details about those errors.

4.9.1 Email reports

Server errors

When *DEBUG* is `False`, Django will email the users listed in the *ADMINS* setting whenever your code raises an unhandled exception and results in an internal server error (HTTP status code 500). This gives the administrators immediate notification of any errors. The *ADMINS* will get a description of the error, a complete Python traceback, and details about the HTTP request that caused the error.

: In order to send email, Django requires a few settings telling it how to connect to your mail server. At the very least, you'll need to specify *EMAIL_HOST* and possibly *EMAIL_HOST_USER* and *EMAIL_HOST_PASSWORD*, though other settings may be also required depending on your mail server's configuration. Consult the Django settings documentation for a full list of email-related settings.

By default, Django will send email from root@localhost. However, some mail providers reject all email from this address. To use a different sender address, modify the *SERVER_EMAIL* setting.

To activate this behavior, put the email addresses of the recipients in the *ADMINS* setting.

:

Server error emails are sent using the logging framework, so you can customize this behavior by customizing your logging configuration.

404 errors

Django can also be configured to email errors about broken links (404 "page not found" errors). Django sends emails about 404 errors when:

- *DEBUG* is `False`;
- Your *MIDDLEWARE* setting includes *django.middleware.common.BrokenLinkEmailsMiddleware*.

If those conditions are met, Django will email the users listed in the *MANAGERS* setting whenever your code raises a 404 and the request has a referer. It doesn't bother to email for 404s that don't have a referer – those are usually just people typing in broken URLs or broken Web bots. It also ignores 404s when the referer is equal to the requested URL, since this behavior is from broken Web bots too.

In older versions, 404s were not ignored when the referer was equal to the requested URL.

: *BrokenLinkEmailsMiddleware* must appear before other middleware that intercepts 404 errors, such as *LocaleMiddleware* or *FlatpageFallbackMiddleware*. Put it towards the top of your *MIDDLEWARE* setting.

You can tell Django to stop reporting particular 404s by tweaking the *IGNORABLE_404_URLS* setting. It should be a list of compiled regular expression objects. For example:

```
import re
IGNORABLE_404_URLS = [
    re.compile(r'\.(php|cgi)$'),
    re.compile(r'^/phpmyadmin/'),
]
```

In this example, a 404 to any URL ending with `.php` or `.cgi` will *not* be reported. Neither will any URL starting with `/phpmyadmin/`.

The following example shows how to exclude some conventional URLs that browsers and crawlers often request:

```
import re
IGNORABLE_404_URLS = [
    re.compile(r'^/apple-touch-icon.*\.png$'),
    re.compile(r'^/favicon\.ico$'),
    re.compile(r'^/robots\.txt$'),
]
```

(Note that these are regular expressions, so we put a backslash in front of periods to escape them.)

If you'd like to customize the behavior of *django.middleware.common.BrokenLinkEmailsMiddleware* further (for example to ignore requests coming from web crawlers), you should subclass it and override its methods.

:

404 errors are logged using the logging framework. By default, these log records are ignored, but you can use them for error reporting by writing a handler and configuring logging appropriately.

4.9.2 Filtering error reports

: Filtering sensitive data is a hard problem, and it's nearly impossible to guarantee that sensitive won't leak into an error report. Therefore, error reports should only be available to trusted team members and you should avoid transmitting error reports unencrypted over the Internet (such as through email).

Filtering sensitive information

Error reports are really helpful for debugging errors, so it is generally useful to record as much relevant information about those errors as possible. For example, by default Django records the full traceback for the exception raised, each traceback frame's local variables, and the *HttpRequest*'s *attributes*.

However, sometimes certain types of information may be too sensitive and thus may not be appropriate to be kept track of, for example a user's password or credit card number. So in addition to filtering out settings that appear to be sensitive as described in the *DEBUG* documentation, Django offers a set of function decorators to help you control which information should be filtered out of error reports in a production environment (that is, where *DEBUG* is set to `False`): *sensitive_variables()* and *sensitive_post_parameters()*.

sensitive_variables(**variables*)

If a function (either a view or any regular callback) in your code uses local variables susceptible to contain sensitive information, you may prevent the values of those variables from being included in error reports using the `sensitive_variables` decorator:

```
from django.views.decorators.debug import sensitive_variables

@sensitive_variables('user', 'pw', 'cc')
def process_info(user):
    pw = user.pass_word
    cc = user.credit_card_number
    name = user.name
    ...
```

In the above example, the values for the `user`, `pw` and `cc` variables will be hidden and replaced with stars (**********) in the error reports, whereas the value of the `name` variable will be disclosed.

To systematically hide all local variables of a function from error logs, do not provide any argument to the `sensitive_variables` decorator:

```
@sensitive_variables()
def my_function():
    ...
```

When using multiple decorators

If the variable you want to hide is also a function argument (e.g. '`user`' in the following example), and if the decorated function has multiple decorators, then make sure to place `@sensitive_variables` at the top of the decorator chain. This way it will also hide the function argument as it gets passed through the other decorators:

```
@sensitive_variables('user', 'pw', 'cc')
@some_decorator
@another_decorator
def process_info(user):
    ...
```

---

sensitive_post_parameters(**parameters*)

If one of your views receives an *HttpRequest* object with *POST parameters* susceptible to contain sensitive information, you may prevent the values of those parameters from being included in the error reports using the `sensitive_post_parameters` decorator:

```
from django.views.decorators.debug import sensitive_post_parameters

@sensitive_post_parameters('pass_word', 'credit_card_number')
def record_user_profile(request):
    UserProfile.create(
        user=request.user,
        password=request.POST['pass_word'],
        credit_card=request.POST['credit_card_number'],
        name=request.POST['name'],
    )
    ...
```

In the above example, the values for the `pass_word` and `credit_card_number` POST parameters will be hidden and replaced with stars (**********) in the request's representation inside the error reports, whereas the value of the `name` parameter will be disclosed.

To systematically hide all POST parameters of a request in error reports, do not provide any argument to the `sensitive_post_parameters` decorator:

```
    @sensitive_post_parameters()
    def my_view(request):
        ...
```

All POST parameters are systematically filtered out of error reports for certain *django.contrib.auth.views* views (`login`, `password_reset_confirm`, `password_change`, and `add_view` and `user_change_password` in the `auth` admin) to prevent the leaking of sensitive information such as user passwords.

Custom error reports

All *sensitive_variables()* and *sensitive_post_parameters()* do is, respectively, annotate the decorated function with the names of sensitive variables and annotate the `HttpRequest` object with the names of sensitive POST parameters, so that this sensitive information can later be filtered out of reports when an error occurs. The actual filtering is done by Django's default error reporter filter: *django.views.debug.SafeExceptionReporterFilter*. This filter uses the decorators' annotations to replace the corresponding values with stars (**********) when the error reports are produced. If you wish to override or customize this default behavior for your entire site, you need to define your own filter class and tell Django to use it via the *DEFAULT_EXCEPTION_REPORTER_FILTER* setting:

```
DEFAULT_EXCEPTION_REPORTER_FILTER = 'path.to.your.CustomExceptionReporterFilter'
```

You may also control in a more granular way which filter to use within any given view by setting the `HttpRequest`'s `exception_reporter_filter` attribute:

```
def my_view(request):
    if request.user.is_authenticated:
        request.exception_reporter_filter = CustomExceptionReporterFilter()
    ...
```

Your custom filter class needs to inherit from *django.views.debug.SafeExceptionReporterFilter* and may override the following methods:

class SafeExceptionReporterFilter

`SafeExceptionReporterFilter.`**is_active**(*request*)
Returns `True` to activate the filtering operated in the other methods. By default the filter is active if *DEBUG* is `False`.

`SafeExceptionReporterFilter.`**get_post_parameters**(*request*)
Returns the filtered dictionary of POST parameters. By default it replaces the values of sensitive parameters with stars (**********).

`SafeExceptionReporterFilter.`**get_traceback_frame_variables**(*request*, *tb_frame*)
Returns the filtered dictionary of local variables for the given traceback frame. By default it replaces the values of sensitive variables with stars (**********).

:

You can also set up custom error reporting by writing a custom piece of *exception middleware*. If you do write custom error handling, it's a good idea to emulate Django's built-in error handling and only report/log errors if *DEBUG* is `False`.

4.10 Providing initial data for models

It's sometimes useful to pre-populate your database with hard-coded data when you're first setting up an app. You can provide initial data via fixtures.

4.10.1 Providing initial data with fixtures

A fixture is a collection of data that Django knows how to import into a database. The most straightforward way of creating a fixture if you've already got some data is to use the *manage.py dumpdata* command. Or, you can write fixtures by hand; fixtures can be written as JSON, XML or YAML (with PyYAML installed) documents. The serialization documentation has more details about each of these supported *serialization formats*.

As an example, though, here's what a fixture for a simple `Person` model might look like in JSON:

```
[
  {
    "model": "myapp.person",
    "pk": 1,
    "fields": {
      "first_name": "John",
      "last_name": "Lennon"
    }
  },
  {
    "model": "myapp.person",
    "pk": 2,
    "fields": {
      "first_name": "Paul",
      "last_name": "McCartney"
    }
  }
]
```

And here's that same fixture as YAML:

```
- model: myapp.person
  pk: 1
  fields:
    first_name: John
    last_name: Lennon
- model: myapp.person
  pk: 2
  fields:
    first_name: Paul
    last_name: McCartney
```

You'll store this data in a `fixtures` directory inside your app.

Loading data is easy: just call *manage.py loaddata* `<fixturename>`, where `<fixturename>` is the name of the fixture file you've created. Each time you run *loaddata*, the data will be read from the fixture and re-loaded into the database. Note this means that if you change one of the rows created by a fixture and then run *loaddata* again, you'll wipe out any changes you've made.

Where Django finds fixture files

By default, Django looks in the `fixtures` directory inside each app for fixtures. You can set the *`FIXTURE_DIRS`* setting to a list of additional directories where Django should look.

When running *`manage.py loaddata`*, you can also specify a path to a fixture file, which overrides searching the usual directories.

:

Fixtures are also used by the *testing framework* to help set up a consistent test environment.

4.11 Running Django on Jython

Jython is an implementation of Python that runs on the Java platform (JVM). This document will get you up and running with Django on top of Jython.

4.11.1 Installing Jython

Django works with Jython versions 2.7b2 and higher. See the Jython website for download and installation instructions.

4.11.2 Creating a servlet container

If you just want to experiment with Django, skip ahead to the next section; Django includes a lightweight Web server you can use for testing, so you won't need to set up anything else until you're ready to deploy Django in production.

If you want to use Django on a production site, use a Java servlet container, such as Apache Tomcat. Full JavaEE applications servers such as GlassFish or JBoss are also OK, if you need the extra features they include.

4.11.3 Installing Django

The next step is to install Django itself. This is exactly the same as installing Django on standard Python, so see *Remove any old versions of Django* and *Install the Django code* for instructions.

4.11.4 Installing Jython platform support libraries

The django-jython project contains database backends and management commands for Django/Jython development. Note that the builtin Django backends won't work on top of Jython.

To install it, follow the installation instructions detailed on the project website. Also, read the database backends documentation there.

4.11.5 Differences with Django on Jython

At this point, Django on Jython should behave nearly identically to Django running on standard Python. However, are a few differences to keep in mind:

- Remember to use the `jython` command instead of `python`. The documentation uses `python` for consistency, but if you're using Jython you'll want to mentally replace `python` with `jython` every time it occurs.

- Similarly, you'll need to use the `JYTHONPATH` environment variable instead of `PYTHONPATH`.
- Any part of Django that requires Pillow will not work.

4.12 Integrating Django with a legacy database

While Django is best suited for developing new applications, it's quite possible to integrate it into legacy databases. Django includes a couple of utilities to automate as much of this process as possible.

This document assumes you know the Django basics, as covered in the tutorial.

Once you've got Django set up, you'll follow this general process to integrate with an existing database.

4.12.1 Give Django your database parameters

You'll need to tell Django what your database connection parameters are, and what the name of the database is. Do that by editing the *DATABASES* setting and assigning values to the following keys for the `'default'` connection:

- *NAME*
- *ENGINE*
- *USER*
- *PASSWORD*
- *HOST*
- *PORT*

4.12.2 Auto-generate the models

Django comes with a utility called *inspectdb* that can create models by introspecting an existing database. You can view the output by running this command:

```
$ python manage.py inspectdb
```

Save this as a file by using standard Unix output redirection:

```
$ python manage.py inspectdb > models.py
```

This feature is meant as a shortcut, not as definitive model generation. See the *documentation of inspectdb* for more information.

Once you've cleaned up your models, name the file `models.py` and put it in the Python package that holds your app. Then add the app to your *INSTALLED_APPS* setting.

By default, *inspectdb* creates unmanaged models. That is, `managed = False` in the model's `Meta` class tells Django not to manage each table's creation, modification, and deletion:

```
class Person(models.Model):
    id = models.IntegerField(primary_key=True)
    first_name = models.CharField(max_length=70)
    class Meta:
        managed = False
        db_table = 'CENSUS_PERSONS'
```

If you do want to allow Django to manage the table's lifecycle, you'll need to change the *managed* option above to `True` (or simply remove it because `True` is its default value).

4.12.3 Install the core Django tables

Next, run the *migrate* command to install any extra needed database records such as admin permissions and content types:

```
$ python manage.py migrate
```

4.12.4 Test and tweak

Those are the basic steps – from here you'll want to tweak the models Django generated until they work the way you'd like. Try accessing your data via the Django database API, and try editing objects via Django's admin site, and edit the models file accordingly.

4.13 Outputting CSV with Django

This document explains how to output CSV (Comma Separated Values) dynamically using Django views. To do this, you can either use the Python CSV library or the Django template system.

4.13.1 Using the Python CSV library

Python comes with a CSV library, `csv`. The key to using it with Django is that the `csv` module's CSV-creation capability acts on file-like objects, and Django's *HttpResponse* objects are file-like objects.

Here's an example:

```
import csv
from django.http import HttpResponse

def some_view(request):
    # Create the HttpResponse object with the appropriate CSV header.
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment; filename="somefilename.csv"'

    writer = csv.writer(response)
    writer.writerow(['First row', 'Foo', 'Bar', 'Baz'])
    writer.writerow(['Second row', 'A', 'B', 'C', '"Testing"', "Here's a quote"])

    return response
```

The code and comments should be self-explanatory, but a few things deserve a mention:

- The response gets a special MIME type, *text/csv*. This tells browsers that the document is a CSV file, rather than an HTML file. If you leave this off, browsers will probably interpret the output as HTML, which will result in ugly, scary gobbledygook in the browser window.
- The response gets an additional `Content-Disposition` header, which contains the name of the CSV file. This filename is arbitrary; call it whatever you want. It'll be used by browsers in the "Save as..." dialog, etc.
- Hooking into the CSV-generation API is easy: Just pass `response` as the first argument to `csv.writer`. The `csv.writer` function expects a file-like object, and *HttpResponse* objects fit the bill.
- For each row in your CSV file, call `writer.writerow`, passing it an iterable object such as a list or tuple.
- The CSV module takes care of quoting for you, so you don't have to worry about escaping strings with quotes or commas in them. Just pass `writerow()` your raw strings, and it'll do the right thing.

Handling Unicode on Python 2

Python 2's `csv` module does not support Unicode input. Since Django uses Unicode internally this means strings read from sources such as *`HttpRequest`* are potentially problematic. There are a few options for handling this:

- Manually encode all Unicode objects to a compatible encoding.
- Use the `UnicodeWriter` class provided in the csv module's examples section.
- Use the python-unicodecsv module, which aims to be a drop-in replacement for `csv` that gracefully handles Unicode.

For more information, see the Python documentation of the `csv` module.

Streaming large CSV files

When dealing with views that generate very large responses, you might want to consider using Django's *`StreamingHttpResponse`* instead. For example, by streaming a file that takes a long time to generate you can avoid a load balancer dropping a connection that might have otherwise timed out while the server was generating the response.

In this example, we make full use of Python generators to efficiently handle the assembly and transmission of a large CSV file:

```
import csv

from django.utils.six.moves import range
from django.http import StreamingHttpResponse

class Echo(object):
    """An object that implements just the write method of the file-like
    interface.
    """
    def write(self, value):
        """Write the value by returning it, instead of storing in a buffer."""
        return value

def some_streaming_csv_view(request):
    """A view that streams a large CSV file."""
    # Generate a sequence of rows. The range is based on the maximum number of
    # rows that can be handled by a single sheet in most spreadsheet
    # applications.
    rows = (["Row {}".format(idx), str(idx)] for idx in range(65536))
    pseudo_buffer = Echo()
    writer = csv.writer(pseudo_buffer)
    response = StreamingHttpResponse((writer.writerow(row) for row in rows),
                                     content_type="text/csv")
    response['Content-Disposition'] = 'attachment; filename="somefilename.csv"'
    return response
```

4.13.2 Using the template system

Alternatively, you can use the Django template system to generate CSV. This is lower-level than using the convenient Python `csv` module, but the solution is presented here for completeness.

The idea here is to pass a list of items to your template, and have the template output the commas in a *`for`* loop.

Here's an example, which generates the same CSV file as above:

```
from django.http import HttpResponse
from django.template import loader, Context

def some_view(request):
    # Create the HttpResponse object with the appropriate CSV header.
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment; filename="somefilename.csv"'

    # The data is hard-coded here, but you could load it from a database or
    # some other source.
    csv_data = (
        ('First row', 'Foo', 'Bar', 'Baz'),
        ('Second row', 'A', 'B', 'C', '"Testing"', "Here's a quote"),
    )

    t = loader.get_template('my_template_name.txt')
    c = Context({
        'data': csv_data,
    })
    response.write(t.render(c))
    return response
```

The only difference between this example and the previous example is that this one uses template loading instead of the CSV module. The rest of the code – such as the `content_type='text/csv'` – is the same.

Then, create the template `my_template_name.txt`, with this template code:

```
{% for row in data %}"{{ row.0|addslashes }}", "{{ row.1|addslashes }}", "{{ row.2|addslashes }}", "
{% endfor %}
```

This template is quite basic. It just iterates over the given data and displays a line of CSV for each row. It uses the *addslashes* template filter to ensure there aren't any problems with quotes.

4.13.3 Other text-based formats

Notice that there isn't very much specific to CSV here – just the specific output format. You can use either of these techniques to output any text-based format you can dream of. You can also use a similar technique to generate arbitrary binary data; see Outputting PDFs with Django for an example.

4.14 Outputting PDFs with Django

This document explains how to output PDF files dynamically using Django views. This is made possible by the excellent, open-source ReportLab Python PDF library.

The advantage of generating PDF files dynamically is that you can create customized PDFs for different purposes – say, for different users or different pieces of content.

For example, Django was used at kusports.com to generate customized, printer-friendly NCAA tournament brackets, as PDF files, for people participating in a March Madness contest.

4.14.1 Install ReportLab

The ReportLab library is available on PyPI. A user guide (not coincidentally, a PDF file) is also available for download. You can install ReportLab with `pip`:

```
$ pip install reportlab
```

Test your installation by importing it in the Python interactive interpreter:

```
>>> import reportlab
```

If that command doesn't raise any errors, the installation worked.

4.14.2 Write your view

The key to generating PDFs dynamically with Django is that the ReportLab API acts on file-like objects, and Django's *`HttpResponse`* objects are file-like objects.

Here's a "Hello World" example:

```
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def some_view(request):
    # Create the HttpResponse object with the appropriate PDF headers.
    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = 'attachment; filename="somefilename.pdf"'

    # Create the PDF object, using the response object as its "file."
    p = canvas.Canvas(response)

    # Draw things on the PDF. Here's where the PDF generation happens.
    # See the ReportLab documentation for the full list of functionality.
    p.drawString(100, 100, "Hello world.")

    # Close the PDF object cleanly, and we're done.
    p.showPage()
    p.save()
    return response
```

The code and comments should be self-explanatory, but a few things deserve a mention:

- The response gets a special MIME type, *`application/pdf`*. This tells browsers that the document is a PDF file, rather than an HTML file. If you leave this off, browsers will probably interpret the output as HTML, which would result in ugly, scary gobbledygook in the browser window.
- The response gets an additional `Content-Disposition` header, which contains the name of the PDF file. This filename is arbitrary: Call it whatever you want. It'll be used by browsers in the "Save as..." dialog, etc.
- The `Content-Disposition` header starts with `'attachment; '` in this example. This forces Web browsers to pop-up a dialog box prompting/confirming how to handle the document even if a default is set on the machine. If you leave off `'attachment;'`, browsers will handle the PDF using whatever program/plugin they've been configured to use for PDFs. Here's what that code would look like:

```
    response['Content-Disposition'] = 'filename="somefilename.pdf"'
```

- Hooking into the ReportLab API is easy: Just pass `response` as the first argument to `canvas.Canvas`. The `Canvas` class expects a file-like object, and *`HttpResponse`* objects fit the bill.

- Note that all subsequent PDF-generation methods are called on the PDF object (in this case, `p`) – not on `response`.
- Finally, it's important to call `showPage()` and `save()` on the PDF file.

: ReportLab is not thread-safe. Some of our users have reported odd issues with building PDF-generating Django views that are accessed by many people at the same time.

4.14.3 Complex PDFs

If you're creating a complex PDF document with ReportLab, consider using the `io` library as a temporary holding place for your PDF file. This library provides a file-like object interface that is particularly efficient. Here's the above "Hello World" example rewritten to use `io`:

```
from io import BytesIO
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def some_view(request):
    # Create the HttpResponse object with the appropriate PDF headers.
    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = 'attachment; filename="somefilename.pdf"'

    buffer = BytesIO()

    # Create the PDF object, using the BytesIO object as its "file."
    p = canvas.Canvas(buffer)

    # Draw things on the PDF. Here's where the PDF generation happens.
    # See the ReportLab documentation for the full list of functionality.
    p.drawString(100, 100, "Hello world.")

    # Close the PDF object cleanly.
    p.showPage()
    p.save()

    # Get the value of the BytesIO buffer and write it to the response.
    pdf = buffer.getvalue()
    buffer.close()
    response.write(pdf)
    return response
```

4.14.4 Other formats

Notice that there isn't a lot in these examples that's PDF-specific – just the bits using `reportlab`. You can use a similar technique to generate any arbitrary format that you can find a Python library for. Also see Outputting CSV with Django for another example and some techniques you can use when generated text-based formats.

:

Django Packages provides a comparison of packages that help generate PDF files from Django.

4.15 Managing static files (e.g. images, JavaScript, CSS)

Websites generally need to serve additional files such as images, JavaScript, or CSS. In Django, we refer to these files as "static files". Django provides *django.contrib.staticfiles* to help you manage them.

This page describes how you can serve these static files.

4.15.1 Configuring static files

1. Make sure that `django.contrib.staticfiles` is included in your *INSTALLED_APPS*.
2. In your settings file, define *STATIC_URL*, for example:

```
STATIC_URL = '/static/'
```

3. In your templates, either hardcode the url like `/static/my_app/example.jpg` or, preferably, use the *static* template tag to build the URL for the given relative path by using the configured *STATICFILES_STORAGE* storage (this makes it much easier when you want to switch to a content delivery network (CDN) for serving static files).

```
{% load static %}
<img src="{% static "my_app/example.jpg" %}" alt="My image"/>
```

4. Store your static files in a folder called `static` in your app. For example `my_app/static/my_app/example.jpg`.

Serving the files

In addition to these configuration steps, you'll also need to actually serve the static files.

During development, if you use *django.contrib.staticfiles*, this will be done automatically by *runserver* when *DEBUG* is set to `True` (see *django.contrib.staticfiles.views.serve()*).

This method is **grossly inefficient** and probably **insecure**, so it is **unsuitable for production**.

See Deploying static files for proper strategies to serve static files in production environments.

Your project will probably also have static assets that aren't tied to a particular app. In addition to using a `static/` directory inside your apps, you can define a list of directories (*STATICFILES_DIRS*) in your settings file where Django will also look for static files. For example:

```
STATICFILES_DIRS = [
    os.path.join(BASE_DIR, "static"),
    '/var/www/static/',
]
```

See the documentation for the *STATICFILES_FINDERS* setting for details on how `staticfiles` finds your files.

Static file namespacing

Now we *might* be able to get away with putting our static files directly in `my_app/static/` (rather than creating another `my_app` subdirectory), but it would actually be a bad idea. Django will use the first static file it finds whose name matches, and if you had a static file with the same name in a *different* application, Django would be unable to distinguish between them. We need to be able to point Django at the right one, and the easiest way to ensure this is by *namespacing* them. That is, by putting those static files inside *another* directory named for the application itself.

4.15.2 Serving static files during development

If you use *django.contrib.staticfiles* as explained above, *runserver* will do this automatically when *DEBUG* is set to `True`. If you don't have `django.contrib.staticfiles` in *INSTALLED_APPS*, you can still manually serve static files using the *django.contrib.staticfiles.views.serve()* view.

This is not suitable for production use! For some common deployment strategies, see Deploying static files.

For example, if your *STATIC_URL* is defined as `/static/`, you can do this by adding the following snippet to your urls.py:

```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    # ... the rest of your URLconf goes here ...
] + static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

: This helper function works only in debug mode and only if the given prefix is local (e.g. `/static/`) and not a URL (e.g. `http://static.example.com/`).

Also this helper function only serves the actual *STATIC_ROOT* folder; it doesn't perform static files discovery like *django.contrib.staticfiles*.

4.15.3 Serving files uploaded by a user during development

During development, you can serve user-uploaded media files from *MEDIA_ROOT* using the *django.contrib.staticfiles.views.serve()* view.

This is not suitable for production use! For some common deployment strategies, see Deploying static files.

For example, if your *MEDIA_URL* is defined as `/media/`, you can do this by adding the following snippet to your urls.py:

```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    # ... the rest of your URLconf goes here ...
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

: This helper function works only in debug mode and only if the given prefix is local (e.g. `/media/`) and not a URL (e.g. `http://media.example.com/`).

4.15.4 Testing

When running tests that use actual HTTP requests instead of the built-in testing client (i.e. when using the built-in *LiveServerTestCase*) the static assets need to be served along the rest of the content so the test environment reproduces the real one as faithfully as possible, but `LiveServerTestCase` has only very basic static file-serving functionality: It doesn't know about the finders feature of the `staticfiles` application and assumes the static content has already been collected under *STATIC_ROOT*.

Because of this, `staticfiles` ships its own *django.contrib.staticfiles.testing.StaticLiveServerTestCase*, a subclass of the built-in one that has the ability to transparently serve all the assets during execution of these tests in a way very similar to what we get at development time with `DEBUG = True`, i.e. without having to collect them using *collectstatic* first.

4.15.5 Deployment

django.contrib.staticfiles provides a convenience management command for gathering static files in a single directory so you can serve them easily.

1. Set the *STATIC_ROOT* setting to the directory from which you'd like to serve these files, for example:

```
STATIC_ROOT = "/var/www/example.com/static/"
```

2. Run the *collectstatic* management command:

```
$ python manage.py collectstatic
```

   This will copy all files from your static folders into the *STATIC_ROOT* directory.

3. Use a web server of your choice to serve the files. Deploying static files covers some common deployment strategies for static files.

4.15.6 Learn more

This document has covered the basics and some common usage patterns. For complete details on all the settings, commands, template tags, and other pieces included in *django.contrib.staticfiles*, see the staticfiles reference.

4.16 Deploying static files

:

For an introduction to the use of *django.contrib.staticfiles*, see Managing static files (e.g. images, JavaScript, CSS).

4.16.1 Serving static files in production

The basic outline of putting static files into production is simple: run the *collectstatic* command when static files change, then arrange for the collected static files directory (*STATIC_ROOT*) to be moved to the static file server and served. Depending on *STATICFILES_STORAGE*, files may need to be moved to a new location manually or the *post_process* method of the `Storage` class might take care of that.

Of course, as with all deployment tasks, the devil's in the details. Every production setup will be a bit different, so you'll need to adapt the basic outline to fit your needs. Below are a few common patterns that might help.

Serving the site and your static files from the same server

If you want to serve your static files from the same server that's already serving your site, the process may look something like:

- Push your code up to the deployment server.

- On the server, run *collectstatic* to copy all the static files into *STATIC_ROOT*.
- Configure your web server to serve the files in *STATIC_ROOT* under the URL *STATIC_URL*. For example, here's *how to do this with Apache and mod_wsgi*.

You'll probably want to automate this process, especially if you've got multiple web servers. There's any number of ways to do this automation, but one option that many Django developers enjoy is Fabric.

Below, and in the following sections, we'll show off a few example fabfiles (i.e. Fabric scripts) that automate these file deployment options. The syntax of a fabfile is fairly straightforward but won't be covered here; consult Fabric's documentation, for a complete explanation of the syntax.

So, a fabfile to deploy static files to a couple of web servers might look something like:

```
from fabric.api import *

# Hosts to deploy onto
env.hosts = ['www1.example.com', 'www2.example.com']

# Where your project code lives on the server
env.project_root = '/home/www/myproject'

def deploy_static():
    with cd(env.project_root):
        run('./manage.py collectstatic -v0 --noinput')
```

Serving static files from a dedicated server

Most larger Django sites use a separate Web server – i.e., one that's not also running Django – for serving static files. This server often runs a different type of web server – faster but less full-featured. Some common choices are:

- Nginx
- A stripped-down version of Apache

Configuring these servers is out of scope of this document; check each server's respective documentation for instructions.

Since your static file server won't be running Django, you'll need to modify the deployment strategy to look something like:

- When your static files change, run *collectstatic* locally.
- Push your local *STATIC_ROOT* up to the static file server into the directory that's being served. rsync is a common choice for this step since it only needs to transfer the bits of static files that have changed.

Here's how this might look in a fabfile:

```
from fabric.api import *
from fabric.contrib import project

# Where the static files get collected locally. Your STATIC_ROOT setting.
env.local_static_root = '/path/to/static'

# Where the static files should go remotely
env.remote_static_root = '/home/www/static.example.com'

@roles('static')
def deploy_static():
    local('./manage.py collectstatic')
    project.rsync_project(
```

```
        remote_dir=env.remote_static_root,
        local_dir=env.local_static_root,
        delete=True,
    )
```

Serving static files from a cloud service or CDN

Another common tactic is to serve static files from a cloud storage provider like Amazon's S3 and/or a CDN (content delivery network). This lets you ignore the problems of serving static files and can often make for faster-loading Web pages (especially when using a CDN).

When using these services, the basic workflow would look a bit like the above, except that instead of using `rsync` to transfer your static files to the server you'd need to transfer the static files to the storage provider or CDN.

There's any number of ways you might do this, but if the provider has an API a custom file storage backend will make the process incredibly simple. If you've written or are using a 3rd party custom storage backend, you can tell *collectstatic* to use it by setting *STATICFILES_STORAGE* to the storage engine.

For example, if you've written an S3 storage backend in `myproject.storage.S3Storage` you could use it with:

```
STATICFILES_STORAGE = 'myproject.storage.S3Storage'
```

Once that's done, all you have to do is run *collectstatic* and your static files would be pushed through your storage package up to S3. If you later needed to switch to a different storage provider, it could be as simple as changing your *STATICFILES_STORAGE* setting.

For details on how you'd write one of these backends, see Writing a custom storage system. There are 3rd party apps available that provide storage backends for many common file storage APIs. A good starting point is the overview at djangopackages.com.

4.16.2 Learn more

For complete details on all the settings, commands, template tags, and other pieces included in *django.contrib.staticfiles*, see the staticfiles reference.

4.17 How to install Django on Windows

This document will guide you through installing Python 3.5 and Django on Windows. It also provides instructions for installing virtualenv and virtualenvwrapper, which make it easier to work on Python projects. This is meant as a beginner's guide for users working on Django projects and does not reflect how Django should be installed when developing patches for Django itself.

The steps in this guide have been tested with Windows 7, 8, and 10. In other versions, the steps would be similar. You will need to be familiar with using the Windows command prompt.

4.17.1 Install Python

Django is a Python web framework, thus requiring Python to be installed on your machine. At the time of writing, Python 3.5 is the latest version.

To install Python on your machine go to https://python.org/downloads/. The website should offer you a download button for the latest Python version. Download the executable installer and run it. Check the box next to `Add Python 3.5 to PATH` and then click `Install Now`.

After installation, open the command prompt and check that the Python version matches the version you installed by executing:

```
python --version
```

4.17.2 About `pip`

pip is a package manage for Python. It makes installing and uninstalling Python packages (such as Django!) very easy. For the rest of the installation, we'll use `pip` to install Python packages from the command line.

To install pip on your machine, go to https://pip.pypa.io/en/latest/installing/, and follow the `Installing with get-pip.py` instructions.

4.17.3 Install `virtualenv` and `virtualenvwrapper`

virtualenv and virtualenvwrapper provide a dedicated environment for each Django project you create. While not mandatory, this is considered a best practice and will save you time in the future when you're ready to deploy your project. Simply type:

```
pip install virtualenvwrapper-win
```

Then create a virtual environment for your project:

```
mkvirtualenv myproject
```

The virtual environment will be activated automatically and you'll see "(myproject)" next to the command prompt to designate that. If you start a new command prompt, you'll need to activate the environment again using:

```
workon myproject
```

4.17.4 Install Django

Django can be installed easily using `pip` within your virtual environment.

In the command prompt, ensure your virtual environment is active, and execute the following command:

```
pip install django
```

This will download and install the latest Django release.

After the installation has completed, you can verify your Django installation by executing `django-admin --version` in the command prompt.

See *Get your database running* for information on database installation with Django.

4.17.5 Common pitfalls

- If `django-admin` only displays the help text no matter what arguments it is given, there is probably a problem with the file association in Windows. Check if there is more than one environment variable set for running Python scripts in `PATH`. This usually occurs when there is more than one Python version installed.

- If you are connecting to the internet behind a proxy, there might be problem in running the command `pip install django`. Set the environment variables for proxy configuration in the command prompt as follows:

```
set http_proxy=http://username:password@proxyserver:proxyport
set https_proxy=https://username:password@proxyserver:proxyport
```

4.18 Writing database migrations

This document explains how to structure and write database migrations for different scenarios you might encounter. For introductory material on migrations, see the topic guide.

4.18.1 Data migrations and multiple databases

When using multiple databases, you may need to figure out whether or not to run a migration against a particular database. For example, you may want to **only** run a migration on a particular database.

In order to do that you can check the database connection's alias inside a `RunPython` operation by looking at the `schema_editor.connection.alias` attribute:

```
from django.db import migrations

def forwards(apps, schema_editor):
    if not schema_editor.connection.alias == 'default':
        return
    # Your migration code goes here

class Migration(migrations.Migration):

    dependencies = [
        # Dependencies to other migrations
    ]

    operations = [
        migrations.RunPython(forwards),
    ]
```

You can also provide hints that will be passed to the *`allow_migrate()`* method of database routers as `**hints`:

myapp/dbrouters.py

```
class MyRouter(object):

    def allow_migrate(self, db, app_label, model_name=None, **hints):
        if 'target_db' in hints:
            return db == hints['target_db']
        return True
```

Then, to leverage this in your migrations, do the following:

```
from django.db import migrations

def forwards(apps, schema_editor):
    # Your migration code goes here
    ...

class Migration(migrations.Migration):
```

```
    dependencies = [
        # Dependencies to other migrations
    ]

    operations = [
        migrations.RunPython(forwards, hints={'target_db': 'default'}),
    ]
```

If your `RunPython` or `RunSQL` operation only affects one model, it's good practice to pass `model_name` as a hint to make it as transparent as possible to the router. This is especially important for reusable and third-party apps.

4.18.2 Migrations that add unique fields

Applying a "plain" migration that adds a unique non-nullable field to a table with existing rows will raise an error because the value used to populate existing rows is generated only once, thus breaking the unique constraint.

Therefore, the following steps should be taken. In this example, we'll add a non-nullable *`UUIDField`* with a default value. Modify the respective field according to your needs.

- Add the field on your model with `default=uuid.uuid4` and `unique=True` arguments (choose an appropriate default for the type of the field you're adding).
- Run the *`makemigrations`* command. This should generate a migration with an `AddField` operation.
- Generate two empty migration files for the same app by running `makemigrations myapp --empty` twice. We've renamed the migration files to give them meaningful names in the examples below.
- Copy the `AddField` operation from the auto-generated migration (the first of the three new files) to the last migration and change `AddField` to `AlterField`. For example:

  0006_remove_uuid_null.py

```
    # -*- coding: utf-8 -*-
    # Generated by Django A.B on YYYY-MM-DD HH:MM
    from __future__ import unicode_literals

    from django.db import migrations, models
    import uuid


    class Migration(migrations.Migration):

        dependencies = [
            ('myapp', '0005_populate_uuid_values'),
        ]

        operations = [
            migrations.AlterField(
                model_name='mymodel',
                name='uuid',
                field=models.UUIDField(default=uuid.uuid4, unique=True),
            ),
        ]
```

- Edit the first migration file. The generated migration class should look similar to this:

  0004_add_uuid_field.py

```
    class Migration(migrations.Migration):

        dependencies = [
            ('myapp', '0003_auto_20150129_1705'),
        ]

        operations = [
            migrations.AddField(
                model_name='mymodel',
                name='uuid',
                field=models.UUIDField(default=uuid.uuid4, unique=True),
            ),
        ]
```

Change `unique=True` to `null=True` – this will create the intermediary null field and defer creating the unique constraint until we've populated unique values on all the rows.

- In the first empty migration file, add a *RunPython* or *RunSQL* operation to generate a unique value (UUID in the example) for each existing row. For example:

  `0005_populate_uuid_values.py`

```
    # -*- coding: utf-8 -*-
    # Generated by Django A.B on YYYY-MM-DD HH:MM
    from __future__ import unicode_literals

    from django.db import migrations, models
    import uuid

    def gen_uuid(apps, schema_editor):
        MyModel = apps.get_model('myapp', 'MyModel')
        for row in MyModel.objects.all():
            row.uuid = uuid.uuid4()
            row.save()

    class Migration(migrations.Migration):

        dependencies = [
            ('myapp', '0004_add_uuid_field'),
        ]

        operations = [
            # omit reverse_code=... if you don't want the migration to be reversible.
            migrations.RunPython(gen_uuid, reverse_code=migrations.RunPython.noop),
        ]
```

- Now you can apply the migrations as usual with the *migrate* command.

  Note there is a race condition if you allow objects to be created while this migration is running. Objects created after the `AddField` and before `RunPython` will have their original `uuid`'s overwritten.

Non-atomic migrations

On databases that support DDL transactions (SQLite and PostgreSQL), migrations will run inside a transaction by default. For use cases such as performing data migrations on large tables, you may want to prevent a migration from running in a transaction by setting the `atomic` attribute to `False`:

```
from django.db import migrations

class Migration(migrations.Migration):
    atomic = False
```

Within such a migration, all operations are run without a transaction. It's possible to execute parts of the migration inside a transaction using *atomic()* or by passing `atomic=True` to `RunPython`.

Here's an example of a non-atomic data migration that updates a large table in smaller batches:

```
import uuid

from django.db import migrations, transaction

def gen_uuid(apps, schema_editor):
    MyModel = apps.get_model('myapp', 'MyModel')
    while MyModel.objects.filter(uuid__isnull=True).exists():
        with transaction.atomic():
            for row in MyModel.objects.filter(uuid__isnull=True)[:1000]:
                row.uuid = uuid.uuid4()
                row.save()

class Migration(migrations.Migration):
    atomic = False

    operations = [
        migrations.RunPython(gen_uuid),
    ]
```

The `atomic` attribute doesn't have an effect on databases that don't support DDL transactions (e.g. MySQL, Oracle).

4.18.3 Controlling the order of migrations

Django determines the order in which migrations should be applied not by the filename of each migration, but by building a graph using two properties on the `Migration` class: `dependencies` and `run_before`.

If you've used the *makemigrations* command you've probably already seen `dependencies` in action because auto-created migrations have this defined as part of their creation process.

The `dependencies` property is declared like this:

```
from django.db import migrations

class Migration(migrations.Migration):

    dependencies = [
        ('myapp', '0123_the_previous_migration'),
    ]
```

Usually this will be enough, but from time to time you may need to ensure that your migration runs *before* other migrations. This is useful, for example, to make third-party apps' migrations run *after* your *AUTH_USER_MODEL* replacement.

To achieve this, place all migrations that should depend on yours in the `run_before` attribute on your `Migration` class:

```
class Migration(migrations.Migration):
    ...
```

```
    run_before = [
        ('third_party_app', '0001_do_awesome'),
    ]
```

Prefer using `dependencies` over `run_before` when possible. You should only use `run_before` if it is undesirable or impractical to specify `dependencies` in the migration which you want to run after the one you are writing.

4.18.4 Migrating data between third-party apps

You can use a data migration to move data from one third-party application to another.

If you plan to remove the old app later, you'll need to set the `dependencies` property based on whether or not the old app is installed. Otherwise, you'll have missing dependencies once you uninstall the old app. Similarly, you'll need to catch `LookupError` in the `apps.get_model()` call that retrieves models from the old app. This approach allows you to deploy your project anywhere without first installing and then uninstalling the old app.

Here's a sample migration:

myapp/migrations/0124_move_old_app_to_new_app.py

```
from django.apps import apps as global_apps
from django.db import migrations

def forwards(apps, schema_editor):
    try:
        OldModel = apps.get_model('old_app', 'OldModel')
    except LookupError:
        # The old app isn't installed.
        return

    NewModel = apps.get_model('new_app', 'NewModel')
    NewModel.objects.bulk_create(
        NewModel(new_attribute=old_object.old_attribute)
        for old_object in OldModel.objects.all()
    )

class Migration(migrations.Migration):
    operations = [
        migrations.RunPython(forwards, migrations.RunPython.noop),
    ]
    dependencies = [
        ('myapp', '0123_the_previous_migration'),
        ('new_app', '0001_initial'),
    ]

    if global_apps.is_installed('old_app'):
        dependencies.append(('old_app', '0001_initial'))
```

Also consider what you want to happen when the migration is unapplied. You could either do nothing (as in the example above) or remove some or all of the data from the new application. Adjust the second argument of the *RunPython* operation accordingly.

4.18.5 Changing an unmanaged model to managed

If you want to change an unmanaged model (*managed=False*) to managed, you must remove `managed=False` and generate a migration before making other schema-related changes to the model, since schema changes that appear

in the migration that contains the operation to change `Meta.managed` may not be applied.

:

The Django community aggregator, where we aggregate content from the global Django community. Many writers in the aggregator write this sort of how-to material.

CHAPTER 5

Django FAQ

5.1 FAQ: General

5.1.1 Why does this project exist?

Django grew from a very practical need: World Online, a newspaper Web operation, is responsible for building intensive Web applications on journalism deadlines. In the fast-paced newsroom, World Online often has only a matter of hours to take a complicated Web application from concept to public launch.

At the same time, the World Online Web developers have consistently been perfectionists when it comes to following best practices of Web development.

In fall 2003, the World Online developers (Adrian Holovaty and Simon Willison) ditched PHP and began using Python to develop its websites. As they built intensive, richly interactive sites such as Lawrence.com, they began to extract a generic Web development framework that let them build Web applications more and more quickly. They tweaked this framework constantly, adding improvements over two years.

In summer 2005, World Online decided to open-source the resulting software, Django. Django would not be possible without a whole host of open-source projects – Apache, Python, and PostgreSQL to name a few – and we're thrilled to be able to give something back to the open-source community.

5.1.2 What does "Django" mean, and how do you pronounce it?

Django is named after Django Reinhardt, a jazz manouche guitarist from the 1930s to early 1950s. To this day, he's considered one of the best guitarists of all time.

Listen to his music. You'll like it.

Django is pronounced **JANG**-oh. Rhymes with FANG-oh. The "D" is silent.

We've also recorded an audio clip of the pronunciation.

5.1.3 Is Django stable?

Yes, it's quite stable. Companies like Disqus, Instagram, Pinterest, and Mozilla have been using Django for many years. Sites built on Django have weathered traffic spikes of over 50 thousand hits per second.

5.1.4 Does Django scale?

Yes. Compared to development time, hardware is cheap, and so Django is designed to take advantage of as much hardware as you can throw at it.

Django uses a "shared-nothing" architecture, which means you can add hardware at any level – database servers, caching servers or Web/application servers.

The framework cleanly separates components such as its database layer and application layer. And it ships with a simple-yet-powerful cache framework.

5.1.5 Who's behind this?

Django was originally developed at World Online, the Web department of a newspaper in Lawrence, Kansas, USA. Django's now run by an international team of volunteers.

5.1.6 Which sites use Django?

DjangoSites.org features a constantly growing list of Django-powered sites.

5.1.7 Django appears to be a MVC framework, but you call the Controller the "view", and the View the "template". How come you don't use the standard names?

Well, the standard names are debatable.

In our interpretation of MVC, the "view" describes the data that gets presented to the user. It's not necessarily *how* the data *looks*, but *which* data is presented. The view describes *which data you see*, not *how you see it.* It's a subtle distinction.

So, in our case, a "view" is the Python callback function for a particular URL, because that callback function describes which data is presented.

Furthermore, it's sensible to separate content from presentation – which is where templates come in. In Django, a "view" describes which data is presented, but a view normally delegates to a template, which describes *how* the data is presented.

Where does the "controller" fit in, then? In Django's case, it's probably the framework itself: the machinery that sends a request to the appropriate view, according to the Django URL configuration.

If you're hungry for acronyms, you might say that Django is a "MTV" framework – that is, "model", "template", and "view." That breakdown makes much more sense.

At the end of the day, of course, it comes down to getting stuff done. And, regardless of how things are named, Django gets stuff done in a way that's most logical to us.

5.1.8 <Framework X> does <feature Y> – why doesn't Django?

We're well aware that there are other awesome Web frameworks out there, and we're not averse to borrowing ideas where appropriate. However, Django was developed precisely because we were unhappy with the status quo, so please be aware that "because <Framework X> does it" is not going to be sufficient reason to add a given feature to Django.

5.1.9 Why did you write all of Django from scratch, instead of using other Python libraries?

When Django was originally written a couple of years ago, Adrian and Simon spent quite a bit of time exploring the various Python Web frameworks available.

In our opinion, none of them were completely up to snuff.

We're picky. You might even call us perfectionists. (With deadlines.)

Over time, we stumbled across open-source libraries that did things we'd already implemented. It was reassuring to see other people solving similar problems in similar ways, but it was too late to integrate outside code: We'd already written, tested and implemented our own framework bits in several production settings – and our own code met our needs delightfully.

In most cases, however, we found that existing frameworks/tools inevitably had some sort of fundamental, fatal flaw that made us squeamish. No tool fit our philosophies 100%.

Like we said: We're picky.

We've documented our philosophies on the design philosophies page.

5.1.10 Is Django a content-management-system (CMS)?

No, Django is not a CMS, or any sort of "turnkey product" in and of itself. It's a Web framework; it's a programming tool that lets you build websites.

For example, it doesn't make much sense to compare Django to something like Drupal, because Django is something you use to *create* things like Drupal.

Of course, Django's automatic admin site is fantastic and timesaving – but the admin site is one module of Django the framework. Furthermore, although Django has special conveniences for building "CMS-y" apps, that doesn't mean it's not just as appropriate for building "non-CMS-y" apps (whatever that means!).

5.1.11 How can I download the Django documentation to read it offline?

The Django docs are available in the `docs` directory of each Django tarball release. These docs are in reST (reStructuredText) format, and each text file corresponds to a Web page on the official Django site.

Because the documentation is stored in revision control, you can browse documentation changes just like you can browse code changes.

Technically, the docs on Django's site are generated from the latest development versions of those reST documents, so the docs on the Django site may offer more information than the docs that come with the latest Django release.

5.1.12 Where can I find Django developers for hire?

Consult our developers for hire page for a list of Django developers who would be happy to help you.

You might also be interested in posting a job to https://djangogigs.com/ . If you want to find Django-capable people in your local area, try https://people.djangoproject.com/ .

5.1.13 How do I cite Django?

It's difficult to give an official citation format, for two reasons: citation formats can vary wildly between publications, and citation standards for software are still a matter of some debate.

For example, APA style, would dictate something like:

```
Django (Version 1.5) [Computer Software]. (2013). Retrieved from https://djangoproject.com.
```

However, the only true guide is what your publisher will accept, so get a copy of those guidelines and fill in the gaps as best you can.

If your referencing style guide requires a publisher name, use "Django Software Foundation".

If you need a publishing location, use "Lawrence, Kansas".

If you need a web address, use https://djangoproject.com.

If you need a name, just use "Django", without any tagline.

If you need a publication date, use the year of release of the version you're referencing (e.g., 2013 for v1.5)

5.2 FAQ: Installation

5.2.1 How do I get started?

1. Download the code.
2. Install Django (read the installation guide).
3. Walk through the tutorial.
4. Check out the rest of the documentation, and ask questions if you run into trouble.

5.2.2 What are Django's prerequisites?

Django requires Python. See the table in the next question for the versions of Python that work with each version of Django. Other Python libraries may be required for some uses, but you'll receive an error about it as they're needed.

For a development environment – if you just want to experiment with Django – you don't need to have a separate Web server installed; Django comes with its own lightweight development server. For a production environment, Django follows the WSGI spec, **PEP 3333**, which means it can run on a variety of server platforms. See Deploying Django for some popular alternatives.

If you want to use Django with a database, which is probably the case, you'll also need a database engine. PostgreSQL is recommended, because we're PostgreSQL fans, and MySQL, SQLite 3, and Oracle are also supported.

5.2.3 What Python version can I use with Django?

Django version	Python versions
1.8	2.7, 3.2 (until the end of 2016), 3.3, 3.4, 3.5
1.9, 1.10	2.7, 3.4, 3.5
1.11	2.7, 3.4, 3.5, 3.6
2.0	3.5+

For each version of Python, only the latest micro release (A.B.C) is officially supported. You can find the latest micro version for each series on the Python download page.

Typically, we will support a Python version up to and including the first Django LTS release whose security support ends after security support for that version of Python ends. For example, Python 3.3 security support ends September 2017 and Django 1.8 LTS security support ends April 2018. Therefore Django 1.8 is the last version to support Python 3.3.

5.2.4 What Python version should I use with Django?

As of Django 1.6, Python 3 support is considered stable and you can safely use it in production. See also Porting to Python 3. However, the community is still in the process of migrating third-party packages and applications to Python 3.

If you're starting a new project, and the dependencies you plan to use work on Python 3, you should use Python 3. If they don't, consider contributing to the porting efforts, or stick to Python 2.

Since newer versions of Python are often faster, have more features, and are better supported, all else being equal, we recommend that you use the latest 2.x.y or 3.x.y release.

You don't lose anything in Django by using an older release, but you don't take advantage of the improvements and optimizations in newer Python releases. Third-party applications for use with Django are, of course, free to set their own version requirements.

5.2.5 Should I use the stable version or development version?

Generally, if you're using code in production, you should be using a stable release. The Django project publishes a full stable release every nine months or so, with bugfix updates in between. These stable releases contain the API that is covered by our backwards compatibility guarantees; if you write code against stable releases, you shouldn't have any problems upgrading when the next official version is released.

5.3 FAQ: Using Django

5.3.1 Why do I get an error about importing DJANGO_SETTINGS_MODULE?

Make sure that:

- The environment variable DJANGO_SETTINGS_MODULE is set to a fully-qualified Python module (i.e. "mysite.settings").
- Said module is on `sys.path` (`import mysite.settings` should work).
- The module doesn't contain syntax errors (of course).

5.3.2 I can't stand your template language. Do I have to use it?

We happen to think our template engine is the best thing since chunky bacon, but we recognize that choosing a template language runs close to religion. There's nothing about Django that requires using the template language, so if you're attached to Jinja2, Mako, or whatever, feel free to use those.

5.3.3 Do I have to use your model/database layer?

Nope. Just like the template system, the model/database layer is decoupled from the rest of the framework.

The one exception is: If you use a different database library, you won't get to use Django's automatically-generated admin site. That app is coupled to the Django database layer.

5.3.4 How do I use image and file fields?

Using a *FileField* or an *ImageField* in a model takes a few steps:

1. In your settings file, you'll need to define *MEDIA_ROOT* as the full path to a directory where you'd like Django to store uploaded files. (For performance, these files are not stored in the database.) Define *MEDIA_URL* as the base public URL of that directory. Make sure that this directory is writable by the Web server's user account.
2. Add the *FileField* or *ImageField* to your model, defining the *upload_to* option to specify a subdirectory of *MEDIA_ROOT* to use for uploaded files.
3. All that will be stored in your database is a path to the file (relative to *MEDIA_ROOT*). You'll most likely want to use the convenience *url* attribute provided by Django. For example, if your *ImageField* is called `mug_shot`, you can get the absolute path to your image in a template with `{{ object.mug_shot.url }}`.

5.3.5 How do I make a variable available to all my templates?

Sometimes your templates just all need the same thing. A common example would be dynamically-generated menus. At first glance, it seems logical to simply add a common dictionary to the template context.

The correct solution is to use a `RequestContext`. Details on how to do this are here: *Using RequestContext*.

5.4 FAQ: Getting Help

5.4.1 How do I do X? Why doesn't Y work? Where can I go to get help?

If this FAQ doesn't contain an answer to your question, you might want to try the *django-users* mailing list. Feel free to ask any question related to installing, using, or debugging Django.

If you prefer IRC, the #django IRC channel on the Freenode IRC network is an active community of helpful individuals who may be able to solve your problem.

5.4.2 Why hasn't my message appeared on django-users?

django-users has a lot of subscribers. This is good for the community, as it means many people are available to contribute answers to questions. Unfortunately, it also means that *django-users* is an attractive target for spammers.

In order to combat the spam problem, when you join the *django-users* mailing list, we manually moderate the first message you send to the list. This means that spammers get caught, but it also means that your first question to the list might take a little longer to get answered. We apologize for any inconvenience that this policy may cause.

5.4.3 Nobody on django-users answered my question! What should I do?

Try making your question more specific, or provide a better example of your problem.

As with most open-source mailing lists, the folks on *django-users* are volunteers. If nobody has answered your question, it may be because nobody knows the answer, it may be because nobody can understand the question, or it

may be that everybody that can help is busy. One thing you might try is to ask the question on IRC – visit the #django IRC channel on the Freenode IRC network.

You might notice we have a second mailing list, called *django-developers* – but please don't email support questions to this mailing list. This list is for discussion of the development of Django itself. Asking a tech support question there is considered quite impolite.

5.4.4 I think I've found a bug! What should I do?

Detailed instructions on how to handle a potential bug can be found in our *Guide to contributing to Django*.

5.4.5 I think I've found a security problem! What should I do?

If you think you've found a security problem with Django, please send a message to security@djangoproject.com. This is a private list only open to long-time, highly trusted Django developers, and its archives are not publicly readable.

Due to the sensitive nature of security issues, we ask that if you think you have found a security problem, *please* don't send a message to one of the public mailing lists. Django has a *policy for handling security issues*; while a defect is outstanding, we would like to minimize any damage that could be inflicted through public knowledge of that defect.

5.5 FAQ: Databases and models

5.5.1 How can I see the raw SQL queries Django is running?

Make sure your Django *DEBUG* setting is set to `True`. Then, just do this:

```
>>> from django.db import connection
>>> connection.queries
[{'sql': 'SELECT polls_polls.id, polls_polls.question, polls_polls.pub_date FROM polls_polls',
'time': '0.002'}]
```

`connection.queries` is only available if *DEBUG* is `True`. It's a list of dictionaries in order of query execution. Each dictionary has the following:

```
``sql`` -- The raw SQL statement
``time`` -- How long the statement took to execute, in seconds.
```

`connection.queries` includes all SQL statements – INSERTs, UPDATES, SELECTs, etc. Each time your app hits the database, the query will be recorded.

If you are using multiple databases, you can use the same interface on each member of the `connections` dictionary:

```
>>> from django.db import connections
>>> connections['my_db_alias'].queries
```

If you need to clear the query list manually at any point in your functions, just call `reset_queries()`, like this:

```
from django.db import reset_queries
reset_queries()
```

5.5.2 Can I use Django with a pre-existing database?

Yes. See Integrating with a legacy database.

5.5.3 If I make changes to a model, how do I update the database?

Take a look at Django's support for *schema migrations*.

If you don't mind clearing data, your project's `manage.py` utility has a *flush* option to reset the database to the state it was in immediately after *migrate* was executed.

5.5.4 Do Django models support multiple-column primary keys?

No. Only single-column primary keys are supported.

But this isn't an issue in practice, because there's nothing stopping you from adding other constraints (using the `unique_together` model option or creating the constraint directly in your database), and enforcing the uniqueness at that level. Single-column primary keys are needed for things such as the admin interface to work; e.g., you need a simple way of being able to specify an object to edit or delete.

5.5.5 Does Django support NoSQL databases?

NoSQL databases are not officially supported by Django itself. There are, however, a number of side project and forks which allow NoSQL functionality in Django, like Django non-rel.

You can also take a look on the wiki page which discusses some alternatives.

5.5.6 How do I add database-specific options to my CREATE TABLE statements, such as specifying MyISAM as the table type?

We try to avoid adding special cases in the Django code to accommodate all the database-specific options such as table type, etc. If you'd like to use any of these options, create a migration with a *RunSQL* operation that contains `ALTER TABLE` statements that do what you want to do.

For example, if you're using MySQL and want your tables to use the MyISAM table type, use the following SQL:

```
ALTER TABLE myapp_mytable ENGINE=MyISAM;
```

5.6 FAQ: The admin

5.6.1 I can't log in. When I enter a valid username and password, it just brings up the login page again, with no error messages.

The login cookie isn't being set correctly, because the domain of the cookie sent out by Django doesn't match the domain in your browser. Try setting the *SESSION_COOKIE_DOMAIN* setting to match your domain. For example, if you're going to "https://www.example.com/admin/" in your browser, set `SESSION_COOKIE_DOMAIN = 'www.example.com'`.

5.6.2 I can't log in. When I enter a valid username and password, it brings up the login page again, with a "Please enter a correct username and password" error.

If you're sure your username and password are correct, make sure your user account has *is_active* and *is_staff* set to True. The admin site only allows access to users with those two fields both set to True.

5.6.3 How do I automatically set a field's value to the user who last edited the object in the admin?

The *ModelAdmin* class provides customization hooks that allow you to transform an object as it saved, using details from the request. By extracting the current user from the request, and customizing the *save_model()* hook, you can update an object to reflect the user that edited it. See *the documentation on ModelAdmin methods* for an example.

5.6.4 How do I limit admin access so that objects can only be edited by the users who created them?

The *ModelAdmin* class also provides customization hooks that allow you to control the visibility and editability of objects in the admin. Using the same trick of extracting the user from the request, the *get_queryset()* and *has_change_permission()* can be used to control the visibility and editability of objects in the admin.

5.6.5 My admin-site CSS and images showed up fine using the development server, but they're not displaying when using mod_wsgi.

See *serving the admin files* in the "How to use Django with mod_wsgi" documentation.

5.6.6 My "list_filter" contains a ManyToManyField, but the filter doesn't display.

Django won't bother displaying the filter for a `ManyToManyField` if there are fewer than two related objects.

For example, if your *list_filter* includes sites, and there's only one site in your database, it won't display a "Site" filter. In that case, filtering by site would be meaningless.

5.6.7 Some objects aren't appearing in the admin.

Inconsistent row counts may be caused by missing foreign key values or a foreign key field incorrectly set to *null=False*. If you have a record with a *ForeignKey* pointing to a non-existent object and that foreign key is included is *list_display*, the record will not be shown in the admin changelist because the Django model is declaring an integrity constraint that is not implemented at the database level.

5.6.8 How can I customize the functionality of the admin interface?

You've got several options. If you want to piggyback on top of an add/change form that Django automatically generates, you can attach arbitrary JavaScript modules to the page via the model's class Admin *js parameter*. That parameter is a list of URLs, as strings, pointing to JavaScript modules that will be included within the admin form via a `<script>` tag.

If you want more flexibility than simply tweaking the auto-generated forms, feel free to write custom views for the admin. The admin is powered by Django itself, and you can write custom views that hook into the authentication system, check permissions and do whatever else they need to do.

If you want to customize the look-and-feel of the admin interface, read the next question.

5.6.9 The dynamically-generated admin site is ugly! How can I change it?

We like it, but if you don't agree, you can modify the admin site's presentation by editing the CSS stylesheet and/or associated image files. The site is built using semantic HTML and plenty of CSS hooks, so any changes you'd like to make should be possible by editing the stylesheet.

5.6.10 What browsers are supported for using the admin?

The admin provides a fully-functional experience to YUI's A-grade browsers, with the notable exception of IE6, which is not supported.

There *may* be minor stylistic differences between supported browsers—for example, some browsers may not support rounded corners. These are considered acceptable variations in rendering.

5.7 FAQ: Contributing code

5.7.1 How can I get started contributing code to Django?

Thanks for asking! We've written an entire document devoted to this question. It's titled Contributing to Django.

5.7.2 I submitted a bug fix in the ticket system several weeks ago. Why are you ignoring my patch?

Don't worry: We're not ignoring you!

It's important to understand there is a difference between "a ticket is being ignored" and "a ticket has not been attended to yet." Django's ticket system contains hundreds of open tickets, of various degrees of impact on end-user functionality, and Django's developers have to review and prioritize.

On top of that: the people who work on Django are all volunteers. As a result, the amount of time that we have to work on the framework is limited and will vary from week to week depending on our spare time. If we're busy, we may not be able to spend as much time on Django as we might want.

The best way to make sure tickets do not get hung up on the way to checkin is to make it dead easy, even for someone who may not be intimately familiar with that area of the code, to understand the problem and verify the fix:

- Are there clear instructions on how to reproduce the bug? If this touches a dependency (such as Pillow), a contrib module, or a specific database, are those instructions clear enough even for someone not familiar with it?
- If there are several patches attached to the ticket, is it clear what each one does, which ones can be ignored and which matter?
- Does the patch include a unit test? If not, is there a very clear explanation why not? A test expresses succinctly what the problem is, and shows that the patch actually fixes it.

If your patch stands no chance of inclusion in Django, we won't ignore it – we'll just close the ticket. So if your ticket is still open, it doesn't mean we're ignoring you; it just means we haven't had time to look at it yet.

5.7.3 When and how might I remind the core team of a patch I care about?

A polite, well-timed message to the mailing list is one way to get attention. To determine the right time, you need to keep an eye on the schedule. If you post your message when the core developers are trying to hit a feature deadline or

manage a planning phase, you're not going to get the sort of attention you require. However, if you draw attention to a ticket when the core developers are paying particular attention to bugs – just before a bug fixing sprint, or in the lead up to a beta release for example – you're much more likely to get a productive response.

Gentle IRC reminders can also work – again, strategically timed if possible. During a bug sprint would be a very good time, for example.

Another way to get traction is to pull several related tickets together. When the core developers sit down to fix a bug in an area they haven't touched for a while, it can take a few minutes to remember all the fine details of how that area of code works. If you collect several minor bug fixes together into a similarly themed group, you make an attractive target, as the cost of coming up to speed on an area of code can be spread over multiple tickets.

Please refrain from emailing core developers personally, or repeatedly raising the same issue over and over. This sort of behavior will not gain you any additional attention – certainly not the attention that you need in order to get your pet bug addressed.

5.7.4 But I've reminded you several times and you keep ignoring my patch!

Seriously - we're not ignoring you. If your patch stands no chance of inclusion in Django, we'll close the ticket. For all the other tickets, we need to prioritize our efforts, which means that some tickets will be addressed before others.

One of the criteria that is used to prioritize bug fixes is the number of people that will likely be affected by a given bug. Bugs that have the potential to affect many people will generally get priority over those that are edge cases.

Another reason that bugs might be ignored for while is if the bug is a symptom of a larger problem. While we can spend time writing, testing and applying lots of little patches, sometimes the right solution is to rebuild. If a rebuild or refactor of a particular component has been proposed or is underway, you may find that bugs affecting that component will not get as much attention. Again, this is just a matter of prioritizing scarce resources. By concentrating on the rebuild, we can close all the little bugs at once, and hopefully prevent other little bugs from appearing in the future.

Whatever the reason, please keep in mind that while you may hit a particular bug regularly, it doesn't necessarily follow that every single Django user will hit the same bug. Different users use Django in different ways, stressing different parts of the code under different conditions. When we evaluate the relative priorities, we are generally trying to consider the needs of the entire community, not just the severity for one particular user. This doesn't mean that we think your problem is unimportant – just that in the limited time we have available, we will always err on the side of making 10 people happy rather than making 1 person happy.

5.8 Troubleshooting

This page contains some advice about errors and problems commonly encountered during the development of Django applications.

5.8.1 Problems running django-admin

"command not found: django-admin"

django-admin should be on your system path if you installed Django via `python setup.py`. If it's not on your path, you can find it in `site-packages/django/bin`, where `site-packages` is a directory within your Python installation. Consider symlinking to django-admin from some place on your path, such as `/usr/local/bin`.

If `django-admin` doesn't work but `django-admin.py` does, you're probably using a version of Django that doesn't match the version of this documentation. `django-admin` is new in Django 1.7.

Mac OS X permissions

If you're using Mac OS X, you may see the message "permission denied" when you try to run `django-admin`. This is because, on Unix-based systems like OS X, a file must be marked as "executable" before it can be run as a program. To do this, open Terminal.app and navigate (using the `cd` command) to the directory where django-admin is installed, then run the command `sudo chmod +x django-admin`.

5.8.2 Miscellaneous

I'm getting a `UnicodeDecodeError`. What am I doing wrong?

This class of errors happen when a bytestring containing non-ASCII sequences is transformed into a Unicode string and the specified encoding is incorrect. The output generally looks like this:

```
UnicodeDecodeError: 'ascii' codec can't decode byte 0x?? in position ?:
ordinal not in range(128)
```

The resolution mostly depends on the context, however here are two common pitfalls producing this error:

- Your system locale may be a default ASCII locale, like the "C" locale on UNIX-like systems (can be checked by the `locale` command). If it's the case, please refer to your system documentation to learn how you can change this to a UTF-8 locale.
- You created raw bytestrings, which is easy to do on Python 2:

```
my_string = 'café'
```

  Either use the `u''` prefix or even better, add the `from __future__ import unicode_literals` line at the top of your file so that your code will be compatible with Python 3.2 which doesn't support the `u''` prefix.

Related resources:

- Unicode in Django
- https://wiki.python.org/moin/UnicodeDecodeError

CHAPTER 6

API Reference

6.1 Applications

Django contains a registry of installed applications that stores configuration and provides introspection. It also maintains a list of available models.

This registry is simply called *apps* and it's available in *django.apps*:

```
>>> from django.apps import apps
>>> apps.get_app_config('admin').verbose_name
'Admin'
```

6.1.1 Projects and applications

The term **project** describes a Django web application. The project Python package is defined primarily by a settings module, but it usually contains other things. For example, when you run `django-admin startproject mysite` you'll get a `mysite` project directory that contains a `mysite` Python package with `settings.py`, `urls.py`, and `wsgi.py`. The project package is often extended to include things like fixtures, CSS, and templates which aren't tied to a particular application.

A **project's root directory** (the one that contains `manage.py`) is usually the container for all of a project's applications which aren't installed separately.

The term **application** describes a Python package that provides some set of features. Applications may be reused in various projects.

Applications include some combination of models, views, templates, template tags, static files, URLs, middleware, etc. They're generally wired into projects with the *INSTALLED_APPS* setting and optionally with other mechanisms such as URLconfs, the *MIDDLEWARE* setting, or template inheritance.

It is important to understand that a Django application is just a set of code that interacts with various parts of the framework. There's no such thing as an `Application` object. However, there's a few places where Django needs to interact with installed applications, mainly for configuration and also for introspection. That's why the application registry maintains metadata in an *AppConfig* instance for each installed application.

There's no restriction that a project package can't also be considered an application and have models, etc. (which would require adding it to *INSTALLED_APPS*).

6.1.2 Configuring applications

To configure an application, subclass *AppConfig* and put the dotted path to that subclass in *INSTALLED_APPS*.

When *INSTALLED_APPS* simply contains the dotted path to an application module, Django checks for a `default_app_config` variable in that module.

If it's defined, it's the dotted path to the *AppConfig* subclass for that application.

If there is no `default_app_config`, Django uses the base *AppConfig* class.

`default_app_config` allows applications that predate Django 1.7 such as `django.contrib.admin` to opt-in to *AppConfig* features without requiring users to update their *INSTALLED_APPS*.

New applications should avoid `default_app_config`. Instead they should require the dotted path to the appropriate *AppConfig* subclass to be configured explicitly in *INSTALLED_APPS*.

For application authors

If you're creating a pluggable app called "Rock 'n' roll", here's how you would provide a proper name for the admin:

```
# rock_n_roll/apps.py

from django.apps import AppConfig

class RockNRollConfig(AppConfig):
    name = 'rock_n_roll'
    verbose_name = "Rock 'n' roll"
```

You can make your application load this *AppConfig* subclass by default as follows:

```
# rock_n_roll/__init__.py

default_app_config = 'rock_n_roll.apps.RockNRollConfig'
```

That will cause `RockNRollConfig` to be used when *INSTALLED_APPS* just contains `'rock_n_roll'`. This allows you to make use of *AppConfig* features without requiring your users to update their *INSTALLED_APPS* setting. Besides this use case, it's best to avoid using `default_app_config` and instead specify the app config class in *INSTALLED_APPS* as described next.

Of course, you can also tell your users to put `'rock_n_roll.apps.RockNRollConfig'` in their *INSTALLED_APPS* setting. You can even provide several different *AppConfig* subclasses with different behaviors and allow your users to choose one via their *INSTALLED_APPS* setting.

The recommended convention is to put the configuration class in a submodule of the application called `apps`. However, this isn't enforced by Django.

You must include the *name* attribute for Django to determine which application this configuration applies to. You can define any attributes documented in the *AppConfig* API reference.

: If your code imports the application registry in an application's `__init__.py`, the name `apps` will clash with the `apps` submodule. The best practice is to move that code to a submodule and import it. A workaround is to import the registry under a different name:

```
from django.apps import apps as django_apps
```

For application users

If you're using "Rock 'n' roll" in a project called `anthology`, but you want it to show up as "Jazz Manouche" instead, you can provide your own configuration:

```
# anthology/apps.py

from rock_n_roll.apps import RockNRollConfig

class JazzManoucheConfig(RockNRollConfig):
    verbose_name = "Jazz Manouche"

# anthology/settings.py

INSTALLED_APPS = [
    'anthology.apps.JazzManoucheConfig',
    # ...
]
```

Again, defining project-specific configuration classes in a submodule called `apps` is a convention, not a requirement.

6.1.3 Application configuration

class AppConfig

Application configuration objects store metadata for an application. Some attributes can be configured in *AppConfig* subclasses. Others are set by Django and read-only.

Configurable attributes

`AppConfig.`**name**

Full Python path to the application, e.g. `'django.contrib.admin'`.

This attribute defines which application the configuration applies to. It must be set in all *AppConfig* subclasses.

It must be unique across a Django project.

`AppConfig.`**label**

Short name for the application, e.g. `'admin'`

This attribute allows relabeling an application when two applications have conflicting labels. It defaults to the last component of `name`. It should be a valid Python identifier.

It must be unique across a Django project.

`AppConfig.`**verbose_name**

Human-readable name for the application, e.g. "Administration".

This attribute defaults to `label.title()`.

`AppConfig.`**path**

Filesystem path to the application directory, e.g. `'/usr/lib/python3.4/dist-packages/django/contrib/admin`

In most cases, Django can automatically detect and set this, but you can also provide an explicit override as a class attribute on your *AppConfig* subclass. In a few situations this is required; for instance if the app package is a *namespace package* with multiple paths.

Read-only attributes

`AppConfig.`**module**

Root module for the application, e.g. `<module 'django.contrib.admin' from 'django/contrib/admin/__init__.pyc'>`.

AppConfig.**models_module**
Module containing the models, e.g. <module 'django.contrib.admin.models' from 'django/contrib/admin/models.pyc'>.

It may be None if the application doesn't contain a models module. Note that the database related signals such as *pre_migrate* and *post_migrate* are only emitted for applications that have a models module.

Methods

AppConfig.**get_models**()
Returns an iterable of *Model* classes for this application.

AppConfig.**get_model**(*model_name*)
Returns the *Model* with the given model_name. Raises LookupError if no such model exists in this application. model_name is case-insensitive.

AppConfig.**ready**()
Subclasses can override this method to perform initialization tasks such as registering signals. It is called as soon as the registry is fully populated.

Although you can't import models at the module-level where *AppConfig* classes are defined, you can import them in ready(), using either an import statement or *get_model()*.

If you're registering *model signals*, you can refer to the sender by its string label instead of using the model class itself.

Example:

```
from django.db.models.signals import pre_save

def ready(self):
    # importing model classes
    from .models import MyModel  # or...
    MyModel = self.get_model('MyModel')

    # registering signals with the model's string label
    pre_save.connect(receiver, sender='app_label.MyModel')
```

: Although you can access model classes as described above, avoid interacting with the database in your *ready()* implementation. This includes model methods that execute queries (*save()*, *delete()*, manager methods etc.), and also raw SQL queries via django.db.connection. Your *ready()* method will run during startup of every management command. For example, even though the test database configuration is separate from the production settings, manage.py test would still execute some queries against your **production** database!

: In the usual initialization process, the ready method is only called once by Django. But in some corner cases, particularly in tests which are fiddling with installed applications, ready might be called more than once. In that case, either write idempotent methods, or put a flag on your AppConfig classes to prevent re-running code which should be executed exactly one time.

Namespace packages as apps (Python 3.3+)

Python versions 3.3 and later support Python packages without an __init__.py file. These packages are known as "namespace packages" and may be spread across multiple directories at different locations on sys.path (see **PEP**

420).

Django applications require a single base filesystem path where Django (depending on configuration) will search for templates, static assets, etc. Thus, namespace packages may only be Django applications if one of the following is true:

1. The namespace package actually has only a single location (i.e. is not spread across more than one directory.)
2. The *AppConfig* class used to configure the application has a *path* class attribute, which is the absolute directory path Django will use as the single base path for the application.

If neither of these conditions is met, Django will raise *ImproperlyConfigured*.

6.1.4 Application registry

apps

The application registry provides the following public API. Methods that aren't listed below are considered private and may change without notice.

apps.**ready**

Boolean attribute that is set to `True` after the registry is fully populated and all *AppConfig.ready()* methods are called.

apps.**get_app_configs**()

Returns an iterable of *AppConfig* instances.

apps.**get_app_config**(*app_label*)

Returns an *AppConfig* for the application with the given `app_label`. Raises `LookupError` if no such application exists.

apps.**is_installed**(*app_name*)

Checks whether an application with the given name exists in the registry. `app_name` is the full name of the app, e.g. `'django.contrib.admin'`.

apps.**get_model**(*app_label*, *model_name*)

Returns the *Model* with the given `app_label` and `model_name`. As a shortcut, this method also accepts a single argument in the form `app_label.model_name`. `model_name` is case-insensitive.

Raises `LookupError` if no such application or model exists. Raises `ValueError` when called with a single argument that doesn't contain exactly one dot.

6.1.5 Initialization process

How applications are loaded

When Django starts, *django.setup()* is responsible for populating the application registry.

setup(*set_prefix=True*)

Configures Django by:

- Loading the settings.
- Setting up logging.
- If `set_prefix` is True, setting the URL resolver script prefix to *FORCE_SCRIPT_NAME* if defined, or `/` otherwise.
- Initializing the application registry.

The ability to set the URL resolver script prefix is new.

This function is called automatically:

- When running an HTTP server via Django's WSGI support.
- When invoking a management command.

It must be called explicitly in other cases, for instance in plain Python scripts.

The application registry is initialized in three stages. At each stage, Django processes all applications in the order of *INSTALLED_APPS*.

1. First Django imports each item in *INSTALLED_APPS*.

   If it's an application configuration class, Django imports the root package of the application, defined by its *name* attribute. If it's a Python package, Django creates a default application configuration.

   At this stage, your code shouldn't import any models!

   In other words, your applications' root packages and the modules that define your application configuration classes shouldn't import any models, even indirectly.

   Strictly speaking, Django allows importing models once their application configuration is loaded. However, in order to avoid needless constraints on the order of *INSTALLED_APPS*, it's strongly recommended not import any models at this stage.

   Once this stage completes, APIs that operate on application configurations such as *get_app_config()* become usable.

2. Then Django attempts to import the `models` submodule of each application, if there is one.

   You must define or import all models in your application's `models.py` or `models/__init__.py`. Otherwise, the application registry may not be fully populated at this point, which could cause the ORM to malfunction.

   Once this stage completes, APIs that operate on models such as *get_model()* become usable.

3. Finally Django runs the *ready()* method of each application configuration.

Troubleshooting

Here are some common problems that you may encounter during initialization:

- *AppRegistryNotReady*: This happens when importing an application configuration or a models module triggers code that depends on the app registry.

  For example, *ugettext()* uses the app registry to look up translation catalogs in applications. To translate at import time, you need *ugettext_lazy()* instead. (Using *ugettext()* would be a bug, because the translation would happen at import time, rather than at each request depending on the active language.)

  Executing database queries with the ORM at import time in models modules will also trigger this exception. The ORM cannot function properly until all models are available.

  Another common culprit is *django.contrib.auth.get_user_model()*. Use the *AUTH_USER_MODEL* setting to reference the User model at import time.

  This exception also happens if you forget to call *django.setup()* in a standalone Python script.

- `ImportError: cannot import name ...` This happens if the import sequence ends up in a loop.

  To eliminate such problems, you should minimize dependencies between your models modules and do as little work as possible at import time. To avoid executing code at import time, you can move it into a function and

cache its results. The code will be executed when you first need its results. This concept is known as "lazy evaluation".

- `django.contrib.admin` automatically performs autodiscovery of `admin` modules in installed applications. To prevent it, change your *INSTALLED_APPS* to contain `'django.contrib.admin.apps.SimpleAdminConfig'` instead of `'django.contrib.admin'`.

6.2 System check framework

The system check framework is a set of static checks for validating Django projects. It detects common problems and provides hints for how to fix them. The framework is extensible so you can easily add your own checks.

For details on how to add your own checks and integrate them with Django's system checks, see the System check topic guide.

6.2.1 API Reference

CheckMessage

class CheckMessage(*level*, *msg*, *hint=None*, *obj=None*, *id=None*)

The warnings and errors raised by system checks must be instances of `CheckMessage`. An instance encapsulates a single reportable error or warning. It also provides context and hints applicable to the message, and a unique identifier that is used for filtering purposes.

Constructor arguments are:

level The severity of the message. Use one of the predefined values: `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`. If the level is greater or equal to `ERROR`, then Django will prevent management commands from executing. Messages with level lower than `ERROR` (i.e. warnings) are reported to the console, but can be silenced.

msg A short (less than 80 characters) string describing the problem. The string should *not* contain newlines.

hint A single-line string providing a hint for fixing the problem. If no hint can be provided, or the hint is self-evident from the error message, the hint can be omitted, or a value of `None` can be used.

obj Optional. An object providing context for the message (for example, the model where the problem was discovered). The object should be a model, field, or manager or any other object that defines `__str__` method (on Python 2 you need to define `__unicode__` method). The method is used while reporting all messages and its result precedes the message.

id Optional string. A unique identifier for the issue. Identifiers should follow the pattern `applabel.X001`, where `X` is one of the letters `CEWID`, indicating the message severity (`C` for criticals, `E` for errors and so). The number can be allocated by the application, but should be unique within that application.

There are subclasses to make creating messages with common levels easier. When using them you can omit the `level` argument because it is implied by the class name.

class Debug(*msg*, *hint=None*, *obj=None*, *id=None*)

class Info(*msg*, *hint=None*, *obj=None*, *id=None*)

class Warning(*msg*, *hint=None obj=None*, *id=None*)

class Error(*msg*, *hint=None*, *obj=None*, *id=None*)

class Critical(*msg*, *hint=None*, *obj=None*, *id=None*)

6.2.2 Builtin checks

Builtin tags

Django's system checks are organized using the following tags:

- `models`: Checks governing model, field and manager definitions.
- `signals`: Checks on signal declarations and handler registrations.
- `admin`: Checks of any admin site declarations.
- `compatibility`: Flagging potential problems with version upgrades.
- `security`: Checks security related configuration.
- `templates`: Checks template related configuration.
- `caches`: Checks cache related configuration.
- `urls`: Checks URL configuration.
- `database`: Checks database-related configuration issues. Database checks are not run by default because they do more than static code analysis as regular checks do. They are only run by the *`migrate`* command or if you specify the `database` tag when calling the *`check`* command.

The `database` tag was added.

Some checks may be registered with multiple tags.

Core system checks

Models

- **models.E001**: `<swappable>` is not of the form `app_label.app_name`.
- **models.E002**: `<SETTING>` references `<model>`, which has not been installed, or is abstract.
- **models.E003**: The model has two many-to-many relations through the intermediate model `<app_label>.<model>`.
- **models.E004**: `id` can only be used as a field name if the field also sets `primary_key=True`.
- **models.E005**: The field `<field name>` from parent model `<model>` clashes with the field `<field name>` from parent model `<model>`.
- **models.E006**: The field clashes with the field `<field name>` from model `<model>`.
- **models.E007**: Field `<field name>` has column name `<column name>` that is used by another field.
- **models.E008**: `index_together` must be a list or tuple.
- **models.E009**: All `index_together` elements must be lists or tuples.
- **models.E010**: `unique_together` must be a list or tuple.
- **models.E011**: All `unique_together` elements must be lists or tuples.
- **models.E012**: `index_together/unique_together` refers to the non-existent field `<field name>`.
- **models.E013**: `index_together/unique_together` refers to a `ManyToManyField` `<field name>`, but `ManyToManyFields` are not supported for that option.
- **models.E014**: `ordering` must be a tuple or list (even if you want to order by only one field).

- **models.E015**: `ordering` refers to the non-existent field `<field name>`.
- **models.E016**: `index_together/unique_together` refers to field `<field_name>` which is not local to model `<model>`.
- **models.E017**: Proxy model `<model>` contains model fields.
- **models.E018**: Autogenerated column name too long for field `<field>`. Maximum length is `<maximum length>` for database `<alias>`.
- **models.E019**: Autogenerated column name too long for M2M field `<M2M field>`. Maximum length is `<maximum length>` for database `<alias>`.
- **models.E020**: The `<model>.check()` class method is currently overridden.
- **models.E021**: `ordering` and `order_with_respect_to` cannot be used together.
- **models.E022**: `<function>` contains a lazy reference to `<app label>.<model>`, but app `<app label>` isn't installed or doesn't provide model `<model>`.

Fields

- **fields.E001**: Field names must not end with an underscore.
- **fields.E002**: Field names must not contain `"__"`.
- **fields.E003**: `pk` is a reserved word that cannot be used as a field name.
- **fields.E004**: `choices` must be an iterable (e.g., a list or tuple).
- **fields.E005**: `choices` must be an iterable returning `(actual value, human readable name)` tuples.
- **fields.E006**: `db_index` must be `None`, `True` or `False`.
- **fields.E007**: Primary keys must not have `null=True`.
- **fields.E100**: `AutoField`s must set primary_key=True.
- **fields.E110**: `BooleanField`s do not accept null values.
- **fields.E120**: `CharField`s must define a `max_length` attribute.
- **fields.E121**: `max_length` must be a positive integer.
- **fields.W122**: `max_length` is ignored when used with `IntegerField`.
- **fields.E130**: `DecimalField`s must define a `decimal_places` attribute.
- **fields.E131**: `decimal_places` must be a non-negative integer.
- **fields.E132**: `DecimalField`s must define a `max_digits` attribute.
- **fields.E133**: `max_digits` must be a non-negative integer.
- **fields.E134**: `max_digits` must be greater or equal to `decimal_places`.
- **fields.E140**: `FilePathField`s must have either `allow_files` or `allow_folders` set to True.
- **fields.E150**: `GenericIPAddressField`s cannot accept blank values if null values are not allowed, as blank values are stored as nulls.
- **fields.E160**: The options `auto_now`, `auto_now_add`, and `default` are mutually exclusive. Only one of these options may be present.
- **fields.W161**: Fixed default value provided.

- **fields.E900**: `IPAddressField` has been removed except for support in historical migrations.
- **fields.W900**: `IPAddressField` has been deprecated. Support for it (except in historical migrations) will be removed in Django 1.9. *This check appeared in Django 1.7 and 1.8.*
- **fields.W901**: `CommaSeparatedIntegerField` has been deprecated. Support for it (except in historical migrations) will be removed in Django 2.0.

File Fields

- **fields.E200**: `unique` is not a valid argument for a `FileField`.
- **fields.E201**: `primary_key` is not a valid argument for a `FileField`.
- **fields.E210**: Cannot use `ImageField` because Pillow is not installed.

Related Fields

- **fields.E300**: Field defines a relation with model `<model>`, which is either not installed, or is abstract.
- **fields.E301**: Field defines a relation with the model `<model>` which has been swapped out.
- **fields.E302**: Accessor for field `<field name>` clashes with field `<field name>`.
- **fields.E303**: Reverse query name for field `<field name>` clashes with field `<field name>`.
- **fields.E304**: Field name `<field name>` clashes with accessor for `<field name>`.
- **fields.E305**: Field name `<field name>` clashes with reverse query name for `<field name>`.
- **fields.E306**: Related name must be a valid Python identifier or end with a `'+'`.
- **fields.E307**: The field `<app label>.<model>.<field name>` was declared with a lazy reference to `<app label>.<model>`, but app `<app label>` isn't installed or doesn't provide model `<model>`.
- **fields.E310**: No subset of the fields `<field1>`, `<field2>`, ... on model `<model>` is unique. Add `unique=True` on any of those fields or add at least a subset of them to a unique_together constraint.
- **fields.E311**: `<model>` must set `unique=True` because it is referenced by a `ForeignKey`.
- **fields.E320**: Field specifies `on_delete=SET_NULL`, but cannot be null.
- **fields.E321**: The field specifies `on_delete=SET_DEFAULT`, but has no default value.
- **fields.E330**: `ManyToManyField`s cannot be unique.
- **fields.E331**: Field specifies a many-to-many relation through model `<model>`, which has not been installed.
- **fields.E332**: Many-to-many fields with intermediate tables must not be symmetrical.
- **fields.E333**: The model is used as an intermediate model by `<model>`, but it has more than two foreign keys to `<model>`, which is ambiguous. You must specify which two foreign keys Django should use via the `through_fields` keyword argument.
- **fields.E334**: The model is used as an intermediate model by `<model>`, but it has more than one foreign key from `<model>`, which is ambiguous. You must specify which foreign key Django should use via the `through_fields` keyword argument.
- **fields.E335**: The model is used as an intermediate model by `<model>`, but it has more than one foreign key to `<model>`, which is ambiguous. You must specify which foreign key Django should use via the `through_fields` keyword argument.

- **fields.E336**: The model is used as an intermediary model by `<model>`, but it does not have foreign key to `<model>` or `<model>`.
- **fields.E337**: Field specifies `through_fields` but does not provide the names of the two link fields that should be used for the relation through `<model>`.
- **fields.E338**: The intermediary model `<through model>` has no field `<field name>`.
- **fields.E339**: `<model>.<field name>` is not a foreign key to `<model>`.
- **fields.W340**: `null` has no effect on `ManyToManyField`.
- **fields.W341**: `ManyToManyField` does not support `validators`.
- **fields.W342**: Setting `unique=True` on a `ForeignKey` has the same effect as using a `OneToOneField`.

Signals

- **signals.E001**: `<handler>` was connected to the `<signal>` signal with a lazy reference to the sender `<app label>.<model>`, but app `<app label>` isn't installed or doesn't provide model `<model>`.

Backwards Compatibility

The following checks are performed to warn the user of any potential problems that might occur as a result of a version upgrade.

- **1_6.W001**: Some project unit tests may not execute as expected. *This check was removed in Django 1.8 due to false positives.*
- **1_6.W002**: `BooleanField` does not have a default value. *This check was removed in Django 1.8 due to false positives.*
- **1_7.W001**: Django 1.7 changed the global defaults for the `MIDDLEWARE_CLASSES`. `django.contrib.sessions.middleware.SessionMiddleware`, `django.contrib.auth.middleware.AuthenticationMiddleware`, and `django.contrib.messages.middleware.MessageMiddleware` were removed from the defaults. If your project needs these middleware then you should configure this setting. *This check was removed in Django 1.9.*
- **1_8.W001**: The standalone `TEMPLATE_*` settings were deprecated in Django 1.8 and the *`TEMPLATES`* dictionary takes precedence. You must put the values of the following settings into your defaults `TEMPLATES` dict: `TEMPLATE_DIRS`, `TEMPLATE_CONTEXT_PROCESSORS`, `TEMPLATE_DEBUG`, `TEMPLATE_LOADERS`, `TEMPLATE_STRING_IF_INVALID`.
- **1_10.W001**: The `MIDDLEWARE_CLASSES` setting is deprecated in Django 1.10 and the *`MIDDLEWARE`* setting takes precedence. Since you've set `MIDDLEWARE`, the value of `MIDDLEWARE_CLASSES` is ignored.

Admin

Admin checks are all performed as part of the `admin` tag.

The following checks are performed on any *`ModelAdmin`* (or subclass) that is registered with the admin site:

- **admin.E001**: The value of `raw_id_fields` must be a list or tuple.
- **admin.E002**: The value of `raw_id_fields[n]` refers to `<field name>`, which is not an attribute of `<model>`.
- **admin.E003**: The value of `raw_id_fields[n]` must be a foreign key or a many-to-many field.

- **admin.E004**: The value of `fields` must be a list or tuple.
- **admin.E005**: Both `fieldsets` and `fields` are specified.
- **admin.E006**: The value of `fields` contains duplicate field(s).
- **admin.E007**: The value of `fieldsets` must be a list or tuple.
- **admin.E008**: The value of `fieldsets[n]` must be a list or tuple.
- **admin.E009**: The value of `fieldsets[n]` must be of length 2.
- **admin.E010**: The value of `fieldsets[n][1]` must be a dictionary.
- **admin.E011**: The value of `fieldsets[n][1]` must contain the key `fields`.
- **admin.E012**: There are duplicate field(s) in `fieldsets[n][1]`.
- **admin.E013**: `fields[n]/fieldsets[n][m]` cannot include the `ManyToManyField <field name>`, because that field manually specifies a relationship model.
- **admin.E014**: The value of `exclude` must be a list or tuple.
- **admin.E015**: The value of `exclude` contains duplicate field(s).
- **admin.E016**: The value of `form` must inherit from `BaseModelForm`.
- **admin.E017**: The value of `filter_vertical` must be a list or tuple.
- **admin.E018**: The value of `filter_horizontal` must be a list or tuple.
- **admin.E019**: The value of `filter_vertical[n]/filter_vertical[n]` refers to `<field name>`, which is not an attribute of `<model>`.
- **admin.E020**: The value of `filter_vertical[n]/filter_vertical[n]` must be a many-to-many field.
- **admin.E021**: The value of `radio_fields` must be a dictionary.
- **admin.E022**: The value of `radio_fields` refers to `<field name>`, which is not an attribute of `<model>`.
- **admin.E023**: The value of `radio_fields` refers to `<field name>`, which is not a `ForeignKey`, and does not have a `choices` definition.
- **admin.E024**: The value of `radio_fields[<field name>]` must be either `admin.HORIZONTAL` or `admin.VERTICAL`.
- **admin.E025**: The value of `view_on_site` must be either a callable or a boolean value.
- **admin.E026**: The value of `prepopulated_fields` must be a dictionary.
- **admin.E027**: The value of `prepopulated_fields` refers to `<field name>`, which is not an attribute of `<model>`.
- **admin.E028**: The value of `prepopulated_fields` refers to `<field name>`, which must not be a `DateTimeField`, a `ForeignKey`, or a `ManyToManyField` field.
- **admin.E029**: The value of `prepopulated_fields[<field name>]` must be a list or tuple.
- **admin.E030**: The value of `prepopulated_fields` refers to `<field name>`, which is not an attribute of `<model>`.
- **admin.E031**: The value of `ordering` must be a list or tuple.
- **admin.E032**: The value of `ordering` has the random ordering marker `?`, but contains other fields as well.
- **admin.E033**: The value of `ordering` refers to `<field name>`, which is not an attribute of `<model>`.
- **admin.E034**: The value of `readonly_fields` must be a list or tuple.

- **admin.E035**: The value of `readonly_fields[n]` is not a callable, an attribute of `<ModelAdmin class>`, or an attribute of `<model>`.

ModelAdmin

The following checks are performed on any *ModelAdmin* that is registered with the admin site:

- **admin.E101**: The value of `save_as` must be a boolean.
- **admin.E102**: The value of `save_on_top` must be a boolean.
- **admin.E103**: The value of `inlines` must be a list or tuple.
- **admin.E104**: `<InlineModelAdmin class>` must inherit from `BaseModelAdmin`.
- **admin.E105**: `<InlineModelAdmin class>` must have a `model` attribute.
- **admin.E106**: The value of `<InlineModelAdmin class>.model` must be a `Model`.
- **admin.E107**: The value of `list_display` must be a list or tuple.
- **admin.E108**: The value of `list_display[n]` refers to `<label>`, which is not a callable, an attribute of `<ModelAdmin class>`, or an attribute or method on `<model>`.
- **admin.E109**: The value of `list_display[n]` must not be a `ManyToManyField` field.
- **admin.E110**: The value of `list_display_links` must be a list, a tuple, or `None`.
- **admin.E111**: The value of `list_display_links[n]` refers to `<label>`, which is not defined in `list_display`.
- **admin.E112**: The value of `list_filter` must be a list or tuple.
- **admin.E113**: The value of `list_filter[n]` must inherit from `ListFilter`.
- **admin.E114**: The value of `list_filter[n]` must not inherit from `FieldListFilter`.
- **admin.E115**: The value of `list_filter[n][1]` must inherit from `FieldListFilter`.
- **admin.E116**: The value of `list_filter[n]` refers to `<label>`, which does not refer to a Field.
- **admin.E117**: The value of `list_select_related` must be a boolean, tuple or list.
- **admin.E118**: The value of `list_per_page` must be an integer.
- **admin.E119**: The value of `list_max_show_all` must be an integer.
- **admin.E120**: The value of `list_editable` must be a list or tuple.
- **admin.E121**: The value of `list_editable[n]` refers to `<label>`, which is not an attribute of `<model>`.
- **admin.E122**: The value of `list_editable[n]` refers to `<label>`, which is not contained in `list_display`.
- **admin.E123**: The value of `list_editable[n]` cannot be in both `list_editable` and `list_display_links`.
- **admin.E124**: The value of `list_editable[n]` refers to the first field in `list_display` (`<label>`), which cannot be used unless `list_display_links` is set.
- **admin.E125**: The value of `list_editable[n]` refers to `<field name>`, which is not editable through the admin.
- **admin.E126**: The value of `search_fields` must be a list or tuple.

- **admin.E127**: The value of `date_hierarchy` refers to `<field name>`, which is not an attribute of `<model>`.
- **admin.E128**: The value of `date_hierarchy` must be a `DateField` or `DateTimeField`.

InlineModelAdmin

The following checks are performed on any *InlineModelAdmin* that is registered as an inline on a *ModelAdmin*.

- **admin.E201**: Cannot exclude the field `<field name>`, because it is the foreign key to the parent model `<app_label>.<model>`.
- **admin.E202**: `<model>` has no `ForeignKey` to `<parent model>`./ `<model>` has more than one `ForeignKey` to `<parent model>`.
- **admin.E203**: The value of `extra` must be an integer.
- **admin.E204**: The value of `max_num` must be an integer.
- **admin.E205**: The value of `min_num` must be an integer.
- **admin.E206**: The value of `formset` must inherit from `BaseModelFormSet`.

GenericInlineModelAdmin

The following checks are performed on any *GenericInlineModelAdmin* that is registered as an inline on a *ModelAdmin*.

- **admin.E301**: `'ct_field'` references `<label>`, which is not a field on `<model>`.
- **admin.E302**: `'ct_fk_field'` references `<label>`, which is not a field on `<model>`.
- **admin.E303**: `<model>` has no `GenericForeignKey`.
- **admin.E304**: `<model>` has no `GenericForeignKey` using content type field `<field name>` and object ID field `<field name>`.

AdminSite

The following checks are performed on the default *AdminSite*:

- **admin.E401**: *django.contrib.contenttypes* must be in *INSTALLED_APPS* in order to use the admin application.
- **admin.E402**: *django.contrib.auth.context_processors.auth* must be in *TEMPLATES* in order to use the admin application.

Auth

- **auth.E001**: `REQUIRED_FIELDS` must be a list or tuple.
- **auth.E002**: The field named as the `USERNAME_FIELD` for a custom user model must not be included in `REQUIRED_FIELDS`.
- **auth.E003**: `<field>` must be unique because it is named as the `USERNAME_FIELD`.
- **auth.W004**: `<field>` is named as the `USERNAME_FIELD`, but it is not unique.
- **auth.E005**: The permission codenamed `<codename>` clashes with a builtin permission for model `<model>`.

- **auth.E006**: The permission codenamed `<codename>` is duplicated for model `<model>`.
- **auth.E007**: The *verbose_name* of model `<model>` must be at most 244 characters for its builtin permission names to be at most 255 characters.
- **auth.E008**: The permission named `<name>` of model `<model>` is longer than 255 characters.
- **auth.C009**: `<User model>.is_anonymous` must be an attribute or property rather than a method. Ignoring this is a security issue as anonymous users will be treated as authenticated!
- **auth.C010**: `<User model>.is_authenticated` must be an attribute or property rather than a method. Ignoring this is a security issue as anonymous users will be treated as authenticated!

Content Types

The following checks are performed when a model contains a *GenericForeignKey* or *GenericRelation*:

- **contenttypes.E001**: The `GenericForeignKey` object ID references the non-existent field `<field>`.
- **contenttypes.E002**: The `GenericForeignKey` content type references the non-existent field `<field>`.
- **contenttypes.E003**: `<field>` is not a `ForeignKey`.
- **contenttypes.E004**: `<field>` is not a `ForeignKey` to `contenttypes.ContentType`.

Security

The security checks do not make your site secure. They do not audit code, do intrusion detection, or do anything particularly complex. Rather, they help perform an automated, low-hanging-fruit checklist. They help you remember the simple things that improve your site's security.

Some of these checks may not be appropriate for your particular deployment configuration. For instance, if you do your HTTP to HTTPS redirection in a load balancer, it'd be irritating to be constantly warned about not having enabled *SECURE_SSL_REDIRECT*. Use *SILENCED_SYSTEM_CHECKS* to silence unneeded checks.

The following checks are run if you use the *check --deploy* option:

- **security.W001**: You do not have *django.middleware.security.SecurityMiddleware* in your *MIDDLEWARE*/*MIDDLEWARE_CLASSES* so the *SECURE_HSTS_SECONDS*, *SECURE_CONTENT_TYPE_NOSNIFF*, *SECURE_BROWSER_XSS_FILTER*, and *SECURE_SSL_REDIRECT* settings will have no effect.
- **security.W002**: You do not have *django.middleware.clickjacking.XFrameOptionsMiddleware* in your *MIDDLEWARE*/*MIDDLEWARE_CLASSES*, so your pages will not be served with an `'x-frame-options'` header. Unless there is a good reason for your site to be served in a frame, you should consider enabling this header to help prevent clickjacking attacks.
- **security.W003**: You don't appear to be using Django's built-in cross-site request forgery protection via the middleware (*django.middleware.csrf.CsrfViewMiddleware* is not in your *MIDDLEWARE*/*MIDDLEWARE_CLASSES*). Enabling the middleware is the safest approach to ensure you don't leave any holes.
- **security.W004**: You have not set a value for the *SECURE_HSTS_SECONDS* setting. If your entire site is served only over SSL, you may want to consider setting a value and enabling *HTTP Strict Transport Security*. Be sure to read the documentation first; enabling HSTS carelessly can cause serious, irreversible problems.
- **security.W005**: You have not set the *SECURE_HSTS_INCLUDE_SUBDOMAINS* setting to `True`. Without this, your site is potentially vulnerable to attack via an insecure connection to a subdomain. Only set this to `True` if you are certain that all subdomains of your domain should be served exclusively via SSL.

- **security.W006**: Your *SECURE_CONTENT_TYPE_NOSNIFF* setting is not set to `True`, so your pages will not be served with an `'x-content-type-options: nosniff'` header. You should consider enabling this header to prevent the browser from identifying content types incorrectly.
- **security.W007**: Your *SECURE_BROWSER_XSS_FILTER* setting is not set to `True`, so your pages will not be served with an `'x-xss-protection: 1; mode=block'` header. You should consider enabling this header to activate the browser's XSS filtering and help prevent XSS attacks.
- **security.W008**: Your *SECURE_SSL_REDIRECT* setting is not set to `True`. Unless your site should be available over both SSL and non-SSL connections, you may want to either set this setting to `True` or configure a load balancer or reverse-proxy server to redirect all connections to HTTPS.
- **security.W009**: Your *SECRET_KEY* has less than 50 characters or less than 5 unique characters. Please generate a long and random `SECRET_KEY`, otherwise many of Django's security-critical features will be vulnerable to attack.
- **security.W010**: You have *django.contrib.sessions* in your *INSTALLED_APPS* but you have not set *SESSION_COOKIE_SECURE* to `True`. Using a secure-only session cookie makes it more difficult for network traffic sniffers to hijack user sessions.
- **security.W011**: You have *django.contrib.sessions.middleware.SessionMiddleware* in your *MIDDLEWARE*/*MIDDLEWARE_CLASSES*, but you have not set *SESSION_COOKIE_SECURE* to `True`. Using a secure-only session cookie makes it more difficult for network traffic sniffers to hijack user sessions.
- **security.W012**: *SESSION_COOKIE_SECURE* is not set to `True`. Using a secure-only session cookie makes it more difficult for network traffic sniffers to hijack user sessions.
- **security.W013**: You have *django.contrib.sessions* in your *INSTALLED_APPS*, but you have not set *SESSION_COOKIE_HTTPONLY* to `True`. Using an `HttpOnly` session cookie makes it more difficult for cross-site scripting attacks to hijack user sessions.
- **security.W014**: You have *django.contrib.sessions.middleware.SessionMiddleware* in your *MIDDLEWARE*/*MIDDLEWARE_CLASSES*, but you have not set *SESSION_COOKIE_HTTPONLY* to `True`. Using an `HttpOnly` session cookie makes it more difficult for cross-site scripting attacks to hijack user sessions.
- **security.W015**: *SESSION_COOKIE_HTTPONLY* is not set to `True`. Using an `HttpOnly` session cookie makes it more difficult for cross-site scripting attacks to hijack user sessions.
- **security.W016**: *CSRF_COOKIE_SECURE* is not set to `True`. Using a secure-only CSRF cookie makes it more difficult for network traffic sniffers to steal the CSRF token.
- **security.W017**: *CSRF_COOKIE_HTTPONLY* is not set to `True`. Using an `HttpOnly` CSRF cookie makes it more difficult for cross-site scripting attacks to steal the CSRF token.
- **security.W018**: You should not have *DEBUG* set to `True` in deployment.
- **security.W019**: You have *django.middleware.clickjacking.XFrameOptionsMiddleware* in your *MIDDLEWARE*/*MIDDLEWARE_CLASSES*, but *X_FRAME_OPTIONS* is not set to `'DENY'`. The default is `'SAMEORIGIN'`, but unless there is a good reason for your site to serve other parts of itself in a frame, you should change it to `'DENY'`.
- **security.W020**: *ALLOWED_HOSTS* must not be empty in deployment.

Sites

The following checks are performed on any model using a *CurrentSiteManager*:

- **sites.E001**: `CurrentSiteManager` could not find a field named `<field name>`.
- **sites.E002**: `CurrentSiteManager` cannot use `<field>` as it is not a foreign key or a many-to-many field.

Database

MySQL

If you're using MySQL, the following checks will be performed:

- **mysql.E001**: MySQL does not allow unique `CharFields` to have a `max_length` > 255.
- **mysql.W002**: MySQL Strict Mode is not set for database connection '<alias>'. See also *Setting sql_mode*.

Templates

The following checks verify that your *TEMPLATES* setting is correctly configured:

- **templates.E001**: You have `'APP_DIRS': True` in your *TEMPLATES* but also specify `'loaders'` in `OPTIONS`. Either remove `APP_DIRS` or remove the `'loaders'` option.
- **templates.E002**: `string_if_invalid` in *TEMPLATES OPTIONS* must be a string but got: `{value}` (`{type}`).

Caches

The following checks verify that your *CACHES* setting is correctly configured:

- **caches.E001**: You must define a `'default'` cache in your *CACHES* setting.

URLs

The following checks are performed on your URL configuration:

- **urls.W001**: Your URL pattern `<pattern>` uses *include()* with a `regex` ending with a `$`. Remove the dollar from the `regex` to avoid problems including URLs.
- **urls.W002**: Your URL pattern `<pattern>` has a `regex` beginning with a `/`. Remove this slash as it is unnecessary. If this pattern is targeted in an *include()*, ensure the *include()* pattern has a trailing `/`.
- **urls.W003**: Your URL pattern `<pattern>` has a `name` including a `:`. Remove the colon, to avoid ambiguous namespace references.
- **urls.E004**: Your URL pattern `<pattern>` is invalid. Ensure that `urlpatterns` is a list of *url()* instances.

6.3 Built-in class-based views API

Class-based views API reference. For introductory material, see the Class-based views topic guide.

6.3.1 Base views

The following three classes provide much of the functionality needed to create Django views. You may think of them as *parent* views, which can be used by themselves or inherited from. They may not provide all the capabilities required for projects, in which case there are Mixins and Generic class-based views.

Many of Django's built-in class-based views inherit from other class-based views or various mixins. Because this inheritance chain is very important, the ancestor classes are documented under the section title of **Ancestors (MRO)**. MRO is an acronym for Method Resolution Order.

View

class django.views.generic.base.**View**

The master class-based base view. All other class-based views inherit from this base class. It isn't strictly a generic view and thus can also be imported from `django.views`.

The ability to import from `django.views` was added.

Method Flowchart

1. *dispatch()*
2. *http_method_not_allowed()*
3. *options()*

Example views.py:

```
from django.http import HttpResponse
from django.views import View

class MyView(View):

    def get(self, request, *args, **kwargs):
        return HttpResponse('Hello, World!')
```

Example urls.py:

```
from django.conf.urls import url

from myapp.views import MyView

urlpatterns = [
    url(r'^mine/$', MyView.as_view(), name='my-view'),
]
```

Attributes

http_method_names

The list of HTTP method names that this view will accept.

Default:

```
['get', 'post', 'put', 'patch', 'delete', 'head', 'options', 'trace']
```

Methods

classmethod as_view(***initkwargs*)

Returns a callable view that takes a request and returns a response:

```
response = MyView.as_view()(request)
```

The returned view has `view_class` and `view_initkwargs` attributes.

dispatch(*request*, **args*, ***kwargs*)

The `view` part of the view – the method that accepts a `request` argument plus arguments, and returns a HTTP response.

The default implementation will inspect the HTTP method and attempt to delegate to a method that matches the HTTP method; a `GET` will be delegated to `get()`, a `POST` to `post()`, and so on.

By default, a `HEAD` request will be delegated to `get()`. If you need to handle `HEAD` requests in a different way than `GET`, you can override the `head()` method. See *Supporting other HTTP methods* for an example.

http_method_not_allowed(*request*, **args*, ***kwargs*)
If the view was called with a HTTP method it doesn't support, this method is called instead.

The default implementation returns `HttpResponseNotAllowed` with a list of allowed methods in plain text.

options(*request*, **args*, ***kwargs*)
Handles responding to requests for the OPTIONS HTTP verb. Returns a response with the `Allow` header containing a list of the view's allowed HTTP method names.

TemplateView

class `django.views.generic.base.`**TemplateView**
Renders a given template, with the context containing parameters captured in the URL.

Ancestors (MRO)

This view inherits methods and attributes from the following views:

- *django.views.generic.base.TemplateResponseMixin*
- *django.views.generic.base.ContextMixin*
- *django.views.generic.base.View*

Method Flowchart

1. *dispatch()*
2. *http_method_not_allowed()*
3. *get_context_data()*

Example views.py:

```
from django.views.generic.base import TemplateView

from articles.models import Article

class HomePageView(TemplateView):

    template_name = "home.html"

    def get_context_data(self, **kwargs):
        context = super(HomePageView, self).get_context_data(**kwargs)
        context['latest_articles'] = Article.objects.all()[:5]
        return context
```

Example urls.py:

```
from django.conf.urls import url

from myapp.views import HomePageView

urlpatterns = [
    url(r'^$', HomePageView.as_view(), name='home'),
]
```

Context

- Populated (through *ContextMixin*) with the keyword arguments captured from the URL pattern that served the view.

RedirectView

class django.views.generic.base.**RedirectView**

Redirects to a given URL.

The given URL may contain dictionary-style string formatting, which will be interpolated against the parameters captured in the URL. Because keyword interpolation is *always* done (even if no arguments are passed in), any `"%"` characters in the URL must be written as `"%%"` so that Python will convert them to a single percent sign on output.

If the given URL is `None`, Django will return an `HttpResponseGone` (410).

Ancestors (MRO)

This view inherits methods and attributes from the following view:

- *django.views.generic.base.View*

Method Flowchart

1. *dispatch()*
2. *http_method_not_allowed()*
3. *get_redirect_url()*

Example views.py:

```
from django.shortcuts import get_object_or_404
from django.views.generic.base import RedirectView

from articles.models import Article

class ArticleCounterRedirectView(RedirectView):

    permanent = False
    query_string = True
    pattern_name = 'article-detail'

    def get_redirect_url(self, *args, **kwargs):
        article = get_object_or_404(Article, pk=kwargs['pk'])
        article.update_counter()
        return super(ArticleCounterRedirectView, self).get_redirect_url(*args, **kwargs)
```

Example urls.py:

```
from django.conf.urls import url
from django.views.generic.base import RedirectView

from article.views import ArticleCounterRedirectView, ArticleDetail

urlpatterns = [
    url(r'^counter/(?P<pk>[0-9]+)/$', ArticleCounterRedirectView.as_view(), name='article-counte
    url(r'^details/(?P<pk>[0-9]+)/$', ArticleDetail.as_view(), name='article-detail'),
    url(r'^go-to-django/$', RedirectView.as_view(url='https://djangoproject.com'), name='go-to-d
]
```

Attributes

url

The URL to redirect to, as a string. Or `None` to raise a 410 (Gone) HTTP error.

pattern_name
The name of the URL pattern to redirect to. Reversing will be done using the same args and kwargs as are passed in for this view.

permanent
Whether the redirect should be permanent. The only difference here is the HTTP status code returned. If `True`, then the redirect will use status code 301. If `False`, then the redirect will use status code 302. By default, `permanent` is `False`.

The default value of the `permanent` attribute changed from `True` to `False`.

query_string
Whether to pass along the GET query string to the new location. If `True`, then the query string is appended to the URL. If `False`, then the query string is discarded. By default, `query_string` is `False`.

Methods

get_redirect_url(**args*, ***kwargs*)
Constructs the target URL for redirection.

The default implementation uses *url* as a starting string and performs expansion of `%` named parameters in that string using the named groups captured in the URL.

If *url* is not set, `get_redirect_url()` tries to reverse the *pattern_name* using what was captured in the URL (both named and unnamed groups are used).

If requested by *query_string*, it will also append the query string to the generated URL. Subclasses may implement any behavior they wish, as long as the method returns a redirect-ready URL string.

6.3.2 Generic display views

The two following generic class-based views are designed to display data. On many projects they are typically the most commonly used views.

DetailView

class `django.views.generic.detail.`**DetailView**
While this view is executing, `self.object` will contain the object that the view is operating upon.

Ancestors (MRO)

This view inherits methods and attributes from the following views:

- *django.views.generic.detail.SingleObjectTemplateResponseMixin*
- *django.views.generic.base.TemplateResponseMixin*
- `django.views.generic.detail.BaseDetailView`
- *django.views.generic.detail.SingleObjectMixin*
- *django.views.generic.base.View*

Method Flowchart

1. *dispatch()*
2. *http_method_not_allowed()*
3. *get_template_names()*
4. *get_slug_field()*

5. *get_queryset()*
6. *get_object()*
7. *get_context_object_name()*
8. *get_context_data()*
9. get()
10. *render_to_response()*

Example myapp/views.py:

```
from django.views.generic.detail import DetailView
from django.utils import timezone

from articles.models import Article

class ArticleDetailView(DetailView):

    model = Article

    def get_context_data(self, **kwargs):
        context = super(ArticleDetailView, self).get_context_data(**kwargs)
        context['now'] = timezone.now()
        return context
```

Example myapp/urls.py:

```
from django.conf.urls import url

from article.views import ArticleDetailView

urlpatterns = [
    url(r'^(?P<slug>[-\w]+)/$', ArticleDetailView.as_view(), name='article-detail'),
]
```

Example myapp/article_detail.html:

```
<h1>{{ object.headline }}</h1>
<p>{{ object.content }}</p>
<p>Reporter: {{ object.reporter }}</p>
<p>Published: {{ object.pub_date|date }}</p>
<p>Date: {{ now|date }}</p>
```

ListView

class django.views.generic.list.**ListView**

A page representing a list of objects.

While this view is executing, `self.object_list` will contain the list of objects (usually, but not necessarily a queryset) that the view is operating upon.

Ancestors (MRO)

This view inherits methods and attributes from the following views:

- *django.views.generic.list.MultipleObjectTemplateResponseMixin*
- *django.views.generic.base.TemplateResponseMixin*

- *django.views.generic.list.BaseListView*
- *django.views.generic.list.MultipleObjectMixin*
- *django.views.generic.base.View*

Method Flowchart

1. *dispatch()*
2. *http_method_not_allowed()*
3. *get_template_names()*
4. *get_queryset()*
5. *get_context_object_name()*
6. *get_context_data()*
7. get()
8. *render_to_response()*

Example views.py:

```
from django.views.generic.list import ListView
from django.utils import timezone

from articles.models import Article

class ArticleListView(ListView):

    model = Article

    def get_context_data(self, **kwargs):
        context = super(ArticleListView, self).get_context_data(**kwargs)
        context['now'] = timezone.now()
        return context
```

Example myapp/urls.py:

```
from django.conf.urls import url

from article.views import ArticleListView

urlpatterns = [
    url(r'^$', ArticleListView.as_view(), name='article-list'),
]
```

Example myapp/article_list.html:

```
<h1>Articles</h1>
<ul>
{% for article in object_list %}
    <li>{{ article.pub_date|date }} - {{ article.headline }}</li>
{% empty %}
    <li>No articles yet.</li>
{% endfor %}
</ul>
```

class django.views.generic.list.**BaseListView**

A base view for displaying a list of objects. It is not intended to be used directly, but rather as a parent class of the *django.views.generic.list.ListView* or other views representing lists of objects.

Ancestors (MRO)

This view inherits methods and attributes from the following views:

- *django.views.generic.list.MultipleObjectMixin*
- *django.views.generic.base.View*

Methods

get(*request*, **args*, ***kwargs*)
Adds `object_list` to the context. If *allow_empty* is True then display an empty list. If *allow_empty* is False then raise a 404 error.

6.3.3 Generic editing views

The following views are described on this page and provide a foundation for editing content:

- *django.views.generic.edit.FormView*
- *django.views.generic.edit.CreateView*
- *django.views.generic.edit.UpdateView*
- *django.views.generic.edit.DeleteView*

: Some of the examples on this page assume that an `Author` model has been defined as follows in `myapp/models.py`:

```
from django.urls import reverse
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=200)

    def get_absolute_url(self):
        return reverse('author-detail', kwargs={'pk': self.pk})
```

FormView

class django.views.generic.edit.**FormView**
A view that displays a form. On error, redisplays the form with validation errors; on success, redirects to a new URL.

Ancestors (MRO)

This view inherits methods and attributes from the following views:

- *django.views.generic.base.TemplateResponseMixin*
- django.views.generic.edit.BaseFormView
- *django.views.generic.edit.FormMixin*
- *django.views.generic.edit.ProcessFormView*
- *django.views.generic.base.View*

Example myapp/forms.py:

```
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField()
    message = forms.CharField(widget=forms.Textarea)

    def send_email(self):
        # send email using the self.cleaned_data dictionary
        pass
```

Example myapp/views.py:

```
from myapp.forms import ContactForm
from django.views.generic.edit import FormView

class ContactView(FormView):
    template_name = 'contact.html'
    form_class = ContactForm
    success_url = '/thanks/'

    def form_valid(self, form):
        # This method is called when valid form data has been POSTed.
        # It should return an HttpResponse.
        form.send_email()
        return super(ContactView, self).form_valid(form)
```

Example myapp/contact.html:

```
<form action="" method="post">{% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Send message" />
</form>
```

CreateView

class django.views.generic.edit.**CreateView**

A view that displays a form for creating an object, redisplaying the form with validation errors (if there are any) and saving the object.

Ancestors (MRO)

This view inherits methods and attributes from the following views:

- *django.views.generic.detail.SingleObjectTemplateResponseMixin*
- *django.views.generic.base.TemplateResponseMixin*
- django.views.generic.edit.BaseCreateView
- *django.views.generic.edit.ModelFormMixin*
- *django.views.generic.edit.FormMixin*
- *django.views.generic.detail.SingleObjectMixin*
- *django.views.generic.edit.ProcessFormView*
- *django.views.generic.base.View*

Attributes

template_name_suffix

The `CreateView` page displayed to a `GET` request uses a `template_name_suffix` of `'_form'`. For example, changing this attribute to `'_create_form'` for a view creating objects for the example `Author` model would cause the default `template_name` to be `'myapp/author_create_form.html'`.

object

When using `CreateView` you have access to `self.object`, which is the object being created. If the object hasn't been created yet, the value will be `None`.

Example myapp/views.py:

```
from django.views.generic.edit import CreateView
from myapp.models import Author

class AuthorCreate(CreateView):
    model = Author
    fields = ['name']
```

Example myapp/author_form.html:

```
<form action="" method="post">{% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Save" />
</form>
```

UpdateView

class `django.views.generic.edit.`**UpdateView**

A view that displays a form for editing an existing object, redisplaying the form with validation errors (if there are any) and saving changes to the object. This uses a form automatically generated from the object's model class (unless a form class is manually specified).

Ancestors (MRO)

This view inherits methods and attributes from the following views:

- *django.views.generic.detail.SingleObjectTemplateResponseMixin*
- *django.views.generic.base.TemplateResponseMixin*
- django.views.generic.edit.BaseUpdateView
- *django.views.generic.edit.ModelFormMixin*
- *django.views.generic.edit.FormMixin*
- *django.views.generic.detail.SingleObjectMixin*
- *django.views.generic.edit.ProcessFormView*
- *django.views.generic.base.View*

Attributes

template_name_suffix

The `UpdateView` page displayed to a `GET` request uses a `template_name_suffix` of `'_form'`. For example, changing this attribute to `'_update_form'` for a view updating objects for the example `Author` model would cause the default `template_name` to be `'myapp/author_update_form.html'`.

object
When using `UpdateView` you have access to `self.object`, which is the object being updated.

Example myapp/views.py:

```
from django.views.generic.edit import UpdateView
from myapp.models import Author

class AuthorUpdate(UpdateView):
    model = Author
    fields = ['name']
    template_name_suffix = '_update_form'
```

Example myapp/author_update_form.html:

```
<form action="" method="post">{% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Update" />
</form>
```

DeleteView

class `django.views.generic.edit.`**DeleteView**
A view that displays a confirmation page and deletes an existing object. The given object will only be deleted if the request method is `POST`. If this view is fetched via `GET`, it will display a confirmation page that should contain a form that POSTs to the same URL.

Ancestors (MRO)

This view inherits methods and attributes from the following views:

- *django.views.generic.detail.SingleObjectTemplateResponseMixin*
- *django.views.generic.base.TemplateResponseMixin*
- django.views.generic.edit.BaseDeleteView
- *django.views.generic.edit.DeletionMixin*
- django.views.generic.detail.BaseDetailView
- *django.views.generic.detail.SingleObjectMixin*
- *django.views.generic.base.View*

Attributes

template_name_suffix
The `DeleteView` page displayed to a `GET` request uses a `template_name_suffix` of `'_confirm_delete'`. For example, changing this attribute to `'_check_delete'` for a view deleting objects for the example `Author` model would cause the default `template_name` to be `'myapp/author_check_delete.html'`.

Example myapp/views.py:

```
from django.views.generic.edit import DeleteView
from django.urls import reverse_lazy
from myapp.models import Author

class AuthorDelete(DeleteView):
    model = Author
    success_url = reverse_lazy('author-list')
```

Example myapp/author_confirm_delete.html:

```
<form action="" method="post">{% csrf_token %}
    <p>Are you sure you want to delete "{{ object }}"?</p>
    <input type="submit" value="Confirm" />
</form>
```

6.3.4 Generic date views

Date-based generic views, provided in *django.views.generic.dates*, are views for displaying drilldown pages for date-based data.

: Some of the examples on this page assume that an `Article` model has been defined as follows in `myapp/models.py`:

```
from django.db import models
from django.urls import reverse

class Article(models.Model):
    title = models.CharField(max_length=200)
    pub_date = models.DateField()

    def get_absolute_url(self):
        return reverse('article-detail', kwargs={'pk': self.pk})
```

ArchiveIndexView

class ArchiveIndexView

A top-level index page showing the "latest" objects, by date. Objects with a date in the *future* are not included unless you set `allow_future` to `True`.

Ancestors (MRO)

- *django.views.generic.list.MultipleObjectTemplateResponseMixin*
- *django.views.generic.base.TemplateResponseMixin*
- *django.views.generic.dates.BaseArchiveIndexView*
- *django.views.generic.dates.BaseDateListView*
- *django.views.generic.list.MultipleObjectMixin*
- *django.views.generic.dates.DateMixin*
- *django.views.generic.base.View*

Context

In addition to the context provided by *django.views.generic.list.MultipleObjectMixin* (via *django.views.generic.dates.BaseDateListView*), the template's context will be:

- `date_list`: A *QuerySet* object containing all years that have objects available according to `queryset`, represented as `datetime.datetime` objects, in descending order.

Notes

- Uses a default `context_object_name` of `latest`.

- Uses a default `template_name_suffix` of `_archive`.
- Defaults to providing `date_list` by year, but this can be altered to month or day using the attribute `date_list_period`. This also applies to all subclass views.

Example myapp/urls.py:

```
from django.conf.urls import url
from django.views.generic.dates import ArchiveIndexView

from myapp.models import Article

urlpatterns = [
    url(r'^archive/$',
        ArchiveIndexView.as_view(model=Article, date_field="pub_date"),
        name="article_archive"),
]
```

Example myapp/article_archive.html:

```
<ul>
    {% for article in latest %}
        <li>{{ article.pub_date }}: {{ article.title }}</li>
    {% endfor %}
</ul>
```

This will output all articles.

YearArchiveView

class YearArchiveView

A yearly archive page showing all available months in a given year. Objects with a date in the *future* are not displayed unless you set `allow_future` to `True`.

Ancestors (MRO)

- *django.views.generic.list.MultipleObjectTemplateResponseMixin*
- *django.views.generic.base.TemplateResponseMixin*
- *django.views.generic.dates.BaseYearArchiveView*
- *django.views.generic.dates.YearMixin*
- *django.views.generic.dates.BaseDateListView*
- *django.views.generic.list.MultipleObjectMixin*
- *django.views.generic.dates.DateMixin*
- *django.views.generic.base.View*

make_object_list

A boolean specifying whether to retrieve the full list of objects for this year and pass those to the template. If `True`, the list of objects will be made available to the context. If `False`, the `None` queryset will be used as the object list. By default, this is `False`.

get_make_object_list()

Determine if an object list will be returned as part of the context. Returns *make_object_list* by default.

Context

In addition to the context provided by *django.views.generic.list.MultipleObjectMixin* (via *django.views.generic.dates.BaseDateListView*), the template's context will be:

- `date_list`: A *QuerySet* object containing all months that have objects available according to `queryset`, represented as `datetime.datetime` objects, in ascending order.
- `year`: A `date` object representing the given year.
- `next_year`: A `date` object representing the first day of the next year, according to *allow_empty* and *allow_future*.
- `previous_year`: A `date` object representing the first day of the previous year, according to *allow_empty* and *allow_future*.

Notes

- Uses a default `template_name_suffix` of `_archive_year`.

Example myapp/views.py:

```
from django.views.generic.dates import YearArchiveView

from myapp.models import Article

class ArticleYearArchiveView(YearArchiveView):
    queryset = Article.objects.all()
    date_field = "pub_date"
    make_object_list = True
    allow_future = True
```

Example myapp/urls.py:

```
from django.conf.urls import url

from myapp.views import ArticleYearArchiveView

urlpatterns = [
    url(r'^(?P<year>[0-9]{4})/$',
        ArticleYearArchiveView.as_view(),
        name="article_year_archive"),
]
```

Example myapp/article_archive_year.html:

```
<ul>
    {% for date in date_list %}
        <li>{{ date|date }}</li>
    {% endfor %}
</ul>

<div>
    <h1>All Articles for {{ year|date:"Y" }}</h1>
    {% for obj in object_list %}
        <p>
            {{ obj.title }} - {{ obj.pub_date|date:"F j, Y" }}
        </p>
    {% endfor %}
</div>
```

MonthArchiveView

class MonthArchiveView

A monthly archive page showing all objects in a given month. Objects with a date in the *future* are not displayed unless you set `allow_future` to `True`.

Ancestors (MRO)

- *django.views.generic.list.MultipleObjectTemplateResponseMixin*
- *django.views.generic.base.TemplateResponseMixin*
- *django.views.generic.dates.BaseMonthArchiveView*
- *django.views.generic.dates.YearMixin*
- *django.views.generic.dates.MonthMixin*
- *django.views.generic.dates.BaseDateListView*
- *django.views.generic.list.MultipleObjectMixin*
- *django.views.generic.dates.DateMixin*
- *django.views.generic.base.View*

Context

In addition to the context provided by *MultipleObjectMixin* (via *BaseDateListView*), the template's context will be:

- `date_list`: A *QuerySet* object containing all days that have objects available in the given month, according to `queryset`, represented as `datetime.datetime` objects, in ascending order.
- `month`: A `date` object representing the given month.
- `next_month`: A `date` object representing the first day of the next month, according to *allow_empty* and *allow_future*.
- `previous_month`: A `date` object representing the first day of the previous month, according to *allow_empty* and *allow_future*.

Notes

- Uses a default `template_name_suffix` of `_archive_month`.

Example myapp/views.py:

```
from django.views.generic.dates import MonthArchiveView

from myapp.models import Article

class ArticleMonthArchiveView(MonthArchiveView):
    queryset = Article.objects.all()
    date_field = "pub_date"
    allow_future = True
```

Example myapp/urls.py:

```
from django.conf.urls import url

from myapp.views import ArticleMonthArchiveView

urlpatterns = [
    # Example: /2012/aug/
```

```
        url(r'^(?P<year>[0-9]{4})/(?P<month>[-\w]+)/$',
            ArticleMonthArchiveView.as_view(),
            name="archive_month"),
        # Example: /2012/08/
        url(r'^(?P<year>[0-9]{4})/(?P<month>[0-9]+)/$',
            ArticleMonthArchiveView.as_view(month_format='%m'),
            name="archive_month_numeric"),
    ]
```

Example myapp/article_archive_month.html:

```
    <ul>
        {% for article in object_list %}
            <li>{{ article.pub_date|date:"F j, Y" }}: {{ article.title }}</li>
        {% endfor %}
    </ul>

    <p>
        {% if previous_month %}
            Previous Month: {{ previous_month|date:"F Y" }}
        {% endif %}
        {% if next_month %}
            Next Month: {{ next_month|date:"F Y" }}
        {% endif %}
    </p>
```

WeekArchiveView

class WeekArchiveView

A weekly archive page showing all objects in a given week. Objects with a date in the *future* are not displayed unless you set `allow_future` to `True`.

Ancestors (MRO)

- *django.views.generic.list.MultipleObjectTemplateResponseMixin*
- *django.views.generic.base.TemplateResponseMixin*
- *django.views.generic.dates.BaseWeekArchiveView*
- *django.views.generic.dates.YearMixin*
- *django.views.generic.dates.WeekMixin*
- *django.views.generic.dates.BaseDateListView*
- *django.views.generic.list.MultipleObjectMixin*
- *django.views.generic.dates.DateMixin*
- *django.views.generic.base.View*

Context

In addition to the context provided by *MultipleObjectMixin* (via *BaseDateListView*), the template's context will be:

- `week`: A `date` object representing the first day of the given week.
- `next_week`: A `date` object representing the first day of the next week, according to *allow_empty* and *allow_future*.

- previous_week: A date object representing the first day of the previous week, according to allow_empty and allow_future.

Notes

- Uses a default template_name_suffix of _archive_week.
- The week_format attribute is a strptime() format string used to parse the week number. The following values are supported:
  - '%U': Based on the United States week system where the week begins on Sunday. This is the default value.
  - '%W': Similar to '%U', except it assumes that the week begins on Monday. This is not the same as the ISO 8601 week number.

Example myapp/views.py:

```
from django.views.generic.dates import WeekArchiveView

from myapp.models import Article

class ArticleWeekArchiveView(WeekArchiveView):
    queryset = Article.objects.all()
    date_field = "pub_date"
    week_format = "%W"
    allow_future = True
```

Example myapp/urls.py:

```
from django.conf.urls import url

from myapp.views import ArticleWeekArchiveView

urlpatterns = [
    # Example: /2012/week/23/
    url(r'^(?P<year>[0-9]{4})/week/(?P<week>[0-9]+)/$',
        ArticleWeekArchiveView.as_view(),
        name="archive_week"),
]
```

Example myapp/article_archive_week.html:

```
<h1>Week {{ week|date:'W' }}</h1>

<ul>
    {% for article in object_list %}
        <li>{{ article.pub_date|date:"F j, Y" }}: {{ article.title }}</li>
    {% endfor %}
</ul>

<p>
    {% if previous_week %}
        Previous Week: {{ previous_week|date:"W" }} of year {{ previous_week|date:"Y" }}
    {% endif %}
    {% if previous_week and next_week %}--{% endif %}
    {% if next_week %}
        Next week: {{ next_week|date:"W" }} of year {{ next_week|date:"Y" }}
    {% endif %}
</p>
```

In this example, you are outputting the week number. Keep in mind that week numbers computed by the *date* template filter with the `'W'` format character are not always the same as those computed by `strftime()` and `strptime()` with the `'%W'` format string. For year 2015, for example, week numbers output by *date* are higher by one compared to those output by `strftime()`. There isn't an equivalent for the `'%U'` `strftime()` format string in *date*. Therefore, you should avoid using *date* to generate URLs for `WeekArchiveView`.

DayArchiveView

class DayArchiveView

A day archive page showing all objects in a given day. Days in the future throw a 404 error, regardless of whether any objects exist for future days, unless you set `allow_future` to `True`.

Ancestors (MRO)

- *django.views.generic.list.MultipleObjectTemplateResponseMixin*
- *django.views.generic.base.TemplateResponseMixin*
- *django.views.generic.dates.BaseDayArchiveView*
- *django.views.generic.dates.YearMixin*
- *django.views.generic.dates.MonthMixin*
- *django.views.generic.dates.DayMixin*
- *django.views.generic.dates.BaseDateListView*
- *django.views.generic.list.MultipleObjectMixin*
- *django.views.generic.dates.DateMixin*
- *django.views.generic.base.View*

Context

In addition to the context provided by *MultipleObjectMixin* (via *BaseDateListView*), the template's context will be:

- `day`: A `date` object representing the given day.
- `next_day`: A `date` object representing the next day, according to *allow_empty* and *allow_future*.
- `previous_day`: A `date` object representing the previous day, according to *allow_empty* and *allow_future*.
- `next_month`: A `date` object representing the first day of the next month, according to *allow_empty* and *allow_future*.
- `previous_month`: A `date` object representing the first day of the previous month, according to *allow_empty* and *allow_future*.

Notes

- Uses a default `template_name_suffix` of `_archive_day`.

Example myapp/views.py:

```
from django.views.generic.dates import DayArchiveView

from myapp.models import Article
```

```
class ArticleDayArchiveView(DayArchiveView):
    queryset = Article.objects.all()
    date_field = "pub_date"
    allow_future = True
```

Example myapp/urls.py:

```
from django.conf.urls import url

from myapp.views import ArticleDayArchiveView

urlpatterns = [
    # Example: /2012/nov/10/
    url(r'^(?P<year>[0-9]{4})/(?P<month>[-\w]+)/(?P<day>[0-9]+)/$',
        ArticleDayArchiveView.as_view(),
        name="archive_day"),
]
```

Example myapp/article_archive_day.html:

```
<h1>{{ day }}</h1>

<ul>
    {% for article in object_list %}
        <li>{{ article.pub_date|date:"F j, Y" }}: {{ article.title }}</li>
    {% endfor %}
</ul>

<p>
    {% if previous_day %}
        Previous Day: {{ previous_day }}
    {% endif %}
    {% if previous_day and next_day %}--{% endif %}
    {% if next_day %}
        Next Day: {{ next_day }}
    {% endif %}
</p>
```

TodayArchiveView

class TodayArchiveView

A day archive page showing all objects for *today*. This is exactly the same as *django.views.generic.dates.DayArchiveView*, except today's date is used instead of the `year`/`month`/`day` arguments.

Ancestors (MRO)

- *django.views.generic.list.MultipleObjectTemplateResponseMixin*
- *django.views.generic.base.TemplateResponseMixin*
- *django.views.generic.dates.BaseTodayArchiveView*
- *django.views.generic.dates.BaseDayArchiveView*
- *django.views.generic.dates.YearMixin*
- *django.views.generic.dates.MonthMixin*
- *django.views.generic.dates.DayMixin*

- *django.views.generic.dates.BaseDateListView*
- *django.views.generic.list.MultipleObjectMixin*
- *django.views.generic.dates.DateMixin*
- *django.views.generic.base.View*

Notes

- Uses a default `template_name_suffix` of `_archive_today`.

Example myapp/views.py:

```
from django.views.generic.dates import TodayArchiveView

from myapp.models import Article

class ArticleTodayArchiveView(TodayArchiveView):
    queryset = Article.objects.all()
    date_field = "pub_date"
    allow_future = True
```

Example myapp/urls.py:

```
from django.conf.urls import url

from myapp.views import ArticleTodayArchiveView

urlpatterns = [
    url(r'^today/$',
        ArticleTodayArchiveView.as_view(),
        name="archive_today"),
]
```

Where is the example template for `TodayArchiveView`?

This view uses by default the same template as the *DayArchiveView*, which is in the previous example. If you need a different template, set the `template_name` attribute to be the name of the new template.

DateDetailView

class `DateDetailView`

A page representing an individual object. If the object has a date value in the future, the view will throw a 404 error by default, unless you set `allow_future` to `True`.

Ancestors (MRO)

- *django.views.generic.detail.SingleObjectTemplateResponseMixin*
- *django.views.generic.base.TemplateResponseMixin*
- *django.views.generic.dates.BaseDateDetailView*
- *django.views.generic.dates.YearMixin*
- *django.views.generic.dates.MonthMixin*
- *django.views.generic.dates.DayMixin*
- *django.views.generic.dates.DateMixin*

- `django.views.generic.detail.BaseDetailView`
- *`django.views.generic.detail.SingleObjectMixin`*
- *`django.views.generic.base.View`*

Context

- Includes the single object associated with the `model` specified in the `DateDetailView`.

Notes

- Uses a default `template_name_suffix` of `_detail`.

Example myapp/urls.py:

```
from django.conf.urls import url
from django.views.generic.dates import DateDetailView

urlpatterns = [
    url(r'^(?P<year>[0-9]{4})/(?P<month>[-\w]+)/(?P<day>[0-9]+)/(?P<pk>[0-9]+)/$',
        DateDetailView.as_view(model=Article, date_field="pub_date"),
        name="archive_date_detail"),
]
```

Example myapp/article_detail.html:

```
<h1>{{ object.title }}</h1>
```

: All of the generic views listed above have matching `Base` views that only differ in that they do not include the *`MultipleObjectTemplateResponseMixin`* (for the archive views) or *`SingleObjectTemplateResponseMixin`* (for the *`DateDetailView`*):

class BaseArchiveIndexView

class BaseYearArchiveView

class BaseMonthArchiveView

class BaseWeekArchiveView

class BaseDayArchiveView

class BaseTodayArchiveView

class BaseDateDetailView

6.3.5 Class-based views mixins

Class-based views API reference. For introductory material, see Using mixins with class-based views.

Simple mixins

ContextMixin

class `django.views.generic.base.`**ContextMixin**

Methods

get_context_data(***kwargs*)
Returns a dictionary representing the template context. The keyword arguments provided will make up the returned context. Example usage:

```
def get_context_data(self, **kwargs):
    context = super(RandomNumberView, self).get_context_data(**kwargs)
    context['number'] = random.randrange(1, 100)
    return context
```

The template context of all class-based generic views include a `view` variable that points to the `View` instance.

Use `alters_data` where appropriate

Note that having the view instance in the template context may expose potentially hazardous methods to template authors. To prevent methods like this from being called in the template, set `alters_data=True` on those methods. For more information, read the documentation on *rendering a template context*.

TemplateResponseMixin

class `django.views.generic.base.`**TemplateResponseMixin**
Provides a mechanism to construct a *`TemplateResponse`*, given suitable context. The template to use is configurable and can be further customized by subclasses.

Attributes

template_name
The full name of a template to use as defined by a string. Not defining a `template_name` will raise a *`django.core.exceptions.ImproperlyConfigured`* exception.

template_engine
The *`NAME`* of a template engine to use for loading the template. `template_engine` is passed as the `using` keyword argument to `response_class`. Default is `None`, which tells Django to search for the template in all configured engines.

response_class
The response class to be returned by `render_to_response` method. Default is *`TemplateResponse`*. The template and context of `TemplateResponse` instances can be altered later (e.g. in *template response middleware*).

If you need custom template loading or custom context object instantiation, create a `TemplateResponse` subclass and assign it to `response_class`.

content_type
The content type to use for the response. `content_type` is passed as a keyword argument to `response_class`. Default is `None` – meaning that Django uses *`DEFAULT_CONTENT_TYPE`*.

Methods

render_to_response(*context*, ***response_kwargs*)
Returns a `self.response_class` instance.

If any keyword arguments are provided, they will be passed to the constructor of the response class.

Calls *`get_template_names()`* to obtain the list of template names that will be searched looking for an existent template.

get_template_names()
Returns a list of template names to search for when rendering the template. The first template that is found will be used.

If *template_name* is specified, the default implementation will return a list containing *template_name* (if it is specified).

Single object mixins

SingleObjectMixin

class django.views.generic.detail.**SingleObjectMixin**
Provides a mechanism for looking up an object associated with the current HTTP request.

Methods and Attributes

model
The model that this view will display data for. Specifying `model = Foo` is effectively the same as specifying `queryset = Foo.objects.all()`, where `objects` stands for `Foo`'s *default manager*.

queryset
A `QuerySet` that represents the objects. If provided, the value of `queryset` supersedes the value provided for *model*.

: `queryset` is a class attribute with a *mutable* value so care must be taken when using it directly. Before using it, either call its *all()* method or retrieve it with *get_queryset()* which takes care of the cloning behind the scenes.

slug_field
The name of the field on the model that contains the slug. By default, `slug_field` is `'slug'`.

slug_url_kwarg
The name of the URLConf keyword argument that contains the slug. By default, `slug_url_kwarg` is `'slug'`.

pk_url_kwarg
The name of the URLConf keyword argument that contains the primary key. By default, `pk_url_kwarg` is `'pk'`.

context_object_name
Designates the name of the variable to use in the context.

query_pk_and_slug
If `True`, causes *get_object()* to perform its lookup using both the primary key and the slug. Defaults to `False`.

This attribute can help mitigate insecure direct object reference attacks. When applications allow access to individual objects by a sequential primary key, an attacker could brute-force guess all URLs; thereby obtaining a list of all objects in the application. If users with access to individual objects should be prevented from obtaining this list, setting `query_pk_and_slug` to `True` will help prevent the guessing of URLs as each URL will require two correct, non-sequential arguments. Simply using a unique slug may serve the same purpose, but this scheme allows you to have non-unique slugs.

get_object(*queryset=None*)
Returns the single object that this view will display. If `queryset` is provided, that queryset will be used as the source of objects; otherwise, *get_queryset()* will be used. `get_object()` looks for a *pk_url_kwarg* argument in the arguments to the view; if this argument is found, this method

performs a primary-key based lookup using that value. If this argument is not found, it looks for a *slug_url_kwarg* argument, and performs a slug lookup using the *slug_field*.

When *query_pk_and_slug* is `True`, `get_object()` will perform its lookup using both the primary key and the slug.

get_queryset()
Returns the queryset that will be used to retrieve the object that this view will display. By default, *get_queryset()* returns the value of the *queryset* attribute if it is set, otherwise it constructs a *QuerySet* by calling the `all()` method on the *model* attribute's default manager.

get_context_object_name(*obj*)
Return the context variable name that will be used to contain the data that this view is manipulating. If *context_object_name* is not set, the context name will be constructed from the `model_name` of the model that the queryset is composed from. For example, the model `Article` would have context object named `'article'`.

get_context_data(***kwargs*)
Returns context data for displaying the list of objects.

The base implementation of this method requires that the `self.object` attribute be set by the view (even if `None`). Be sure to do this if you are using this mixin without one of the built-in views that does so.

It returns a dictionary with these contents:

- `object`: The object that this view is displaying (`self.object`).
- `context_object_name`: `self.object` will also be stored under the name returned by *get_context_object_name()*, which defaults to the lowercased version of the model name.

Context variables override values from template context processors

Any variables from *get_context_data()* take precedence over context variables from *context processors*. For example, if your view sets the *model* attribute to *User*, the default context object name of `user` would override the `user` variable from the *django.contrib.auth.context_processors.auth()* context processor. Use *get_context_object_name()* to avoid a clash.

---

get_slug_field()
Returns the name of a slug field to be used to look up by slug. By default this simply returns the value of *slug_field*.

SingleObjectTemplateResponseMixin

class `django.views.generic.detail.`**SingleObjectTemplateResponseMixin**
A mixin class that performs template-based response rendering for views that operate upon a single object instance. Requires that the view it is mixed with provides `self.object`, the object instance that the view is operating on. `self.object` will usually be, but is not required to be, an instance of a Django model. It may be `None` if the view is in the process of constructing a new instance.

Extends

- *TemplateResponseMixin*

Methods and Attributes

template_name_field
The field on the current object instance that can be used to determine the name of a candidate template. If either `template_name_field` itself or the value of the `template_name_field` on the current object instance is `None`, the object will not be used for a candidate template name.

template_name_suffix
The suffix to append to the auto-generated candidate template name. Default suffix is `_detail`.

get_template_names()
Returns a list of candidate template names. Returns the following list:

- the value of `template_name` on the view (if provided)
- the contents of the `template_name_field` field on the object instance that the view is operating upon (if available)
- `<app_label>/<model_name><template_name_suffix>.html`

Multiple object mixins

MultipleObjectMixin

class `django.views.generic.list.`**MultipleObjectMixin**
A mixin that can be used to display a list of objects.

If `paginate_by` is specified, Django will paginate the results returned by this. You can specify the page number in the URL in one of two ways:

- Use the `page` parameter in the URLconf. For example, this is what your URLconf might look like:

```
url(r'^objects/page(?P<page>[0-9]+)/$', PaginatedView.as_view()),
```

- Pass the page number via the `page` query-string parameter. For example, a URL would look like this:

```
/objects/?page=3
```

These values and lists are 1-based, not 0-based, so the first page would be represented as page `1`.

For more on pagination, read the pagination documentation.

As a special case, you are also permitted to use `last` as a value for `page`:

```
/objects/?page=last
```

This allows you to access the final page of results without first having to determine how many pages there are.

Note that `page` *must* be either a valid page number or the value `last`; any other value for `page` will result in a 404 error.

Extends

- *django.views.generic.base.ContextMixin*

Methods and Attributes

allow_empty
A boolean specifying whether to display the page if no objects are available. If this is `False` and no objects are available, the view will raise a 404 instead of displaying an empty page. By default, this is `True`.

model
The model that this view will display data for. Specifying `model = Foo` is effectively the same as specifying `queryset = Foo.objects.all()`, where `objects` stands for `Foo`'s *default manager*.

queryset
A `QuerySet` that represents the objects. If provided, the value of `queryset` supersedes the value provided for *model*.

: `queryset` is a class attribute with a *mutable* value so care must be taken when using it directly. Before using it, either call its *all()* method or retrieve it with *get_queryset()* which takes care of the cloning behind the scenes.

ordering
A string or list of strings specifying the ordering to apply to the `queryset`. Valid values are the same as those for *order_by()*.

paginate_by
An integer specifying how many objects should be displayed per page. If this is given, the view will paginate objects with `paginate_by` objects per page. The view will expect either a `page` query string parameter (via `request.GET`) or a `page` variable specified in the URLconf.

paginate_orphans
An integer specifying the number of "overflow" objects the last page can contain. This extends the *paginate_by* limit on the last page by up to `paginate_orphans`, in order to keep the last page from having a very small number of objects.

page_kwarg
A string specifying the name to use for the page parameter. The view will expect this parameter to be available either as a query string parameter (via `request.GET`) or as a kwarg variable specified in the URLconf. Defaults to `page`.

paginator_class
The paginator class to be used for pagination. By default, *django.core.paginator.Paginator* is used. If the custom paginator class doesn't have the same constructor interface as *django.core.paginator.Paginator*, you will also need to provide an implementation for *get_paginator()*.

context_object_name
Designates the name of the variable to use in the context.

get_queryset()
Get the list of items for this view. This must be an iterable and may be a queryset (in which queryset-specific behavior will be enabled).

get_ordering()
Returns a string (or iterable of strings) that defines the ordering that will be applied to the `queryset`.

Returns *ordering* by default.

paginate_queryset(*queryset*, *page_size*)
Returns a 4-tuple containing (`paginator`, `page`, `object_list`, `is_paginated`).

Constructed by paginating `queryset` into pages of size `page_size`. If the request contains a `page` argument, either as a captured URL argument or as a GET argument, `object_list` will correspond to the objects from that page.

get_paginate_by(*queryset*)
Returns the number of items to paginate by, or `None` for no pagination. By default this simply returns the value of *paginate_by*.

get_paginator(*queryset*, *per_page*, *orphans=0*, *allow_empty_first_page=True*)
Returns an instance of the paginator to use for this view. By default, instantiates an instance of *paginator_class*.

get_paginate_orphans()
An integer specifying the number of "overflow" objects the last page can contain. By default this simply returns the value of *paginate_orphans*.

get_allow_empty()
Return a boolean specifying whether to display the page if no objects are available. If this method returns `False` and no objects are available, the view will raise a 404 instead of displaying an empty page. By default, this is `True`.

get_context_object_name(*object_list*)
Return the context variable name that will be used to contain the list of data that this view is manipulating. If `object_list` is a queryset of Django objects and *context_object_name* is not set, the context name will be the `model_name` of the model that the queryset is composed from, with postfix `'_list'` appended. For example, the model `Article` would have a context object named `article_list`.

get_context_data(***kwargs*)
Returns context data for displaying the list of objects.

Context

- `object_list`: The list of objects that this view is displaying. If `context_object_name` is specified, that variable will also be set in the context, with the same value as `object_list`.
- `is_paginated`: A boolean representing whether the results are paginated. Specifically, this is set to `False` if no page size has been specified, or if the available objects do not span multiple pages.
- `paginator`: An instance of *django.core.paginator.Paginator*. If the page is not paginated, this context variable will be `None`.
- `page_obj`: An instance of *django.core.paginator.Page*. If the page is not paginated, this context variable will be `None`.

MultipleObjectTemplateResponseMixin

class `django.views.generic.list.`**MultipleObjectTemplateResponseMixin**
A mixin class that performs template-based response rendering for views that operate upon a list of object instances. Requires that the view it is mixed with provides `self.object_list`, the list of object instances that the view is operating on. `self.object_list` may be, but is not required to be, a *QuerySet*.

Extends

- *TemplateResponseMixin*

Methods and Attributes

template_name_suffix
The suffix to append to the auto-generated candidate template name. Default suffix is `_list`.

get_template_names()
Returns a list of candidate template names. Returns the following list:

- the value of `template_name` on the view (if provided)
- `<app_label>/<model_name><template_name_suffix>.html`

Editing mixins

The following mixins are used to construct Django's editing views:

- *django.views.generic.edit.FormMixin*
- *django.views.generic.edit.ModelFormMixin*
- *django.views.generic.edit.ProcessFormView*
- *django.views.generic.edit.DeletionMixin*

: Examples of how these are combined into editing views can be found at the documentation on Generic editing views.

FormMixin

class django.views.generic.edit.**FormMixin**

A mixin class that provides facilities for creating and displaying forms.

Mixins

- *django.views.generic.base.ContextMixin*

Methods and Attributes

initial
A dictionary containing initial data for the form.

form_class
The form class to instantiate.

success_url
The URL to redirect to when the form is successfully processed.

prefix
The *prefix* for the generated form.

get_initial()
Retrieve initial data for the form. By default, returns a copy of *initial*.

get_form_class()
Retrieve the form class to instantiate. By default *form_class*.

get_form(*form_class=None*)
Instantiate an instance of form_class using *get_form_kwargs()*. If form_class isn't provided *get_form_class()* will be used.

get_form_kwargs()
Build the keyword arguments required to instantiate the form.

The initial argument is set to *get_initial()*. If the request is a POST or PUT, the request data (request.POST and request.FILES) will also be provided.

get_prefix()
Determine the *prefix* for the generated form. Returns *prefix* by default.

get_success_url()
Determine the URL to redirect to when the form is successfully validated. Returns *success_url* by default.

form_valid(*form*)
Redirects to *get_success_url()*.

form_invalid(*form*)
Renders a response, providing the invalid form as context.

get_context_data(***kwargs*)
Calls *get_form()* and adds the result to the context data with the name 'form'.

ModelFormMixin

class django.views.generic.edit.**ModelFormMixin**
A form mixin that works on ModelForms, rather than a standalone form.

Since this is a subclass of *SingleObjectMixin*, instances of this mixin have access to the *model* and *queryset* attributes, describing the type of object that the ModelForm is manipulating.

If you specify both the *fields* and *form_class* attributes, an *ImproperlyConfigured* exception will be raised.

Mixins

- *django.views.generic.edit.FormMixin*
- *django.views.generic.detail.SingleObjectMixin*

Methods and Attributes

model
A model class. Can be explicitly provided, otherwise will be determined by examining self.object or *queryset*.

fields
A list of names of fields. This is interpreted the same way as the Meta.fields attribute of *ModelForm*.

This is a required attribute if you are generating the form class automatically (e.g. using model). Omitting this attribute will result in an *ImproperlyConfigured* exception.

success_url
The URL to redirect to when the form is successfully processed.

success_url may contain dictionary string formatting, which will be interpolated against the object's field attributes. For example, you could use success_url="/polls/{slug}/" to redirect to a URL composed out of the slug field on a model.

get_form_class()
Retrieve the form class to instantiate. If *form_class* is provided, that class will be used. Otherwise, a ModelForm will be instantiated using the model associated with the *queryset*, or with the *model*, depending on which attribute is provided.

get_form_kwargs()
Add the current instance (self.object) to the standard *get_form_kwargs()*.

get_success_url()
Determine the URL to redirect to when the form is successfully validated. Returns *django.views.generic.edit.ModelFormMixin.success_url* if it is provided; otherwise, attempts to use the get_absolute_url() of the object.

form_valid(*form*)
Saves the form instance, sets the current object for the view, and redirects to *get_success_url()*.

form_invalid()
Renders a response, providing the invalid form as context.

ProcessFormView

class django.views.generic.edit.**ProcessFormView**
A mixin that provides basic HTTP GET and POST workflow.

: This is named 'ProcessFormView' and inherits directly from *django.views.generic.base.View*, but breaks if used independently, so it is more of a mixin.

Extends

- *django.views.generic.base.View*

Methods and Attributes

get(*request*, **args*, ***kwargs*)
Renders a response using a context created with *get_context_data()*.

Construction of the form was moved from this method to *get_context_data()*.

post(*request*, **args*, ***kwargs*)
Constructs a form, checks the form for validity, and handles it accordingly.

put(**args*, ***kwargs*)
The PUT action is also handled and just passes all parameters through to *post()*.

DeletionMixin

class django.views.generic.edit.**DeletionMixin**
Enables handling of the DELETE http action.

Methods and Attributes

success_url
The url to redirect to when the nominated object has been successfully deleted.

success_url may contain dictionary string formatting, which will be interpolated against the object's field attributes. For example, you could use success_url="/parent/{parent_id}/" to redirect to a URL composed out of the parent_id field on a model.

get_success_url()
Returns the url to redirect to when the nominated object has been successfully deleted. Returns *success_url* by default.

Date-based mixins

: All the date formatting attributes in these mixins use strftime() format characters. Do not try to use the format characters from the *now* template tag as they are not compatible.

YearMixin

class YearMixin

A mixin that can be used to retrieve and provide parsing information for a year component of a date.

Methods and Attributes

year_format

The `strftime()` format to use when parsing the year. By default, this is `'%Y'`.

year

Optional The value for the year, as a string. By default, set to `None`, which means the year will be determined using other means.

get_year_format()

Returns the `strftime()` format to use when parsing the year. Returns *year_format* by default.

get_year()

Returns the year for which this view will display data, as a string. Tries the following sources, in order:

- The value of the *YearMixin.year* attribute.
- The value of the `year` argument captured in the URL pattern.
- The value of the `year` `GET` query argument.

Raises a 404 if no valid year specification can be found.

get_next_year(*date*)

Returns a date object containing the first day of the year after the date provided. This function can also return `None` or raise an *Http404* exception, depending on the values of *allow_empty* and *allow_future*.

get_previous_year(*date*)

Returns a date object containing the first day of the year before the date provided. This function can also return `None` or raise an *Http404* exception, depending on the values of *allow_empty* and *allow_future*.

MonthMixin

class MonthMixin

A mixin that can be used to retrieve and provide parsing information for a month component of a date.

Methods and Attributes

month_format

The `strftime()` format to use when parsing the month. By default, this is `'%b'`.

month

Optional The value for the month, as a string. By default, set to `None`, which means the month will be determined using other means.

get_month_format()

Returns the `strftime()` format to use when parsing the month. Returns *month_format* by default.

get_month()

Returns the month for which this view will display data, as a string. Tries the following sources, in order:

- The value of the *MonthMixin.month* attribute.
- The value of the `month` argument captured in the URL pattern.

- The value of the `month` `GET` query argument.

Raises a 404 if no valid month specification can be found.

get_next_month(*date*)
Returns a date object containing the first day of the month after the date provided. This function can also return `None` or raise an *`Http404`* exception, depending on the values of *`allow_empty`* and *`allow_future`*.

get_previous_month(*date*)
Returns a date object containing the first day of the month before the date provided. This function can also return `None` or raise an *`Http404`* exception, depending on the values of *`allow_empty`* and *`allow_future`*.

DayMixin

class DayMixin
A mixin that can be used to retrieve and provide parsing information for a day component of a date.

Methods and Attributes

day_format
The `strftime()` format to use when parsing the day. By default, this is `'%d'`.

day
Optional The value for the day, as a string. By default, set to `None`, which means the day will be determined using other means.

get_day_format()
Returns the `strftime()` format to use when parsing the day. Returns *`day_format`* by default.

get_day()
Returns the day for which this view will display data, as a string. Tries the following sources, in order:

- The value of the *`DayMixin.day`* attribute.
- The value of the `day` argument captured in the URL pattern.
- The value of the `day` `GET` query argument.

Raises a 404 if no valid day specification can be found.

get_next_day(*date*)
Returns a date object containing the next valid day after the date provided. This function can also return `None` or raise an *`Http404`* exception, depending on the values of *`allow_empty`* and *`allow_future`*.

get_previous_day(*date*)
Returns a date object containing the previous valid day. This function can also return `None` or raise an *`Http404`* exception, depending on the values of *`allow_empty`* and *`allow_future`*.

WeekMixin

class WeekMixin
A mixin that can be used to retrieve and provide parsing information for a week component of a date.

Methods and Attributes

week_format
The `strftime()` format to use when parsing the week. By default, this is `'%U'`, which means the week starts on Sunday. Set it to `'%W'` if your week starts on Monday.

week
Optional The value for the week, as a string. By default, set to `None`, which means the week will be determined using other means.

get_week_format()
Returns the `strftime()` format to use when parsing the week. Returns *week_format* by default.

get_week()
Returns the week for which this view will display data, as a string. Tries the following sources, in order:

- The value of the *WeekMixin.week* attribute.
- The value of the `week` argument captured in the URL pattern
- The value of the `week` `GET` query argument.

Raises a 404 if no valid week specification can be found.

get_next_week(*date*)
Returns a date object containing the first day of the week after the date provided. This function can also return `None` or raise an *Http404* exception, depending on the values of *allow_empty* and *allow_future*.

get_prev_week(*date*)
Returns a date object containing the first day of the week before the date provided. This function can also return `None` or raise an *Http404* exception, depending on the values of *allow_empty* and *allow_future*.

DateMixin

class DateMixin
A mixin class providing common behavior for all date-based views.

Methods and Attributes

date_field
The name of the `DateField` or `DateTimeField` in the `QuerySet`'s model that the date-based archive should use to determine the list of objects to display on the page.

When time zone support is enabled and `date_field` is a `DateTimeField`, dates are assumed to be in the current time zone. Otherwise, the queryset could include objects from the previous or the next day in the end user's time zone.

: In this situation, if you have implemented per-user time zone selection, the same URL may show a different set of objects, depending on the end user's time zone. To avoid this, you should use a `DateField` as the `date_field` attribute.

allow_future
A boolean specifying whether to include "future" objects on this page, where "future" means objects in which the field specified in `date_field` is greater than the current date/time. By default, this is `False`.

get_date_field()
Returns the name of the field that contains the date data that this view will operate on. Returns *date_field* by default.

get_allow_future()
Determine whether to include "future" objects on this page, where "future" means objects in which the field specified in `date_field` is greater than the current date/time. Returns *allow_future* by default.

BaseDateListView

class BaseDateListView

A base class that provides common behavior for all date-based views. There won't normally be a reason to instantiate *BaseDateListView*; instantiate one of the subclasses instead.

While this view (and its subclasses) are executing, `self.object_list` will contain the list of objects that the view is operating upon, and `self.date_list` will contain the list of dates for which data is available.

Mixins

- *DateMixin*
- *MultipleObjectMixin*

Methods and Attributes

allow_empty

A boolean specifying whether to display the page if no objects are available. If this is `True` and no objects are available, the view will display an empty page instead of raising a 404.

This is identical to *django.views.generic.list.MultipleObjectMixin.allow_empty*, except for the default value, which is `False`.

date_list_period

Optional A string defining the aggregation period for `date_list`. It must be one of `'year'` (default), `'month'`, or `'day'`.

get_dated_items()

Returns a 3-tuple containing (`date_list`, `object_list`, `extra_context`).

`date_list` is the list of dates for which data is available. `object_list` is the list of objects. `extra_context` is a dictionary of context data that will be added to any context data provided by the *MultipleObjectMixin*.

get_dated_queryset(***lookup*)

Returns a queryset, filtered using the query arguments defined by `lookup`. Enforces any restrictions on the queryset, such as `allow_empty` and `allow_future`.

get_date_list_period()

Returns the aggregation period for `date_list`. Returns *date_list_period* by default.

get_date_list(*queryset*, *date_type=None*, *ordering='ASC'*)

Returns the list of dates of type `date_type` for which `queryset` contains entries. For example, `get_date_list(qs, 'year')` will return the list of years for which `qs` has entries. If `date_type` isn't provided, the result of *get_date_list_period()* is used. `date_type` and `ordering` are simply passed to *QuerySet.dates()*.

6.3.6 Class-based generic views - flattened index

This index provides an alternate organization of the reference documentation for class-based views. For each view, the effective attributes and methods from the class tree are represented under that view. For the reference documentation organized by the class which defines the behavior, see Class-based views

Simple generic views

View

class View

Attributes (with optional accessor):

- *http_method_names*

Methods

- *as_view()*
- *dispatch()*
- head()
- *http_method_not_allowed()*

TemplateView

class TemplateView

Attributes (with optional accessor):

- *content_type*
- *http_method_names*
- *response_class* [*render_to_response()*]
- *template_engine*
- *template_name* [*get_template_names()*]

Methods

- *as_view()*
- *dispatch()*
- get()
- *get_context_data()*
- head()
- *http_method_not_allowed()*
- *render_to_response()*

RedirectView

class RedirectView

Attributes (with optional accessor):

- *http_method_names*
- *pattern_name*
- *permanent*
- *query_string*

- *url* [*get_redirect_url()*]

Methods

- *as_view()*
- delete()
- *dispatch()*
- get()
- head()
- *http_method_not_allowed()*
- options()
- post()
- put()

Detail Views

DetailView

class DetailView

Attributes (with optional accessor):

- *content_type*
- *context_object_name* [*get_context_object_name()*]
- *http_method_names*
- *model*
- *pk_url_kwarg*
- *queryset* [*get_queryset()*]
- *response_class* [*render_to_response()*]
- *slug_field* [*get_slug_field()*]
- *slug_url_kwarg*
- *template_engine*
- *template_name* [*get_template_names()*]
- *template_name_field*
- *template_name_suffix*

Methods

- *as_view()*
- *dispatch()*
- get()
- *get_context_data()*
- *get_object()*

- head()
- *http_method_not_allowed()*
- *render_to_response()*

List Views

ListView

class ListView

Attributes (with optional accessor):

- *allow_empty* [*get_allow_empty()*]
- *content_type*
- *context_object_name* [*get_context_object_name()*]
- *http_method_names*
- *model*
- *ordering* [*get_ordering()*]
- *paginate_by* [*get_paginate_by()*]
- *paginate_orphans* [*get_paginate_orphans()*]
- *paginator_class*
- *queryset* [*get_queryset()*]
- *response_class* [*render_to_response()*]
- *template_engine*
- *template_name* [*get_template_names()*]
- *template_name_suffix*

Methods

- *as_view()*
- *dispatch()*
- *get()*
- *get_context_data()*
- *get_paginator()*
- head()
- *http_method_not_allowed()*
- *paginate_queryset()*
- *render_to_response()*

Editing views

FormView

class FormView

Attributes (with optional accessor):

- *content_type*
- *form_class* [*get_form_class()*]
- *http_method_names*
- *initial* [*get_initial()*]
- *prefix* [*get_prefix()*]
- *response_class* [*render_to_response()*]
- *success_url* [*get_success_url()*]
- *template_engine*
- *template_name* [*get_template_names()*]

Methods

- *as_view()*
- *dispatch()*
- *form_invalid()*
- *form_valid()*
- *get()*
- *get_context_data()*
- *get_form()*
- *get_form_kwargs()*
- *http_method_not_allowed()*
- *post()*
- *put()*

CreateView

class CreateView

Attributes (with optional accessor):

- *content_type*
- *context_object_name* [*get_context_object_name()*]
- *fields*
- *form_class* [*get_form_class()*]
- *http_method_names*
- *initial* [*get_initial()*]

- *model*
- *pk_url_kwarg*
- *prefix* [*get_prefix()*]
- *queryset* [*get_queryset()*]
- *response_class* [*render_to_response()*]
- *slug_field* [*get_slug_field()*]
- *slug_url_kwarg*
- *success_url* [*get_success_url()*]
- *template_engine*
- *template_name* [*get_template_names()*]
- *template_name_field*
- *template_name_suffix*

Methods

- *as_view()*
- *dispatch()*
- *form_invalid()*
- *form_valid()*
- *get()*
- *get_context_data()*
- *get_form()*
- *get_form_kwargs()*
- *get_object()*
- head()
- *http_method_not_allowed()*
- *post()*
- put()
- *render_to_response()*

UpdateView

class UpdateView

Attributes (with optional accessor):

- *content_type*
- *context_object_name* [*get_context_object_name()*]
- *fields*
- *form_class* [*get_form_class()*]
- *http_method_names*

- *initial* [*get_initial()*]
- *model*
- *pk_url_kwarg*
- *prefix* [*get_prefix()*]
- *queryset* [*get_queryset()*]
- *response_class* [*render_to_response()*]
- *slug_field* [*get_slug_field()*]
- *slug_url_kwarg*
- *success_url* [*get_success_url()*]
- *template_engine*
- *template_name* [*get_template_names()*]
- *template_name_field*
- *template_name_suffix*

Methods

- *as_view()*
- *dispatch()*
- *form_invalid()*
- *form_valid()*
- *get()*
- *get_context_data()*
- *get_form()*
- *get_form_kwargs()*
- *get_object()*
- head()
- *http_method_not_allowed()*
- *post()*
- put()
- *render_to_response()*

DeleteView

class DeleteView

Attributes (with optional accessor):

- *content_type*
- *context_object_name* [*get_context_object_name()*]
- *http_method_names*
- *model*

- *pk_url_kwarg*
- *queryset* [*get_queryset()*]
- *response_class* [*render_to_response()*]
- *slug_field* [*get_slug_field()*]
- *slug_url_kwarg*
- *success_url* [*get_success_url()*]
- *template_engine*
- *template_name* [*get_template_names()*]
- *template_name_field*
- *template_name_suffix*

Methods

- *as_view()*
- delete()
- *dispatch()*
- get()
- *get_context_data()*
- *get_object()*
- head()
- *http_method_not_allowed()*
- post()
- *render_to_response()*

Date-based views

ArchiveIndexView

class ArchiveIndexView

Attributes (with optional accessor):

- *allow_empty* [*get_allow_empty()*]
- *allow_future* [*get_allow_future()*]
- *content_type*
- *context_object_name* [*get_context_object_name()*]
- *date_field* [*get_date_field()*]
- *http_method_names*
- *model*
- *ordering* [*get_ordering()*]
- *paginate_by* [*get_paginate_by()*]

- *paginate_orphans* [*get_paginate_orphans()*]
- *paginator_class*
- *queryset* [*get_queryset()*]
- *response_class* [*render_to_response()*]
- *template_engine*
- *template_name* [*get_template_names()*]
- *template_name_suffix*

Methods

- *as_view()*
- *dispatch()*
- `get()`
- *get_context_data()*
- *get_date_list()*
- *get_dated_items()*
- *get_dated_queryset()*
- *get_paginator()*
- `head()`
- *http_method_not_allowed()*
- *paginate_queryset()*
- *render_to_response()*

YearArchiveView

class YearArchiveView

Attributes (with optional accessor):

- *allow_empty* [*get_allow_empty()*]
- *allow_future* [*get_allow_future()*]
- *content_type*
- *context_object_name* [*get_context_object_name()*]
- *date_field* [*get_date_field()*]
- *http_method_names*
- *make_object_list* [*get_make_object_list()*]
- *model*
- *ordering* [*get_ordering()*]
- *paginate_by* [*get_paginate_by()*]
- *paginate_orphans* [*get_paginate_orphans()*]
- *paginator_class*

- *queryset* [*get_queryset()*]
- *response_class* [*render_to_response()*]
- *template_engine*
- *template_name* [*get_template_names()*]
- *template_name_suffix*
- *year* [*get_year()*]
- *year_format* [*get_year_format()*]

Methods

- *as_view()*
- *dispatch()*
- `get()`
- *get_context_data()*
- *get_date_list()*
- *get_dated_items()*
- *get_dated_queryset()*
- *get_paginator()*
- `head()`
- *http_method_not_allowed()*
- *paginate_queryset()*
- *render_to_response()*

MonthArchiveView

class MonthArchiveView

Attributes (with optional accessor):

- *allow_empty* [*get_allow_empty()*]
- *allow_future* [*get_allow_future()*]
- *content_type*
- *context_object_name* [*get_context_object_name()*]
- *date_field* [*get_date_field()*]
- *http_method_names*
- *model*
- *month* [*get_month()*]
- *month_format* [*get_month_format()*]
- *ordering* [*get_ordering()*]
- *paginate_by* [*get_paginate_by()*]
- *paginate_orphans* [*get_paginate_orphans()*]

- *paginator_class*
- *queryset* [*get_queryset()*]
- *response_class* [*render_to_response()*]
- *template_engine*
- *template_name* [*get_template_names()*]
- *template_name_suffix*
- *year* [*get_year()*]
- *year_format* [*get_year_format()*]

Methods

- *as_view()*
- *dispatch()*
- get()
- *get_context_data()*
- *get_date_list()*
- *get_dated_items()*
- *get_dated_queryset()*
- *get_next_month()*
- *get_paginator()*
- *get_previous_month()*
- head()
- *http_method_not_allowed()*
- *paginate_queryset()*
- *render_to_response()*

WeekArchiveView

class WeekArchiveView

Attributes (with optional accessor):

- *allow_empty* [*get_allow_empty()*]
- *allow_future* [*get_allow_future()*]
- *content_type*
- *context_object_name* [*get_context_object_name()*]
- *date_field* [*get_date_field()*]
- *http_method_names*
- *model*
- *ordering* [*get_ordering()*]
- *paginate_by* [*get_paginate_by()*]

- *paginate_orphans* [*get_paginate_orphans()*]
- *paginator_class*
- *queryset* [*get_queryset()*]
- *response_class* [*render_to_response()*]
- *template_engine*
- *template_name* [*get_template_names()*]
- *template_name_suffix*
- *week* [*get_week()*]
- *week_format* [*get_week_format()*]
- *year* [*get_year()*]
- *year_format* [*get_year_format()*]

Methods

- *as_view()*
- *dispatch()*
- `get()`
- *get_context_data()*
- *get_date_list()*
- *get_dated_items()*
- *get_dated_queryset()*
- *get_paginator()*
- `head()`
- *http_method_not_allowed()*
- *paginate_queryset()*
- *render_to_response()*

DayArchiveView

class DayArchiveView

Attributes (with optional accessor):

- *allow_empty* [*get_allow_empty()*]
- *allow_future* [*get_allow_future()*]
- *content_type*
- *context_object_name* [*get_context_object_name()*]
- *date_field* [*get_date_field()*]
- *day* [*get_day()*]
- *day_format* [*get_day_format()*]
- *http_method_names*

- *model*
- *month* [*get_month()*]
- *month_format* [*get_month_format()*]
- *ordering* [*get_ordering()*]
- *paginate_by* [*get_paginate_by()*]
- *paginate_orphans* [*get_paginate_orphans()*]
- *paginator_class*
- *queryset* [*get_queryset()*]
- *response_class* [*render_to_response()*]
- *template_engine*
- *template_name* [*get_template_names()*]
- *template_name_suffix*
- *year* [*get_year()*]
- *year_format* [*get_year_format()*]

Methods

- *as_view()*
- *dispatch()*
- `get()`
- *get_context_data()*
- *get_date_list()*
- *get_dated_items()*
- *get_dated_queryset()*
- *get_next_day()*
- *get_next_month()*
- *get_paginator()*
- *get_previous_day()*
- *get_previous_month()*
- `head()`
- *http_method_not_allowed()*
- *paginate_queryset()*
- *render_to_response()*

TodayArchiveView

class TodayArchiveView

Attributes (with optional accessor):

- *allow_empty* [*get_allow_empty()*]

- *allow_future* [*get_allow_future()*]
- *content_type*
- *context_object_name* [*get_context_object_name()*]
- *date_field* [*get_date_field()*]
- *day* [*get_day()*]
- *day_format* [*get_day_format()*]
- *http_method_names*
- *model*
- *month* [*get_month()*]
- *month_format* [*get_month_format()*]
- *ordering* [*get_ordering()*]
- *paginate_by* [*get_paginate_by()*]
- *paginate_orphans* [*get_paginate_orphans()*]
- *paginator_class*
- *queryset* [*get_queryset()*]
- *response_class* [*render_to_response()*]
- *template_engine*
- *template_name* [*get_template_names()*]
- *template_name_suffix*
- *year* [*get_year()*]
- *year_format* [*get_year_format()*]

Methods

- *as_view()*
- *dispatch()*
- `get()`
- *get_context_data()*
- *get_date_list()*
- *get_dated_items()*
- *get_dated_queryset()*
- *get_next_day()*
- *get_next_month()*
- *get_paginator()*
- *get_previous_day()*
- *get_previous_month()*
- `head()`
- *http_method_not_allowed()*

- *paginate_queryset()*
- *render_to_response()*

DateDetailView

class DateDetailView

Attributes (with optional accessor):

- *allow_future* [*get_allow_future()*]
- *content_type*
- *context_object_name* [*get_context_object_name()*]
- *date_field* [*get_date_field()*]
- *day* [*get_day()*]
- *day_format* [*get_day_format()*]
- *http_method_names*
- *model*
- *month* [*get_month()*]
- *month_format* [*get_month_format()*]
- *pk_url_kwarg*
- *queryset* [*get_queryset()*]
- *response_class* [*render_to_response()*]
- *slug_field* [*get_slug_field()*]
- *slug_url_kwarg*
- *template_engine*
- *template_name* [*get_template_names()*]
- *template_name_field*
- *template_name_suffix*
- *year* [*get_year()*]
- *year_format* [*get_year_format()*]

Methods

- *as_view()*
- *dispatch()*
- get()
- *get_context_data()*
- *get_next_day()*
- *get_next_month()*
- *get_object()*
- *get_previous_day()*

- *get_previous_month()*
- head()
- *http_method_not_allowed()*
- *render_to_response()*

6.3.7 Specification

Each request served by a class-based view has an independent state; therefore, it is safe to store state variables on the instance (i.e., `self.foo = 3` is a thread-safe operation).

A class-based view is deployed into a URL pattern using the *as_view()* classmethod:

```
urlpatterns = [
    url(r'^view/$', MyView.as_view(size=42)),
]
```

Thread safety with view arguments

Arguments passed to a view are shared between every instance of a view. This means that you shouldn't use a list, dictionary, or any other mutable object as an argument to a view. If you do and the shared object is modified, the actions of one user visiting your view could have an effect on subsequent users visiting the same view.

Arguments passed into *as_view()* will be assigned onto the instance that is used to service a request. Using the previous example, this means that every request on `MyView` is able to use `self.size`. Arguments must correspond to attributes that already exist on the class (return `True` on a `hasattr` check).

6.3.8 Base vs Generic views

Base class-based views can be thought of as *parent* views, which can be used by themselves or inherited from. They may not provide all the capabilities required for projects, in which case there are Mixins which extend what base views can do.

Django's generic views are built off of those base views, and were developed as a shortcut for common usage patterns such as displaying the details of an object. They take certain common idioms and patterns found in view development and abstract them so that you can quickly write common views of data without having to repeat yourself.

Most generic views require the `queryset` key, which is a `QuerySet` instance; see Making queries for more information about `QuerySet` objects.

6.4 Clickjacking Protection

The clickjacking middleware and decorators provide easy-to-use protection against clickjacking. This type of attack occurs when a malicious site tricks a user into clicking on a concealed element of another site which they have loaded in a hidden frame or iframe.

6.4.1 An example of clickjacking

Suppose an online store has a page where a logged in user can click "Buy Now" to purchase an item. A user has chosen to stay logged into the store all the time for convenience. An attacker site might create an "I Like Ponies"

button on one of their own pages, and load the store's page in a transparent iframe such that the "Buy Now" button is invisibly overlaid on the "I Like Ponies" button. If the user visits the attacker's site, clicking "I Like Ponies" will cause an inadvertent click on the "Buy Now" button and an unknowing purchase of the item.

6.4.2 Preventing clickjacking

Modern browsers honor the X-Frame-Options HTTP header that indicates whether or not a resource is allowed to load within a frame or iframe. If the response contains the header with a value of `SAMEORIGIN` then the browser will only load the resource in a frame if the request originated from the same site. If the header is set to `DENY` then the browser will block the resource from loading in a frame no matter which site made the request.

Django provides a few simple ways to include this header in responses from your site:

1. A simple middleware that sets the header in all responses.
2. A set of view decorators that can be used to override the middleware or to only set the header for certain views.

The `X-Frame-Options` HTTP header will only be set by the middleware or view decorators if it is not already present in the response.

6.4.3 How to use it

Setting `X-Frame-Options` for all responses

To set the same `X-Frame-Options` value for all responses in your site, put `'django.middleware.clickjacking.XFrameOptionsMiddleware'` to *MIDDLEWARE*:

```
MIDDLEWARE = [
    ...
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ...
]
```

This middleware is enabled in the settings file generated by *startproject*.

By default, the middleware will set the `X-Frame-Options` header to `SAMEORIGIN` for every outgoing `HttpResponse`. If you want `DENY` instead, set the *X_FRAME_OPTIONS* setting:

```
X_FRAME_OPTIONS = 'DENY'
```

When using the middleware there may be some views where you do **not** want the `X-Frame-Options` header set. For those cases, you can use a view decorator that tells the middleware not to set the header:

```
from django.http import HttpResponse
from django.views.decorators.clickjacking import xframe_options_exempt

@xframe_options_exempt
def ok_to_load_in_a_frame(request):
    return HttpResponse("This page is safe to load in a frame on any site.")
```

Setting `X-Frame-Options` per view

To set the `X-Frame-Options` header on a per view basis, Django provides these decorators:

```
from django.http import HttpResponse
from django.views.decorators.clickjacking import xframe_options_deny
from django.views.decorators.clickjacking import xframe_options_sameorigin

@xframe_options_deny
def view_one(request):
    return HttpResponse("I won't display in any frame!")

@xframe_options_sameorigin
def view_two(request):
    return HttpResponse("Display in a frame if it's from the same origin as me.")
```

Note that you can use the decorators in conjunction with the middleware. Use of a decorator overrides the middleware.

6.4.4 Limitations

The `X-Frame-Options` header will only protect against clickjacking in a modern browser. Older browsers will quietly ignore the header and need other clickjacking prevention techniques.

Browsers that support `X-Frame-Options`

- Internet Explorer 8+
- Firefox 3.6.9+
- Opera 10.5+
- Safari 4+
- Chrome 4.1+

See also

A complete list of browsers supporting `X-Frame-Options`.

6.5 `contrib` packages

Django aims to follow Python's "batteries included" philosophy. It ships with a variety of extra, optional tools that solve common Web-development problems.

This code lives in `django/contrib` in the Django distribution. This document gives a rundown of the packages in `contrib`, along with any dependencies those packages have.

Note

For most of these add-ons – specifically, the add-ons that include either models or template tags – you'll need to add the package name (e.g., `'django.contrib.redirects'`) to your *INSTALLED_APPS* setting and re-run `manage.py migrate`.

6.5.1 The Django admin site

One of the most powerful parts of Django is the automatic admin interface. It reads metadata from your models to provide a quick, model-centric interface where trusted users can manage content on your site. The admin's recommended use is limited to an organization's internal management tool. It's not intended for building your entire front end around.

The admin has many hooks for customization, but beware of trying to use those hooks exclusively. If you need to provide a more process-centric interface that abstracts away the implementation details of database tables and fields, then it's probably time to write your own views.

In this document we discuss how to activate, use, and customize Django's admin interface.

Overview

The admin is enabled in the default project template used by *startproject*.

For reference, here are the requirements:

1. Add `'django.contrib.admin'` to your *INSTALLED_APPS* setting.
2. The admin has four dependencies - *django.contrib.auth*, *django.contrib.contenttypes*, *django.contrib.messages* and *django.contrib.sessions*. If these applications are not in your *INSTALLED_APPS* list, add them.
3. Add `django.contrib.auth.context_processors.auth` and `django.contrib.messages.context_proce` to the `'context_processors'` option of the `DjangoTemplates` backend defined in your *TEMPLATES* as well as *django.contrib.auth.middleware.AuthenticationMiddleware* and *django.contrib.messages.middleware.MessageMiddleware* to *MIDDLEWARE*. These are all active by default, so you only need to do this if you've manually tweaked the settings.
4. Determine which of your application's models should be editable in the admin interface.
5. For each of those models, optionally create a `ModelAdmin` class that encapsulates the customized admin functionality and options for that particular model.
6. Instantiate an `AdminSite` and tell it about each of your models and `ModelAdmin` classes.
7. Hook the `AdminSite` instance into your URLconf.

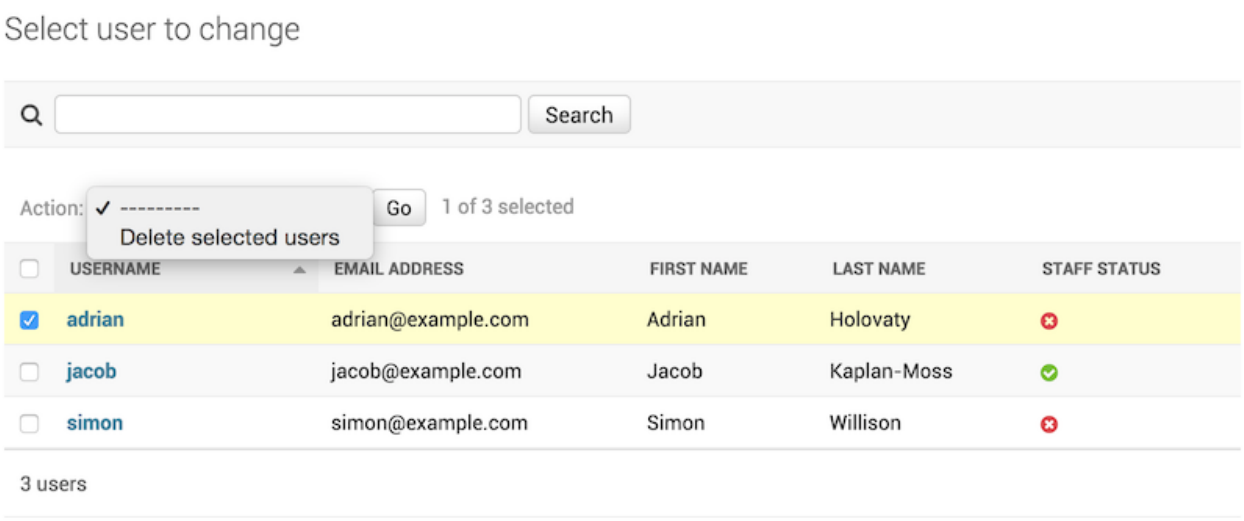
After you've taken these steps, you'll be able to use your Django admin site by visiting the URL you hooked it into (`/admin/`, by default). If you need to create a user to login with, you can use the *createsuperuser* command.

Other topics

Admin actions The basic workflow of Django's admin is, in a nutshell, "select an object, then change it." This works well for a majority of use cases. However, if you need to make the same change to many objects at once, this workflow can be quite tedious.

In these cases, Django's admin lets you write and register "actions" – simple functions that get called with a list of objects selected on the change list page.

If you look at any change list in the admin, you'll see this feature in action; Django ships with a "delete selected objects" action available to all models. For example, here's the user module from Django's built-in *django.contrib.auth* app:



: The "delete selected objects" action uses *`QuerySet.delete()`* for efficiency reasons, which has an important caveat: your model's `delete()` method will not be called.
If you wish to override this behavior, simply write a custom action which accomplishes deletion in your preferred manner – for example, by calling `Model.delete()` for each of the selected items.
For more background on bulk deletion, see the documentation on *object deletion*.

Read on to find out how to add your own actions to this list.

Writing actions The easiest way to explain actions is by example, so let's dive in.

A common use case for admin actions is the bulk updating of a model. Imagine a simple news application with an `Article` model:

```
from django.db import models

STATUS_CHOICES = (
    ('d', 'Draft'),
    ('p', 'Published'),
    ('w', 'Withdrawn'),
)

class Article(models.Model):
    title = models.CharField(max_length=100)
    body = models.TextField()
    status = models.CharField(max_length=1, choices=STATUS_CHOICES)

    def __str__(self):              # __unicode__ on Python 2
        return self.title
```

A common task we might perform with a model like this is to update an article's status from "draft" to "published". We could easily do this in the admin one article at a time, but if we wanted to bulk-publish a group of articles, it'd be tedious. So, let's write an action that lets us change an article's status to "published."

Writing action functions First, we'll need to write a function that gets called when the action is triggered from the admin. Action functions are just regular functions that take three arguments:

- The current *`ModelAdmin`*

- An *HttpRequest* representing the current request,
- A *QuerySet* containing the set of objects selected by the user.

Our publish-these-articles function won't need the *ModelAdmin* or the request object, but we will use the queryset:

```
def make_published(modeladmin, request, queryset):
    queryset.update(status='p')
```

: For the best performance, we're using the queryset's *update method*. Other types of actions might need to deal with each object individually; in these cases we'd just iterate over the queryset:

```
for obj in queryset:
    do_something_with(obj)
```

That's actually all there is to writing an action! However, we'll take one more optional-but-useful step and give the action a "nice" title in the admin. By default, this action would appear in the action list as "Make published" – the function name, with underscores replaced by spaces. That's fine, but we can provide a better, more human-friendly name by giving the `make_published` function a `short_description` attribute:

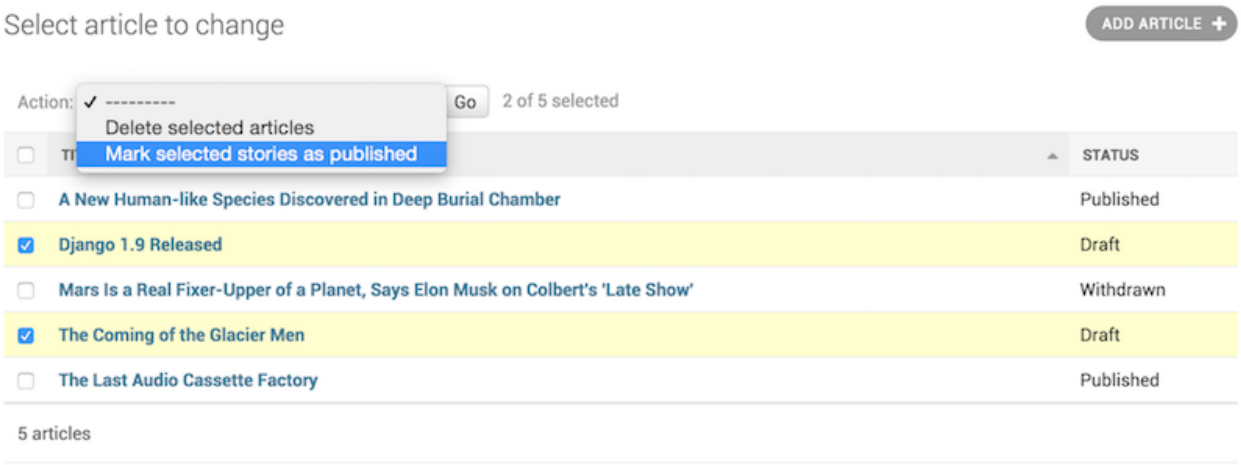
```
def make_published(modeladmin, request, queryset):
    queryset.update(status='p')
make_published.short_description = "Mark selected stories as published"
```

: This might look familiar; the admin's `list_display` option uses the same technique to provide human-readable descriptions for callback functions registered there, too.

Adding actions to the `ModelAdmin` Next, we'll need to inform our *ModelAdmin* of the action. This works just like any other configuration option. So, the complete `admin.py` with the action and its registration would look like:

```
from django.contrib import admin
from myapp.models import Article

def make_published(modeladmin, request, queryset):
    queryset.update(status='p')
make_published.short_description = "Mark selected stories as published"

class ArticleAdmin(admin.ModelAdmin):
    list_display = ['title', 'status']
    ordering = ['title']
    actions = [make_published]

admin.site.register(Article, ArticleAdmin)
```

That code will give us an admin change list that looks something like this:



That's really all there is to it! If you're itching to write your own actions, you now know enough to get started. The rest of this document just covers more advanced techniques.

Handling errors in actions If there are foreseeable error conditions that may occur while running your action, you should gracefully inform the user of the problem. This means handling exceptions and using *`django.contrib.admin.ModelAdmin.message_user()`* to display a user friendly description of the problem in the response.

Advanced action techniques There's a couple of extra options and possibilities you can exploit for more advanced options.

Actions as `ModelAdmin` methods The example above shows the `make_published` action defined as a simple function. That's perfectly fine, but it's not perfect from a code design point of view: since the action is tightly coupled to the `Article` object, it makes sense to hook the action to the `ArticleAdmin` object itself.

That's easy enough to do:

```
class ArticleAdmin(admin.ModelAdmin):
    ...

    actions = ['make_published']

    def make_published(self, request, queryset):
        queryset.update(status='p')
    make_published.short_description = "Mark selected stories as published"
```

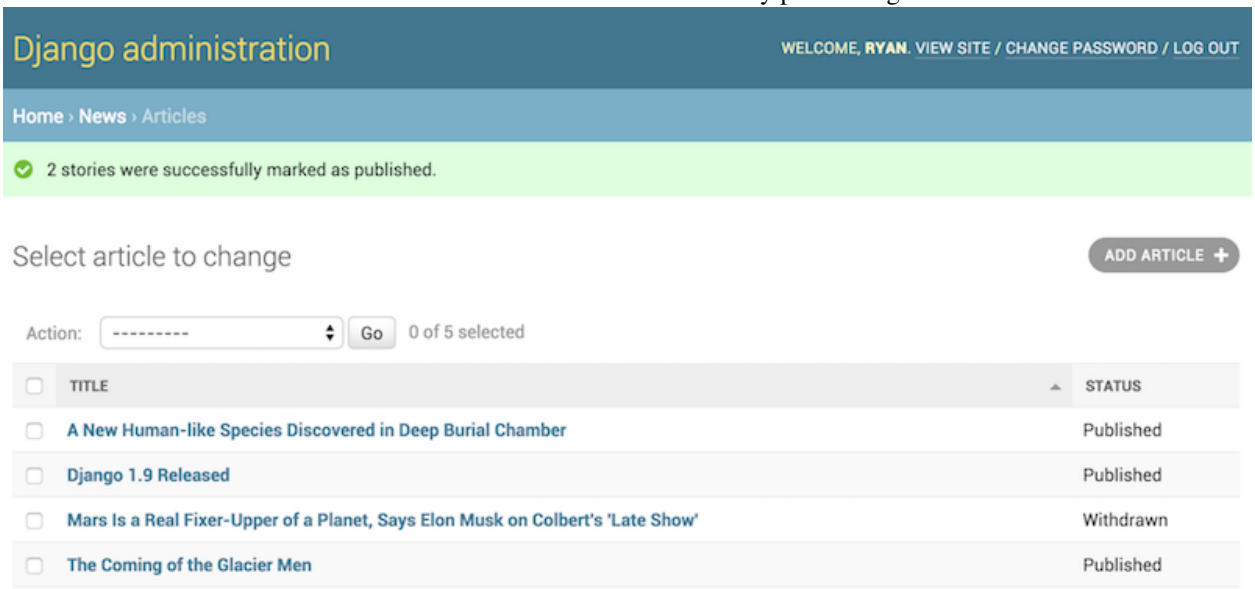
Notice first that we've moved `make_published` into a method and renamed the `modeladmin` parameter to `self`, and second that we've now put the string `'make_published'` in `actions` instead of a direct function reference. This tells the *`ModelAdmin`* to look up the action as a method.

Defining actions as methods gives the action more straightforward, idiomatic access to the *`ModelAdmin`* itself, allowing the action to call any of the methods provided by the admin. For example, we can use `self` to flash a message to the user informing her that the action was successful:

```
class ArticleAdmin(admin.ModelAdmin):
    ...

    def make_published(self, request, queryset):
```

```
        rows_updated = queryset.update(status='p')
        if rows_updated == 1:
            message_bit = "1 story was"
        else:
            message_bit = "%s stories were" % rows_updated
        self.message_user(request, "%s successfully marked as published." % message_bit)
```

This make the action match what the admin itself does after successfully performing an action:



Actions that provide intermediate pages By default, after an action is performed the user is simply redirected back to the original change list page. However, some actions, especially more complex ones, will need to return intermediate pages. For example, the built-in delete action asks for confirmation before deleting the selected objects.

To provide an intermediary page, simply return an *HttpResponse* (or subclass) from your action. For example, you might write a simple export function that uses Django's serialization functions to dump some selected objects as JSON:

```
from django.http import HttpResponse
from django.core import serializers

def export_as_json(modeladmin, request, queryset):
    response = HttpResponse(content_type="application/json")
    serializers.serialize("json", queryset, stream=response)
    return response
```

Generally, something like the above isn't considered a great idea. Most of the time, the best practice will be to return an *HttpResponseRedirect* and redirect the user to a view you've written, passing the list of selected objects in the GET query string. This allows you to provide complex interaction logic on the intermediary pages. For example, if you wanted to provide a more complete export function, you'd want to let the user choose a format, and possibly a list of fields to include in the export. The best thing to do would be to write a small action that simply redirects to your custom export view:

```
from django.contrib import admin
from django.contrib.contenttypes.models import ContentType
from django.http import HttpResponseRedirect

def export_selected_objects(modeladmin, request, queryset):
```

```
    selected = request.POST.getlist(admin.ACTION_CHECKBOX_NAME)
    ct = ContentType.objects.get_for_model(queryset.model)
    return HttpResponseRedirect("/export/?ct=%s&ids=%s" % (ct.pk, ",".join(selected)))
```

As you can see, the action is the simple part; all the complex logic would belong in your export view. This would need to deal with objects of any type, hence the business with the `ContentType`.

Writing this view is left as an exercise to the reader.

Making actions available site-wide

`AdminSite.`**`add_action`**`(`*action*, *name=None*`)`

Some actions are best if they're made available to *any* object in the admin site – the export action defined above would be a good candidate. You can make an action globally available using *`AdminSite.add_action()`*. For example:

```
    from django.contrib import admin

    admin.site.add_action(export_selected_objects)
```

This makes the `export_selected_objects` action globally available as an action named "export_selected_objects". You can explicitly give the action a name – good if you later want to programmatically *remove the action* – by passing a second argument to *`AdminSite.add_action()`*:

```
    admin.site.add_action(export_selected_objects, 'export_selected')
```

Disabling actions Sometimes you need to disable certain actions – especially those *registered site-wide* – for particular objects. There's a few ways you can disable actions:

Disabling a site-wide action

`AdminSite.`**`disable_action`**`(`*name*`)`

If you need to disable a *site-wide action* you can call *`AdminSite.disable_action()`*.

For example, you can use this method to remove the built-in "delete selected objects" action:

```
    admin.site.disable_action('delete_selected')
```

Once you've done the above, that action will no longer be available site-wide.

If, however, you need to re-enable a globally-disabled action for one particular model, simply list it explicitly in your `ModelAdmin.actions` list:

```
    # Globally disable delete selected
    admin.site.disable_action('delete_selected')

    # This ModelAdmin will not have delete_selected available
    class SomeModelAdmin(admin.ModelAdmin):
        actions = ['some_other_action']
        ...

    # This one will
    class AnotherModelAdmin(admin.ModelAdmin):
        actions = ['delete_selected', 'a_third_action']
        ...
```

Disabling all actions for a particular `ModelAdmin` If you want *no* bulk actions available for a given *`ModelAdmin`*, simply set *`ModelAdmin.actions`* to `None`:

```
class MyModelAdmin(admin.ModelAdmin):
    actions = None
```

This tells the *`ModelAdmin`* to not display or allow any actions, including any *site-wide actions*.

Conditionally enabling or disabling actions

`ModelAdmin.`**`get_actions`**`(`*`request`*`)`

Finally, you can conditionally enable or disable actions on a per-request (and hence per-user basis) by overriding *`ModelAdmin.get_actions()`*.

This returns a dictionary of actions allowed. The keys are action names, and the values are `(function, name, short_description)` tuples.

Most of the time you'll use this method to conditionally remove actions from the list gathered by the superclass. For example, if I only wanted users whose names begin with 'J' to be able to delete objects in bulk, I could do the following:

```
class MyModelAdmin(admin.ModelAdmin):
    ...

    def get_actions(self, request):
        actions = super(MyModelAdmin, self).get_actions(request)
        if request.user.username[0].upper() != 'J':
            if 'delete_selected' in actions:
                del actions['delete_selected']
        return actions
```

The Django admin documentation generator Django's *`admindocs`* app pulls documentation from the docstrings of models, views, template tags, and template filters for any app in *`INSTALLED_APPS`* and makes that documentation available from the *`Django admin`*.

Overview To activate the *`admindocs`*, you will need to do the following:

- Add *`django.contrib.admindocs`* to your *`INSTALLED_APPS`*.
- Add `url(r'^admin/doc/', include('django.contrib.admindocs.urls'))` to your `urlpatterns`. Make sure it's included *before* the `r'^admin/'` entry, so that requests to `/admin/doc/` don't get handled by the latter entry.
- Install the docutils Python module (http://docutils.sf.net/).
- **Optional:** Using the admindocs bookmarklets requires `django.contrib.admindocs.middleware.XViewMiddlewar` to be installed.

Once those steps are complete, you can start browsing the documentation by going to your admin interface and clicking the "Documentation" link in the upper right of the page.

Documentation helpers The following special markup can be used in your docstrings to easily create hyperlinks to other components:

Django Component	reStructuredText roles
Models	`:model:'app_label.ModelName'`
Views	`:view:'app_label.view_name'`
Template tags	`:tag:'tagname'`
Template filters	`:filter:'filtername'`
Templates	`:template:'path/to/template.html'`

Model reference The **models** section of the `admindocs` page describes each model in the system along with all the fields and methods available on it. Relationships to other models appear as hyperlinks. Descriptions are pulled from `help_text` attributes on fields or from docstrings on model methods.

The **models** section of the `admindocs` now describes methods that take arguments as well. In previous versions it was restricted to methods without arguments.

A model with useful documentation might look like this:

```
class BlogEntry(models.Model):
    """
    Stores a single blog entry, related to :model:`blog.Blog` and
    :model:`auth.User`.
    """
    slug = models.SlugField(help_text="A short label, generally used in URLs.")
    author = models.ForeignKey(
        User,
        models.SET_NULL,
        blank=True, null=True,
    )
    blog = models.ForeignKey(Blog, models.CASCADE)
    ...

    def publish(self):
        """Makes the blog entry live on the site."""
        ...
```

View reference Each URL in your site has a separate entry in the `admindocs` page, and clicking on a given URL will show you the corresponding view. Helpful things you can document in your view function docstrings include:

- A short description of what the view does.
- The **context**, or a list of variables available in the view's template.
- The name of the template or templates that are used for that view.

For example:

```
from django.shortcuts import render

from myapp.models import MyModel

def my_view(request, slug):
    """
    Display an individual :model:`myapp.MyModel`.

    **Context**

    ``mymodel``
        An instance of :model:`myapp.MyModel`.
```

```
    **Template:**

    :template:`myapp/my_template.html`
    """
    context = {'mymodel': MyModel.objects.get(slug=slug)}
    return render(request, 'myapp/my_template.html', context)
```

Template tags and filters reference The **tags** and **filters** `admindocs` sections describe all the tags and filters that come with Django (in fact, the *built-in tag reference* and *built-in filter reference* documentation come directly from those pages). Any tags or filters that you create or are added by a third-party app will show up in these sections as well.

Template reference While `admindocs` does not include a place to document templates by themselves, if you use the `:template:`path/to/template.html`` syntax in a docstring the resulting page will verify the path of that template with Django's *template loaders*. This can be a handy way to check if the specified template exists and to show where on the filesystem that template is stored.

Included Bookmarklets One bookmarklet is available from the `admindocs` page:

Documentation for this page Jumps you from any page to the documentation for the view that generates that page.

Using this bookmarklet requires that `XViewMiddleware` is installed and that you are logged into the *Django admin* as a *User* with *is_staff* set to `True`.

JavaScript customizations in the admin

Inline form events You may want to execute some JavaScript when an inline form is added or removed in the admin change form. The `formset:added` and `formset:removed` jQuery events allow this. The event handler is passed three arguments:

- `event` is the `jQuery` event.
- `$row` is the newly added (or removed) row.
- `formsetName` is the formset the row belongs to.

The event is fired using the *django.jQuery namespace*.

In your custom `change_form.html` template, extend the `admin_change_form_document_ready` block and add the event listener code:

```
{% extends 'admin/change_form.html' %}
{% load static %}

{% block admin_change_form_document_ready %}
{{ block.super }}
<script type="text/javascript" src="{% static 'app/formset_handlers.js' %}"></script>
{% endblock %}
```

`app/static/app/formset_handlers.js`

```
(function($) {
    $(document).on('formset:added', function(event, $row, formsetName) {
        if (formsetName == 'author_set') {
            // Do something
        }
```

```
    });

    $(document).on('formset:removed', function(event, $row, formsetName) {
        // Row removed
    });
})(django.jQuery);
```

Two points to keep in mind:

- The JavaScript code must go in a template block if you are inheriting `admin/change_form.html` or it won't be rendered in the final HTML.
- `{{ block.super }}` is added because Django's `admin_change_form_document_ready` block contains JavaScript code to handle various operations in the change form and we need that to be rendered too.

Sometimes you'll need to work with `jQuery` plugins that are not registered in the `django.jQuery` namespace. To do that, simply change how the code listens for events. Instead of wrapping the listener in the `django.jQuery` namespace, just listen to the event triggered from there. For example:

```
{% extends 'admin/change_form.html' %}
{% load static %}

{% block admin_change_form_document_ready %}
{{ block.super }}
<script type="text/javascript" src="{% static 'app/unregistered_handlers.js' %}"></script>
{% endblock %}
```

app/static/app/unregistered_handlers.js

```
django.jQuery(document).on('formset:added', function(event, $row, formsetName) {
    // Row added
});

django.jQuery(document).on('formset:removed', function(event, $row, formsetName) {
    // Row removed
});
```

:

For information about serving the static files (images, JavaScript, and CSS) associated with the admin in production, see *Serving files*.

Having problems? Try FAQ: The admin.

`ModelAdmin` objects

class `ModelAdmin`

The `ModelAdmin` class is the representation of a model in the admin interface. Usually, these are stored in a file named `admin.py` in your application. Let's take a look at a very simple example of the `ModelAdmin`:

```
    from django.contrib import admin
    from myproject.myapp.models import Author

    class AuthorAdmin(admin.ModelAdmin):
        pass
    admin.site.register(Author, AuthorAdmin)
```

Do you need a `ModelAdmin` object at all?

In the preceding example, the `ModelAdmin` class doesn't define any custom values (yet). As a result, the default admin interface will be provided. If you are happy with the default admin interface, you don't need to define a `ModelAdmin` object at all – you can register the model class without providing a `ModelAdmin` description. The preceding example could be simplified to:

```
from django.contrib import admin
from myproject.myapp.models import Author

admin.site.register(Author)
```

The `register` decorator

register(**models*, *site=django.admin.sites.site*)

There is also a decorator for registering your `ModelAdmin` classes:

```
from django.contrib import admin
from .models import Author

@admin.register(Author)
class AuthorAdmin(admin.ModelAdmin):
    pass
```

It is given one or more model classes to register with the `ModelAdmin` and an optional keyword argument `site` if you are not using the default `AdminSite`:

```
from django.contrib import admin
from .models import Author, Reader, Editor
from myproject.admin_site import custom_admin_site

@admin.register(Author, Reader, Editor, site=custom_admin_site)
class PersonAdmin(admin.ModelAdmin):
    pass
```

You can't use this decorator if you have to reference your model admin class in its `__init__()` method, e.g. `super(PersonAdmin, self).__init__(*args, **kwargs)`. If you are using Python 3 and don't have to worry about supporting Python 2, you can use `super().__init__(*args, **kwargs)`. Otherwise, you'll have to use `admin.site.register()` instead of this decorator.

Discovery of admin files

When you put `'django.contrib.admin'` in your *INSTALLED_APPS* setting, Django automatically looks for an `admin` module in each application and imports it.

class `apps`.**AdminConfig**

This is the default *AppConfig* class for the admin. It calls *autodiscover()* when Django starts.

class `apps`.**SimpleAdminConfig**

This class works like *AdminConfig*, except it doesn't call *autodiscover()*.

autodiscover()

This function attempts to import an `admin` module in each installed application. Such modules are expected to register models with the admin.

Typically you won't need to call this function directly as *AdminConfig* calls it when Django starts.

If you are using a custom `AdminSite`, it is common to import all of the `ModelAdmin` subclasses into your code and register them to the custom `AdminSite`. In that case, in order to disable auto-discovery, you should put `'django.contrib.admin.apps.SimpleAdminConfig'` instead of `'django.contrib.admin'` in your *INSTALLED_APPS* setting.

`ModelAdmin` options

The `ModelAdmin` is very flexible. It has several options for dealing with customizing the interface. All options are defined on the `ModelAdmin` subclass:

```
from django.contrib import admin

class AuthorAdmin(admin.ModelAdmin):
    date_hierarchy = 'pub_date'
```

`ModelAdmin.`**`actions`**

A list of actions to make available on the change list page. See Admin actions for details.

`ModelAdmin.`**`actions_on_top`**

`ModelAdmin.`**`actions_on_bottom`**

Controls where on the page the actions bar appears. By default, the admin changelist displays actions at the top of the page (`actions_on_top = True; actions_on_bottom = False`).

`ModelAdmin.`**`actions_selection_counter`**

Controls whether a selection counter is displayed next to the action dropdown. By default, the admin changelist will display it (`actions_selection_counter = True`).

`ModelAdmin.`**`date_hierarchy`**

Set `date_hierarchy` to the name of a `DateField` or `DateTimeField` in your model, and the change list page will include a date-based drilldown navigation by that field.

Example:

```
    date_hierarchy = 'pub_date'
```

This will intelligently populate itself based on available data, e.g. if all the dates are in one month, it'll show the day-level drill-down only.

: `date_hierarchy` uses *QuerySet.datetimes()* internally. Please refer to its documentation for some caveats when time zone support is enabled (*USE_TZ = True*).

`ModelAdmin.`**`empty_value_display`**

This attribute overrides the default display value for record's fields that are empty (`None`, empty string, etc.). The default value is – (a dash). For example:

```
    from django.contrib import admin

    class AuthorAdmin(admin.ModelAdmin):
        empty_value_display = '-empty-'
```

You can also override `empty_value_display` for all admin pages with *AdminSite.empty_value_display*, or for specific fields like this:

```
    from django.contrib import admin

    class AuthorAdmin(admin.ModelAdmin):
        fields = ('name', 'title', 'view_birth_date')

        def view_birth_date(self, obj):
            return obj.birth_date

        view_birth_date.empty_value_display = '???'
```

ModelAdmin.**exclude**

This attribute, if given, should be a list of field names to exclude from the form.

For example, let's consider the following model:

```
    from django.db import models

    class Author(models.Model):
        name = models.CharField(max_length=100)
        title = models.CharField(max_length=3)
        birth_date = models.DateField(blank=True, null=True)
```

If you want a form for the `Author` model that includes only the `name` and `title` fields, you would specify `fields` or `exclude` like this:

```
    from django.contrib import admin

    class AuthorAdmin(admin.ModelAdmin):
        fields = ('name', 'title')

    class AuthorAdmin(admin.ModelAdmin):
        exclude = ('birth_date',)
```

Since the Author model only has three fields, `name`, `title`, and `birth_date`, the forms resulting from the above declarations will contain exactly the same fields.

ModelAdmin.**fields**

Use the `fields` option to make simple layout changes in the forms on the "add" and "change" pages such as showing only a subset of available fields, modifying their order, or grouping them into rows. For example, you could define a simpler version of the admin form for the *django.contrib.flatpages.models.FlatPage* model as follows:

```
    class FlatPageAdmin(admin.ModelAdmin):
        fields = ('url', 'title', 'content')
```

In the above example, only the fields `url`, `title` and `content` will be displayed, sequentially, in the form. `fields` can contain values defined in *ModelAdmin.readonly_fields* to be displayed as read-only.

For more complex layout needs, see the *fieldsets* option.

The `fields` option, unlike *list_display*, may only contain names of fields on the model or the form specified by *form*. It may contain callables only if they are listed in *readonly_fields*.

To display multiple fields on the same line, wrap those fields in their own tuple. In this example, the `url` and `title` fields will display on the same line and the `content` field will be displayed below them on its own line:

```
    class FlatPageAdmin(admin.ModelAdmin):
        fields = (('url', 'title'), 'content')
```

Note

This `fields` option should not be confused with the `fields` dictionary key that is within the *`fieldsets`* option, as described in the next section.

If neither `fields` nor *`fieldsets`* options are present, Django will default to displaying each field that isn't an `AutoField` and has `editable=True`, in a single fieldset, in the same order as the fields are defined in the model.

`ModelAdmin.`**`fieldsets`**

Set `fieldsets` to control the layout of admin "add" and "change" pages.

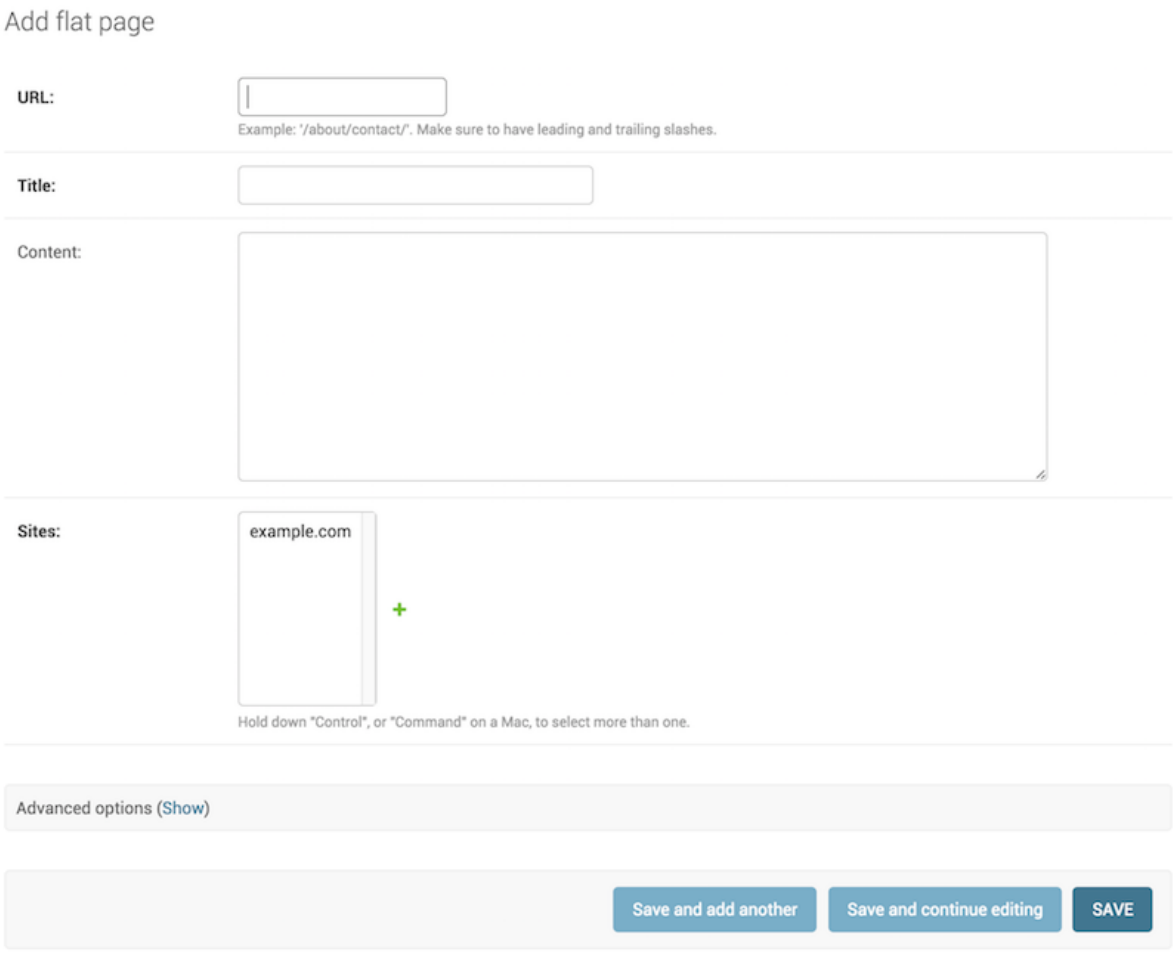
`fieldsets` is a list of two-tuples, in which each two-tuple represents a `<fieldset>` on the admin form page. (A `<fieldset>` is a "section" of the form.)

The two-tuples are in the format `(name, field_options)`, where `name` is a string representing the title of the fieldset and `field_options` is a dictionary of information about the fieldset, including a list of fields to be displayed in it.

A full example, taken from the *`django.contrib.flatpages.models.FlatPage`* model:

```
from django.contrib import admin

class FlatPageAdmin(admin.ModelAdmin):
    fieldsets = (
        (None, {
            'fields': ('url', 'title', 'content', 'sites')
        }),
        ('Advanced options', {
            'classes': ('collapse',),
            'fields': ('registration_required', 'template_name'),
        }),
    )
```

This results in an admin page that looks like:



If neither `fieldsets` nor *`fields`* options are present, Django will default to displaying each field that isn't an `AutoField` and has `editable=True`, in a single fieldset, in the same order as the fields are defined in the model.

The `field_options` dictionary can have the following keys:

- **`fields`** A tuple of field names to display in this fieldset. This key is required.

  Example:

```
{
'fields': ('first_name', 'last_name', 'address', 'city', 'state'),
}
```

  As with the *`fields`* option, to display multiple fields on the same line, wrap those fields in their own tuple. In this example, the `first_name` and `last_name` fields will display on the same line:

```
{
'fields': (('first_name', 'last_name'), 'address', 'city', 'state'),
}
```

  `fields` can contain values defined in *`readonly_fields`* to be displayed as read-only.

  If you add the name of a callable to `fields`, the same rule applies as with the *`fields`* option: the callable must be listed in *`readonly_fields`*.

- **classes** A list or tuple containing extra CSS classes to apply to the fieldset.

  Example:

```
{
'classes': ('wide', 'extrapretty'),
}
```

  Two useful classes defined by the default admin site stylesheet are `collapse` and `wide`. Fieldsets with the `collapse` style will be initially collapsed in the admin and replaced with a small "click to expand" link. Fieldsets with the `wide` style will be given extra horizontal space.

- **description** A string of optional extra text to be displayed at the top of each fieldset, under the heading of the fieldset. This string is not rendered for *`TabularInline`* due to its layout.

  Note that this value is *not* HTML-escaped when it's displayed in the admin interface. This lets you include HTML if you so desire. Alternatively you can use plain text and `django.utils.html.escape()` to escape any HTML special characters.

`ModelAdmin.`**`filter_horizontal`**

By default, a *`ManyToManyField`* is displayed in the admin site with a `<select multiple>`. However, multiple-select boxes can be difficult to use when selecting many items. Adding a *`ManyToManyField`* to this list will instead use a nifty unobtrusive JavaScript "filter" interface that allows searching within the options. The unselected and selected options appear in two boxes side by side. See *`filter_vertical`* to use a vertical interface.

`ModelAdmin.`**`filter_vertical`**

Same as *`filter_horizontal`*, but uses a vertical display of the filter interface with the box of unselected options appearing above the box of selected options.

`ModelAdmin.`**`form`**

By default a `ModelForm` is dynamically created for your model. It is used to create the form presented on both the add/change pages. You can easily provide your own `ModelForm` to override any default form behavior on the add/change pages. Alternatively, you can customize the default form rather than specifying an entirely new one by using the *`ModelAdmin.get_form()`* method.

For an example see the section *Adding custom validation to the admin*.

Note

If you define the `Meta.model` attribute on a *`ModelForm`*, you must also define the `Meta.fields` attribute (or the `Meta.exclude` attribute). However, since the admin has its own way of defining fields, the `Meta.fields` attribute will be ignored.

If the `ModelForm` is only going to be used for the admin, the easiest solution is to omit the `Meta.model` attribute, since `ModelAdmin` will provide the correct model to use. Alternatively, you can set `fields = []` in the `Meta` class to satisfy the validation on the `ModelForm`.

Note

If your `ModelForm` and `ModelAdmin` both define an `exclude` option then `ModelAdmin` takes precedence:

```
from django import forms
from django.contrib import admin
from myapp.models import Person

class PersonForm(forms.ModelForm):
```

```
        class Meta:
            model = Person
            exclude = ['name']

    class PersonAdmin(admin.ModelAdmin):
        exclude = ['age']
        form = PersonForm
```

In the above example, the "age" field will be excluded but the "name" field will be included in the generated form.

ModelAdmin.**formfield_overrides**

This provides a quick-and-dirty way to override some of the *Field* options for use in the admin. `formfield_overrides` is a dictionary mapping a field class to a dict of arguments to pass to the field at construction time.

Since that's a bit abstract, let's look at a concrete example. The most common use of `formfield_overrides` is to add a custom widget for a certain type of field. So, imagine we've written a `RichTextEditorWidget` that we'd like to use for large text fields instead of the default `<textarea>`. Here's how we'd do that:

```
    from django.db import models
    from django.contrib import admin

    # Import our custom widget and our model from where they're defined
    from myapp.widgets import RichTextEditorWidget
    from myapp.models import MyModel

    class MyModelAdmin(admin.ModelAdmin):
        formfield_overrides = {
            models.TextField: {'widget': RichTextEditorWidget},
        }
```

Note that the key in the dictionary is the actual field class, *not* a string. The value is another dictionary; these arguments will be passed to the form field's `__init__()` method. See The Forms API for details.

: If you want to use a custom widget with a relation field (i.e. *ForeignKey* or *ManyToManyField*), make sure you haven't included that field's name in `raw_id_fields` or `radio_fields`. `formfield_overrides` won't let you change the widget on relation fields that have `raw_id_fields` or `radio_fields` set. That's because `raw_id_fields` and `radio_fields` imply custom widgets of their own.

ModelAdmin.**inlines**

See *InlineModelAdmin* objects below as well as *ModelAdmin.get_formsets_with_inlines()*.

ModelAdmin.**list_display**

Set `list_display` to control which fields are displayed on the change list page of the admin.

Example:

```
    list_display = ('first_name', 'last_name')
```

If you don't set `list_display`, the admin site will display a single column that displays the `__str__()` (`__unicode__()` on Python 2) representation of each object.

You have four possible values that can be used in `list_display`:

- A field of the model. For example:

```
class PersonAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name')
```

- A callable that accepts one parameter for the model instance. For example:

```
def upper_case_name(obj):
    return ("%s %s" % (obj.first_name, obj.last_name)).upper()
upper_case_name.short_description = 'Name'

class PersonAdmin(admin.ModelAdmin):
    list_display = (upper_case_name,)
```

- A string representing an attribute on the `ModelAdmin`. This behaves same as the callable. For example:

```
class PersonAdmin(admin.ModelAdmin):
    list_display = ('upper_case_name',)

    def upper_case_name(self, obj):
        return ("%s %s" % (obj.first_name, obj.last_name)).upper()
    upper_case_name.short_description = 'Name'
```

- A string representing an attribute on the model. This behaves almost the same as the callable, but `self` in this context is the model instance. Here's a full model example:

```
from django.db import models
from django.contrib import admin

class Person(models.Model):
    name = models.CharField(max_length=50)
    birthday = models.DateField()

    def decade_born_in(self):
        return self.birthday.strftime('%Y')[:3] + "0's"
    decade_born_in.short_description = 'Birth decade'

class PersonAdmin(admin.ModelAdmin):
    list_display = ('name', 'decade_born_in')
```

A few special cases to note about `list_display`:

- If the field is a `ForeignKey`, Django will display the `__str__()` (`__unicode__()` on Python 2) of the related object.
- `ManyToManyField` fields aren't supported, because that would entail executing a separate SQL statement for each row in the table. If you want to do this nonetheless, give your model a custom method, and add that method's name to `list_display`. (See below for more on custom methods in `list_display`.)
- If the field is a `BooleanField` or `NullBooleanField`, Django will display a pretty "on" or "off" icon instead of `True` or `False`.
- If the string given is a method of the model, `ModelAdmin` or a callable, Django will HTML-escape the output by default. To escape user input and allow your own unescaped tags, use *`format_html()`*.

  Here's a full example model:

```
from django.db import models
from django.contrib import admin
from django.utils.html import format_html
```

```
class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    color_code = models.CharField(max_length=6)

    def colored_name(self):
        return format_html(
            '<span style="color: #{};">{} {}</span>',
            self.color_code,
            self.first_name,
            self.last_name,
        )

class PersonAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'colored_name')
```

1.9 : In older versions, you could add an `allow_tags` attribute to the method to prevent auto-escaping. This attribute is deprecated as it's safer to use *`format_html()`*, *`format_html_join()`*, or *`mark_safe()`* instead.

•If the value of a field is `None`, an empty string, or an iterable without elements, Django will display – (a dash). You can override this with *`AdminSite.empty_value_display`*:

```
from django.contrib import admin

admin.site.empty_value_display = '(None)'
```

You can also use *`ModelAdmin.empty_value_display`*:

```
class PersonAdmin(admin.ModelAdmin):
    empty_value_display = 'unknown'
```

Or on a field level:

```
class PersonAdmin(admin.ModelAdmin):
    list_display = ('name', 'birth_date_view')

    def birth_date_view(self, obj):
         return obj.birth_date

    birth_date_view.empty_value_display = 'unknown'
```

The ability to customize `empty_value_display` was added.

•If the string given is a method of the model, `ModelAdmin` or a callable that returns True or False Django will display a pretty "on" or "off" icon if you give the method a `boolean` attribute whose value is `True`.

Here's a full example model:

```
from django.db import models
from django.contrib import admin

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    birthday = models.DateField()

    def born_in_fifties(self):
        return self.birthday.strftime('%Y')[:3] == '195'
    born_in_fifties.boolean = True
```

```
class PersonAdmin(admin.ModelAdmin):
    list_display = ('name', 'born_in_fifties')
```

- The `__str__()` (`__unicode__()` on Python 2) method is just as valid in `list_display` as any other model method, so it's perfectly OK to do this:

```
list_display = ('__str__', 'some_other_field')
```

- Usually, elements of `list_display` that aren't actual database fields can't be used in sorting (because Django does all the sorting at the database level).

  However, if an element of `list_display` represents a certain database field, you can indicate this fact by setting the `admin_order_field` attribute of the item.

  For example:

```
from django.db import models
from django.contrib import admin
from django.utils.html import format_html

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    color_code = models.CharField(max_length=6)

    def colored_first_name(self):
        return format_html(
            '<span style="color: #{};">{}</span>',
            self.color_code,
            self.first_name,
        )

    colored_first_name.admin_order_field = 'first_name'

class PersonAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'colored_first_name')
```

  The above will tell Django to order by the `first_name` field when trying to sort by `colored_first_name` in the admin.

  To indicate descending order with `admin_order_field` you can use a hyphen prefix on the field name. Using the above example, this would look like:

```
colored_first_name.admin_order_field = '-first_name'
```

  `admin_order_field` supports query lookups to sort by values on related models. This example includes an "author first name" column in the list display and allows sorting it by first name:

```
class Blog(models.Model):
    title = models.CharField(max_length=255)
    author = models.ForeignKey(Person, on_delete=models.CASCADE)

class BlogAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'author_first_name')

    def author_first_name(self, obj):
        return obj.author.first_name

    author_first_name.admin_order_field = 'author__first_name'
```

- Elements of `list_display` can also be properties. Please note however, that due to the way properties work in Python, setting `short_description` on a property is only possible when using the `property()` function and **not** with the `@property` decorator.

  For example:

```
class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)

    def my_property(self):
        return self.first_name + ' ' + self.last_name
    my_property.short_description = "Full name of the person"

    full_name = property(my_property)

class PersonAdmin(admin.ModelAdmin):
    list_display = ('full_name',)
```

- The field names in `list_display` will also appear as CSS classes in the HTML output, in the form of `column-<field_name>` on each `<th>` element. This can be used to set column widths in a CSS file for example.
- Django will try to interpret every element of `list_display` in this order:
  - A field of the model.
  - A callable.
  - A string representing a `ModelAdmin` attribute.
  - A string representing a model attribute.

  For example if you have `first_name` as a model field and as a `ModelAdmin` attribute, the model field will be used.

`ModelAdmin.`**`list_display_links`**

Use `list_display_links` to control if and which fields in *`list_display`* should be linked to the "change" page for an object.

By default, the change list page will link the first column – the first field specified in `list_display` – to the change page for each item. But `list_display_links` lets you change this:

- Set it to `None` to get no links at all.
- Set it to a list or tuple of fields (in the same format as `list_display`) whose columns you want converted to links.

  You can specify one or many fields. As long as the fields appear in `list_display`, Django doesn't care how many (or how few) fields are linked. The only requirement is that if you want to use `list_display_links` in this fashion, you must define `list_display`.

In this example, the `first_name` and `last_name` fields will be linked on the change list page:

```
class PersonAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'birthday')
    list_display_links = ('first_name', 'last_name')
```

In this example, the change list page grid will have no links:

```
class AuditEntryAdmin(admin.ModelAdmin):
    list_display = ('timestamp', 'message')
    list_display_links = None
```

ModelAdmin.**list_editable**

Set `list_editable` to a list of field names on the model which will allow editing on the change list page. That is, fields listed in `list_editable` will be displayed as form widgets on the change list page, allowing users to edit and save multiple rows at once.

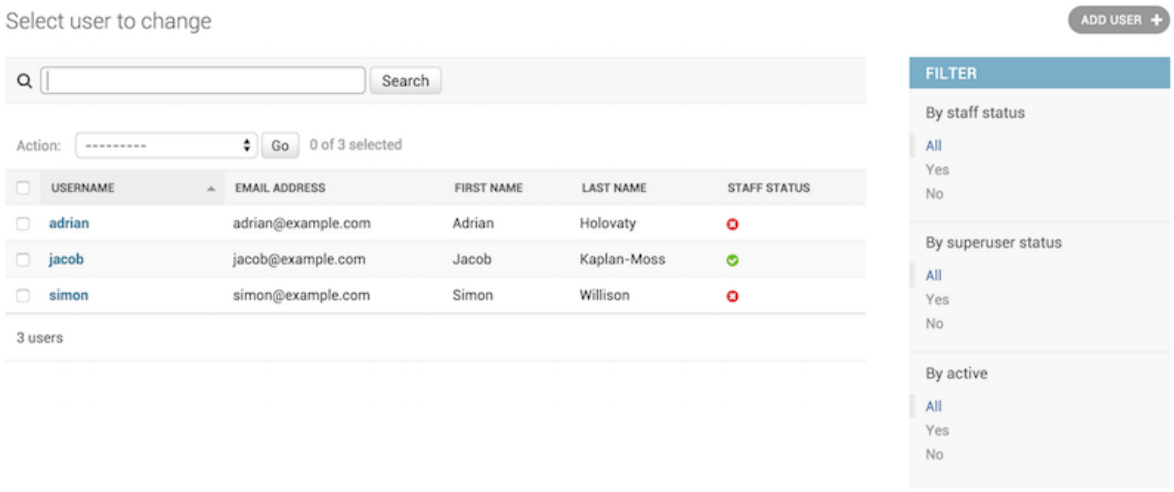
: `list_editable` interacts with a couple of other options in particular ways; you should note the following rules:

- Any field in `list_editable` must also be in `list_display`. You can't edit a field that's not displayed!
- The same field can't be listed in both `list_editable` and `list_display_links` – a field can't be both a form and a link.

You'll get a validation error if either of these rules are broken.

ModelAdmin.**list_filter**

Set `list_filter` to activate filters in the right sidebar of the change list page of the admin, as illustrated in the following screenshot:



`list_filter` should be a list or tuple of elements, where each element should be of one of the following types:

- a field name, where the specified field should be either a `BooleanField`, `CharField`, `DateField`, `DateTimeField`, `IntegerField`, `ForeignKey` or `ManyToManyField`, for example:

```
class PersonAdmin(admin.ModelAdmin):
    list_filter = ('is_staff', 'company')
```

Field names in `list_filter` can also span relations using the __ lookup, for example:

```
class PersonAdmin(admin.UserAdmin):
    list_filter = ('company__name',)
```

- a class inheriting from `django.contrib.admin.SimpleListFilter`, which you need to provide the `title` and `parameter_name` attributes to and override the `lookups` and `queryset` methods, e.g.:

```
from datetime import date
```

```
from django.contrib import admin
from django.utils.translation import ugettext_lazy as _

class DecadeBornListFilter(admin.SimpleListFilter):
    # Human-readable title which will be displayed in the
    # right admin sidebar just above the filter options.
    title = _('decade born')

    # Parameter for the filter that will be used in the URL query.
    parameter_name = 'decade'

    def lookups(self, request, model_admin):
        """
        Returns a list of tuples. The first element in each
        tuple is the coded value for the option that will
        appear in the URL query. The second element is the
        human-readable name for the option that will appear
        in the right sidebar.
        """
        return (
            ('80s', _('in the eighties')),
            ('90s', _('in the nineties')),
        )

    def queryset(self, request, queryset):
        """
        Returns the filtered queryset based on the value
        provided in the query string and retrievable via
        `self.value()`.
        """
        # Compare the requested value (either '80s' or '90s')
        # to decide how to filter the queryset.
        if self.value() == '80s':
            return queryset.filter(birthday__gte=date(1980, 1, 1),
                                   birthday__lte=date(1989, 12, 31))
        if self.value() == '90s':
            return queryset.filter(birthday__gte=date(1990, 1, 1),
                                   birthday__lte=date(1999, 12, 31))

class PersonAdmin(admin.ModelAdmin):
    list_filter = (DecadeBornListFilter,)
```

: As a convenience, the `HttpRequest` object is passed to the `lookups` and `queryset` methods, for example:

```
class AuthDecadeBornListFilter(DecadeBornListFilter):

    def lookups(self, request, model_admin):
        if request.user.is_superuser:
            return super(AuthDecadeBornListFilter, self).lookups(request, model_admin)

    def queryset(self, request, queryset):
        if request.user.is_superuser:
            return super(AuthDecadeBornListFilter, self).queryset(request, queryset)
```

Also as a convenience, the `ModelAdmin` object is passed to the `lookups` method, for example if you want to base the lookups on the available data:

```
        class AdvancedDecadeBornListFilter(DecadeBornListFilter):

            def lookups(self, request, model_admin):
                """
                Only show the lookups if there actually is
                anyone born in the corresponding decades.
                """
                qs = model_admin.get_queryset(request)
                if qs.filter(birthday__gte=date(1980, 1, 1),
                             birthday__lte=date(1989, 12, 31)).exists():
                    yield ('80s', _('in the eighties'))
                if qs.filter(birthday__gte=date(1990, 1, 1),
                             birthday__lte=date(1999, 12, 31)).exists():
                    yield ('90s', _('in the nineties'))
```

- a tuple, where the first element is a field name and the second element is a class inheriting from `django.contrib.admin.FieldListFilter`, for example:

```
        class PersonAdmin(admin.ModelAdmin):
            list_filter = (
                ('is_staff', admin.BooleanFieldListFilter),
            )
```

You can limit the choices of a related model to the objects involved in that relation using `RelatedOnlyFieldListFilter`:

```
        class BookAdmin(admin.ModelAdmin):
            list_filter = (
                ('author', admin.RelatedOnlyFieldListFilter),
            )
```

Assuming `author` is a `ForeignKey` to a `User` model, this will limit the `list_filter` choices to the users who have written a book instead of listing all users.

: The `FieldListFilter` API is considered internal and might be changed.

List filter's typically appear only if the filter has more than one choice. A filter's `has_output()` method controls whether or not it appears.

It is possible to specify a custom template for rendering a list filter:

```
    class FilterWithCustomTemplate(admin.SimpleListFilter):
        template = "custom_template.html"
```

See the default template provided by django (`admin/filter.html`) for a concrete example.

`ModelAdmin.`**`list_max_show_all`**

Set `list_max_show_all` to control how many items can appear on a "Show all" admin change list page. The admin will display a "Show all" link on the change list only if the total result count is less than or equal to this setting. By default, this is set to `200`.

`ModelAdmin.`**`list_per_page`**

Set `list_per_page` to control how many items appear on each paginated admin change list page. By default, this is set to `100`.

`ModelAdmin.`**`list_select_related`**

Set `list_select_related` to tell Django to use *`select_related()`* in retrieving the list of objects on

the admin change list page. This can save you a bunch of database queries.

The value should be either a boolean, a list or a tuple. Default is `False`.

When value is `True`, `select_related()` will always be called. When value is set to `False`, Django will look at `list_display` and call `select_related()` if any `ForeignKey` is present.

If you need more fine-grained control, use a tuple (or list) as value for `list_select_related`. Empty tuple will prevent Django from calling `select_related` at all. Any other tuple will be passed directly to `select_related` as parameters. For example:

```
class ArticleAdmin(admin.ModelAdmin):
    list_select_related = ('author', 'category')
```

will call `select_related('author', 'category')`.

If you need to specify a dynamic value based on the request, you can implement a *get_list_select_related()* method.

`ModelAdmin.`**`ordering`**

Set `ordering` to specify how lists of objects should be ordered in the Django admin views. This should be a list or tuple in the same format as a model's *ordering* parameter.

If this isn't provided, the Django admin will use the model's default ordering.

If you need to specify a dynamic order (for example depending on user or language) you can implement a *get_ordering()* method.

`ModelAdmin.`**`paginator`**

The paginator class to be used for pagination. By default, *django.core.paginator.Paginator* is used. If the custom paginator class doesn't have the same constructor interface as *django.core.paginator.Paginator*, you will also need to provide an implementation for *ModelAdmin.get_paginator()*.

`ModelAdmin.`**`prepopulated_fields`**

Set `prepopulated_fields` to a dictionary mapping field names to the fields it should prepopulate from:

```
class ArticleAdmin(admin.ModelAdmin):
    prepopulated_fields = {"slug": ("title",)}
```

When set, the given fields will use a bit of JavaScript to populate from the fields assigned. The main use for this functionality is to automatically generate the value for `SlugField` fields from one or more other fields. The generated value is produced by concatenating the values of the source fields, and then by transforming that result into a valid slug (e.g. substituting dashes for spaces).

`prepopulated_fields` doesn't accept `DateTimeField`, `ForeignKey`, nor `ManyToManyField` fields.

`ModelAdmin.`**`preserve_filters`**

The admin now preserves filters on the list view after creating, editing or deleting an object. You can restore the previous behavior of clearing filters by setting this attribute to `False`.

`ModelAdmin.`**`radio_fields`**

By default, Django's admin uses a select-box interface (<select>) for fields that are `ForeignKey` or have `choices` set. If a field is present in `radio_fields`, Django will use a radio-button interface instead. Assuming `group` is a `ForeignKey` on the `Person` model:

```
class PersonAdmin(admin.ModelAdmin):
    radio_fields = {"group": admin.VERTICAL}
```

You have the choice of using `HORIZONTAL` or `VERTICAL` from the `django.contrib.admin` module.

Don't include a field in `radio_fields` unless it's a `ForeignKey` or has `choices` set.

ModelAdmin.**raw_id_fields**

By default, Django's admin uses a select-box interface (<select>) for fields that are `ForeignKey`. Sometimes you don't want to incur the overhead of having to select all the related instances to display in the drop-down.

`raw_id_fields` is a list of fields you would like to change into an `Input` widget for either a `ForeignKey` or `ManyToManyField`:

```
class ArticleAdmin(admin.ModelAdmin):
    raw_id_fields = ("newspaper",)
```

The `raw_id_fields` `Input` widget should contain a primary key if the field is a `ForeignKey` or a comma separated list of values if the field is a `ManyToManyField`. The `raw_id_fields` widget shows a magnifying glass button next to the field which allows users to search for and select a value:

ModelAdmin.**readonly_fields**

By default the admin shows all fields as editable. Any fields in this option (which should be a `list` or `tuple`) will display its data as-is and non-editable; they are also excluded from the *ModelForm* used for creating and editing. Note that when specifying *ModelAdmin.fields* or *ModelAdmin.fieldsets* the read-only fields must be present to be shown (they are ignored otherwise).

If `readonly_fields` is used without defining explicit ordering through *ModelAdmin.fields* or *ModelAdmin.fieldsets* they will be added last after all editable fields.

A read-only field can not only display data from a model's field, it can also display the output of a model's method or a method of the `ModelAdmin` class itself. This is very similar to the way *ModelAdmin.list_display* behaves. This provides an easy way to use the admin interface to provide feedback on the status of the objects being edited, for example:

```
from django.contrib import admin
from django.utils.html import format_html_join
from django.utils.safestring import mark_safe

class PersonAdmin(admin.ModelAdmin):
    readonly_fields = ('address_report',)

    def address_report(self, instance):
        # assuming get_full_address() returns a list of strings
        # for each line of the address and you want to separate each
        # line by a linebreak
        return format_html_join(
            mark_safe('<br/>'),
            '{}',
            ((line,) for line in instance.get_full_address()),
        ) or mark_safe("<span class='errors'>I can't determine this address.</span>")

    # short_description functions like a model field's verbose_name
    address_report.short_description = "Address"
```

ModelAdmin.**save_as**

Set `save_as` to enable a "save as new" feature on admin change forms.

Normally, objects have three save options: "Save", "Save and continue editing", and "Save and add another". If `save_as` is `True`, "Save and add another" will be replaced by a "Save as new" button that creates a new

object (with a new ID) rather than updating the existing object.

By default, `save_as` is set to `False`.

`ModelAdmin.`**`save_as_continue`**

When *`save_as=True`*, the default redirect after saving the new object is to the change view for that object. If you set `save_as_continue=False`, the redirect will be to the changelist view.

By default, `save_as_continue` is set to `True`.

`ModelAdmin.`**`save_on_top`**

Set `save_on_top` to add save buttons across the top of your admin change forms.

Normally, the save buttons appear only at the bottom of the forms. If you set `save_on_top`, the buttons will appear both on the top and the bottom.

By default, `save_on_top` is set to `False`.

`ModelAdmin.`**`search_fields`**

Set `search_fields` to enable a search box on the admin change list page. This should be set to a list of field names that will be searched whenever somebody submits a search query in that text box.

These fields should be some kind of text field, such as `CharField` or `TextField`. You can also perform a related lookup on a `ForeignKey` or `ManyToManyField` with the lookup API "follow" notation:

```
search_fields = ['foreign_key__related_fieldname']
```

For example, if you have a blog entry with an author, the following definition would enable searching blog entries by the email address of the author:

```
search_fields = ['user__email']
```

When somebody does a search in the admin search box, Django splits the search query into words and returns all objects that contain each of the words, case insensitive, where each word must be in at least one of `search_fields`. For example, if `search_fields` is set to `['first_name', 'last_name']` and a user searches for `john lennon`, Django will do the equivalent of this SQL `WHERE` clause:

```
WHERE (first_name ILIKE '%john%' OR last_name ILIKE '%john%')
AND (first_name ILIKE '%lennon%' OR last_name ILIKE '%lennon%')
```

For faster and/or more restrictive searches, prefix the field name with an operator:

^ Use the '^' operator to match starting at the beginning of the field. For example, if `search_fields` is set to `['^first_name', '^last_name']` and a user searches for `john lennon`, Django will do the equivalent of this SQL `WHERE` clause:

```
WHERE (first_name ILIKE 'john%' OR last_name ILIKE 'john%')
AND (first_name ILIKE 'lennon%' OR last_name ILIKE 'lennon%')
```

This query is more efficient than the normal `'%john%'` query, because the database only needs to check the beginning of a column's data, rather than seeking through the entire column's data. Plus, if the column has an index on it, some databases may be able to use the index for this query, even though it's a `LIKE` query.

= Use the '=' operator for case-insensitive exact matching. For example, if `search_fields` is set to `['=first_name', '=last_name']` and a user searches for `john lennon`, Django will do the equivalent of this SQL `WHERE` clause:

```
WHERE (first_name ILIKE 'john' OR last_name ILIKE 'john')
AND (first_name ILIKE 'lennon' OR last_name ILIKE 'lennon')
```

Note that the query input is split by spaces, so, following this example, it's currently not possible to search for all records in which `first_name` is exactly `'john winston'` (containing a space).

`@` Using the '@' operator to perform a full text match. This is like the default search method but uses an index. Currently this is only available for MySQL.

If you need to customize search you can use *`ModelAdmin.get_search_results()`* to provide additional or alternate search behavior.

`ModelAdmin.`**`show_full_result_count`**

Set `show_full_result_count` to control whether the full count of objects should be displayed on a filtered admin page (e.g. `99 results (103 total)`). If this option is set to `False`, a text like `99 results (Show all)` is displayed instead.

The default of `show_full_result_count=True` generates a query to perform a full count on the table which can be expensive if the table contains a large number of rows.

`ModelAdmin.`**`view_on_site`**

Set `view_on_site` to control whether or not to display the "View on site" link. This link should bring you to a URL where you can display the saved object.

This value can be either a boolean flag or a callable. If `True` (the default), the object's *`get_absolute_url()`* method will be used to generate the url.

If your model has a *`get_absolute_url()`* method but you don't want the "View on site" button to appear, you only need to set `view_on_site` to `False`:

```
from django.contrib import admin

class PersonAdmin(admin.ModelAdmin):
    view_on_site = False
```

In case it is a callable, it accepts the model instance as a parameter. For example:

```
from django.contrib import admin
from django.urls import reverse

class PersonAdmin(admin.ModelAdmin):
    def view_on_site(self, obj):
        url = reverse('person-detail', kwargs={'slug': obj.slug})
        return 'https://example.com' + url
```

Custom template options The *Overriding admin templates* section describes how to override or extend the default admin templates. Use the following options to override the default templates used by the *`ModelAdmin`* views:

`ModelAdmin.`**`add_form_template`**

Path to a custom template, used by *`add_view()`*.

`ModelAdmin.`**`change_form_template`**

Path to a custom template, used by *`change_view()`*.

`ModelAdmin.`**`change_list_template`**

Path to a custom template, used by *`changelist_view()`*.

`ModelAdmin.`**`delete_confirmation_template`**

Path to a custom template, used by *`delete_view()`* for displaying a confirmation page when deleting one or more objects.

`ModelAdmin.`**`delete_selected_confirmation_template`**

Path to a custom template, used by the `delete_selected` action method for displaying a confirmation page when deleting one or more objects. See the actions documentation.

ModelAdmin.**object_history_template**
Path to a custom template, used by *history_view()*.

ModelAdmin methods

: *ModelAdmin.save_model()* and *ModelAdmin.delete_model()* must save/delete the object, they are not for veto purposes, rather they allow you to perform extra operations.

ModelAdmin.**save_model**(*request*, *obj*, *form*, *change*)
The save_model method is given the HttpRequest, a model instance, a ModelForm instance and a boolean value based on whether it is adding or changing the object. Here you can do any pre- or post-save operations.

For example to attach request.user to the object prior to saving:

```
from django.contrib import admin

class ArticleAdmin(admin.ModelAdmin):
    def save_model(self, request, obj, form, change):
        obj.user = request.user
        obj.save()
```

ModelAdmin.**delete_model**(*request*, *obj*)
The delete_model method is given the HttpRequest and a model instance. Use this method to do pre- or post-delete operations.

ModelAdmin.**save_formset**(*request*, *form*, *formset*, *change*)
The save_formset method is given the HttpRequest, the parent ModelForm instance and a boolean value based on whether it is adding or changing the parent object.

For example, to attach request.user to each changed formset model instance:

```
class ArticleAdmin(admin.ModelAdmin):
    def save_formset(self, request, form, formset, change):
        instances = formset.save(commit=False)
        for obj in formset.deleted_objects:
            obj.delete()
        for instance in instances:
            instance.user = request.user
            instance.save()
        formset.save_m2m()
```

See also *Saving objects in the formset*.

ModelAdmin.**get_ordering**(*request*)
The get_ordering method takes a request as parameter and is expected to return a list or tuple for ordering similar to the *ordering* attribute. For example:

```
class PersonAdmin(admin.ModelAdmin):

    def get_ordering(self, request):
        if request.user.is_superuser:
            return ['name', 'rank']
        else:
            return ['name']
```

ModelAdmin.**get_search_results**(*request*, *queryset*, *search_term*)
The get_search_results method modifies the list of objects displayed into those that match the provided

search term. It accepts the request, a queryset that applies the current filters, and the user-provided search term. It returns a tuple containing a queryset modified to implement the search, and a boolean indicating if the results may contain duplicates.

The default implementation searches the fields named in *ModelAdmin.search_fields*.

This method may be overridden with your own custom search method. For example, you might wish to search by an integer field, or use an external tool such as Solr or Haystack. You must establish if the queryset changes implemented by your search method may introduce duplicates into the results, and return `True` in the second element of the return value.

For example, to search by `name` and `age`, you could use:

```
class PersonAdmin(admin.ModelAdmin):
    list_display = ('name', 'age')
    search_fields = ('name',)

    def get_search_results(self, request, queryset, search_term):
        queryset, use_distinct = super(PersonAdmin, self).get_search_results(request, queryset,
        try:
            search_term_as_int = int(search_term)
        except ValueError:
            pass
        else:
            queryset |= self.model.objects.filter(age=search_term_as_int)
        return queryset, use_distinct
```

This implementation is more efficient than `search_fields = ('name', '=age')` which results in a string comparison for the numeric field, for example `... OR UPPER("polls_choice"."votes"::text) = UPPER('4')` on PostgreSQL.

`ModelAdmin.`**`save_related`**(*request*, *form*, *formsets*, *change*)

The `save_related` method is given the `HttpRequest`, the parent `ModelForm` instance, the list of inline formsets and a boolean value based on whether the parent is being added or changed. Here you can do any pre- or post-save operations for objects related to the parent. Note that at this point the parent object and its form have already been saved.

`ModelAdmin.`**`get_readonly_fields`**(*request*, *obj=None*)

The `get_readonly_fields` method is given the `HttpRequest` and the `obj` being edited (or `None` on an add form) and is expected to return a `list` or `tuple` of field names that will be displayed as read-only, as described above in the *ModelAdmin.readonly_fields* section.

`ModelAdmin.`**`get_prepopulated_fields`**(*request*, *obj=None*)

The `get_prepopulated_fields` method is given the `HttpRequest` and the `obj` being edited (or `None` on an add form) and is expected to return a `dictionary`, as described above in the *ModelAdmin.prepopulated_fields* section.

`ModelAdmin.`**`get_list_display`**(*request*)

The `get_list_display` method is given the `HttpRequest` and is expected to return a `list` or `tuple` of field names that will be displayed on the changelist view as described above in the *ModelAdmin.list_display* section.

`ModelAdmin.`**`get_list_display_links`**(*request*, *list_display*)

The `get_list_display_links` method is given the `HttpRequest` and the `list` or `tuple` returned by *ModelAdmin.get_list_display()*. It is expected to return either `None` or a `list` or `tuple` of field names on the changelist that will be linked to the change view, as described in the *ModelAdmin.list_display_links* section.

`ModelAdmin.`**`get_fields`**(*request*, *obj=None*)

The `get_fields` method is given the `HttpRequest` and the `obj` being edited (or `None` on an add form)

and is expected to return a list of fields, as described above in the *ModelAdmin.fields* section.

ModelAdmin.**get_fieldsets**(*request*, *obj=None*)
The `get_fieldsets` method is given the `HttpRequest` and the `obj` being edited (or `None` on an add form) and is expected to return a list of two-tuples, in which each two-tuple represents a `<fieldset>` on the admin form page, as described above in the *ModelAdmin.fieldsets* section.

ModelAdmin.**get_list_filter**(*request*)
The `get_list_filter` method is given the `HttpRequest` and is expected to return the same kind of sequence type as for the *list_filter* attribute.

ModelAdmin.**get_list_select_related**(*request*)
The `get_list_select_related` method is given the `HttpRequest` and should return a boolean or list as *ModelAdmin.list_select_related* does.

ModelAdmin.**get_search_fields**(*request*)
The `get_search_fields` method is given the `HttpRequest` and is expected to return the same kind of sequence type as for the *search_fields* attribute.

ModelAdmin.**get_inline_instances**(*request*, *obj=None*)
The `get_inline_instances` method is given the `HttpRequest` and the `obj` being edited (or `None` on an add form) and is expected to return a `list` or `tuple` of *InlineModelAdmin* objects, as described below in the *InlineModelAdmin* section. For example, the following would return inlines without the default filtering based on add, change, and delete permissions:

```
class MyModelAdmin(admin.ModelAdmin):
    inlines = (MyInline,)

    def get_inline_instances(self, request, obj=None):
        return [inline(self.model, self.admin_site) for inline in self.inlines]
```

If you override this method, make sure that the returned inlines are instances of the classes defined in *inlines* or you might encounter a "Bad Request" error when adding related objects.

ModelAdmin.**get_urls**()
The `get_urls` method on a `ModelAdmin` returns the URLs to be used for that ModelAdmin in the same way as a URLconf. Therefore you can extend them as documented in URL dispatcher:

```
class MyModelAdmin(admin.ModelAdmin):
    def get_urls(self):
        urls = super(MyModelAdmin, self).get_urls()
        my_urls = [
            url(r'^my_view/$', self.my_view),
        ]
        return my_urls + urls

    def my_view(self, request):
        # ...
        context = dict(
           # Include common variables for rendering the admin template.
           self.admin_site.each_context(request),
           # Anything else you want in the context...
           key=value,
        )
        return TemplateResponse(request, "sometemplate.html", context)
```

If you want to use the admin layout, extend from `admin/base_site.html`:

```
{% extends "admin/base_site.html" %}
{% block content %}
```

```
    ...
    {% endblock %}
```

: Notice that the custom patterns are included *before* the regular admin URLs: the admin URL patterns are very permissive and will match nearly anything, so you'll usually want to prepend your custom URLs to the built-in ones.

In this example, `my_view` will be accessed at `/admin/myapp/mymodel/my_view/` (assuming the admin URLs are included at `/admin/`.)

However, the `self.my_view` function registered above suffers from two problems:

- It will *not* perform any permission checks, so it will be accessible to the general public.
- It will *not* provide any header details to prevent caching. This means if the page retrieves data from the database, and caching middleware is active, the page could show outdated information.

Since this is usually not what you want, Django provides a convenience wrapper to check permissions and mark the view as non-cacheable. This wrapper is `AdminSite.admin_view()` (i.e. `self.admin_site.admin_view` inside a `ModelAdmin` instance); use it like so:

```
class MyModelAdmin(admin.ModelAdmin):
    def get_urls(self):
        urls = super(MyModelAdmin, self).get_urls()
        my_urls = [
            url(r'^my_view/$', self.admin_site.admin_view(self.my_view))
        ]
        return my_urls + urls
```

Notice the wrapped view in the fifth line above:

```
url(r'^my_view/$', self.admin_site.admin_view(self.my_view))
```

This wrapping will protect `self.my_view` from unauthorized access and will apply the *`django.views.decorators.cache.never_cache()`* decorator to make sure it is not cached if the cache middleware is active.

If the page is cacheable, but you still want the permission check to be performed, you can pass a `cacheable=True` argument to `AdminSite.admin_view()`:

```
url(r'^my_view/$', self.admin_site.admin_view(self.my_view, cacheable=True))
```

`ModelAdmin` views have `model_admin` attributes. Other `AdminSite` views have `admin_site` attributes.

`ModelAdmin.`**`get_form`**(*request*, *obj=None*, ***kwargs*)

Returns a *`ModelForm`* class for use in the admin add and change views, see *`add_view()`* and *`change_view()`*.

The base implementation uses *`modelform_factory()`* to subclass *`form`*, modified by attributes such as *`fields`* and *`exclude`*. So, for example, if you wanted to offer additional fields to superusers, you could swap in a different base form like so:

```
class MyModelAdmin(admin.ModelAdmin):
    def get_form(self, request, obj=None, **kwargs):
        if request.user.is_superuser:
            kwargs['form'] = MySuperuserForm
        return super(MyModelAdmin, self).get_form(request, obj, **kwargs)
```

You may also simply return a custom *`ModelForm`* class directly.

ModelAdmin.**get_formsets_with_inlines**(*request*, *obj=None*)

Yields (FormSet, *InlineModelAdmin*) pairs for use in admin add and change views.

For example if you wanted to display a particular inline only in the change view, you could override get_formsets_with_inlines as follows:

```
class MyModelAdmin(admin.ModelAdmin):
    inlines = [MyInline, SomeOtherInline]

    def get_formsets_with_inlines(self, request, obj=None):
        for inline in self.get_inline_instances(request, obj):
            # hide MyInline in the add view
            if isinstance(inline, MyInline) and obj is None:
                continue
            yield inline.get_formset(request, obj), inline
```

ModelAdmin.**formfield_for_foreignkey**(*db_field*, *request*, ***kwargs*)

The formfield_for_foreignkey method on a ModelAdmin allows you to override the default formfield for a foreign keys field. For example, to return a subset of objects for this foreign key field based on the user:

```
class MyModelAdmin(admin.ModelAdmin):
    def formfield_for_foreignkey(self, db_field, request, **kwargs):
        if db_field.name == "car":
            kwargs["queryset"] = Car.objects.filter(owner=request.user)
        return super(MyModelAdmin, self).formfield_for_foreignkey(db_field, request, **kwargs)
```

This uses the HttpRequest instance to filter the Car foreign key field to only display the cars owned by the User instance.

ModelAdmin.**formfield_for_manytomany**(*db_field*, *request*, ***kwargs*)

Like the formfield_for_foreignkey method, the formfield_for_manytomany method can be overridden to change the default formfield for a many to many field. For example, if an owner can own multiple cars and cars can belong to multiple owners – a many to many relationship – you could filter the Car foreign key field to only display the cars owned by the User:

```
class MyModelAdmin(admin.ModelAdmin):
    def formfield_for_manytomany(self, db_field, request, **kwargs):
        if db_field.name == "cars":
            kwargs["queryset"] = Car.objects.filter(owner=request.user)
        return super(MyModelAdmin, self).formfield_for_manytomany(db_field, request, **kwargs)
```

ModelAdmin.**formfield_for_choice_field**(*db_field*, *request*, ***kwargs*)

Like the formfield_for_foreignkey and formfield_for_manytomany methods, the formfield_for_choice_field method can be overridden to change the default formfield for a field that has declared choices. For example, if the choices available to a superuser should be different than those available to regular staff, you could proceed as follows:

```
class MyModelAdmin(admin.ModelAdmin):
    def formfield_for_choice_field(self, db_field, request, **kwargs):
        if db_field.name == "status":
            kwargs['choices'] = (
                ('accepted', 'Accepted'),
                ('denied', 'Denied'),
            )
            if request.user.is_superuser:
                kwargs['choices'] += (('ready', 'Ready for deployment'),)
        return super(MyModelAdmin, self).formfield_for_choice_field(db_field, request, **kwargs)
```

Note

Any `choices` attribute set on the formfield will be limited to the form field only. If the corresponding field on the model has choices set, the choices provided to the form must be a valid subset of those choices, otherwise the form submission will fail with a *`ValidationError`* when the model itself is validated before saving.

`ModelAdmin.`**`get_changelist`**(*request*, ***kwargs*)
Returns the `Changelist` class to be used for listing. By default, `django.contrib.admin.views.main.ChangeList` is used. By inheriting this class you can change the behavior of the listing.

`ModelAdmin.`**`get_changelist_form`**(*request*, ***kwargs*)
Returns a *`ModelForm`* class for use in the `Formset` on the changelist page. To use a custom form, for example:

```
from django import forms

class MyForm(forms.ModelForm):
    pass

class MyModelAdmin(admin.ModelAdmin):
    def get_changelist_form(self, request, **kwargs):
        return MyForm
```

Note

If you define the `Meta.model` attribute on a *`ModelForm`*, you must also define the `Meta.fields` attribute (or the `Meta.exclude` attribute). However, `ModelAdmin` ignores this value, overriding it with the *`ModelAdmin.list_editable`* attribute. The easiest solution is to omit the `Meta.model` attribute, since `ModelAdmin` will provide the correct model to use.

`ModelAdmin.`**`get_changelist_formset`**(*request*, ***kwargs*)
Returns a *ModelFormSet* class for use on the changelist page if *`list_editable`* is used. To use a custom formset, for example:

```
from django.forms import BaseModelFormSet

class MyAdminFormSet(BaseModelFormSet):
    pass

class MyModelAdmin(admin.ModelAdmin):
    def get_changelist_formset(self, request, **kwargs):
        kwargs['formset'] = MyAdminFormSet
        return super(MyModelAdmin, self).get_changelist_formset(request, **kwargs)
```

`ModelAdmin.`**`has_add_permission`**(*request*)
Should return `True` if adding an object is permitted, `False` otherwise.

`ModelAdmin.`**`has_change_permission`**(*request*, *obj=None*)
Should return `True` if editing obj is permitted, `False` otherwise. If obj is `None`, should return `True` or `False` to indicate whether editing of objects of this type is permitted in general (e.g., `False` will be interpreted as meaning that the current user is not permitted to edit any object of this type).

`ModelAdmin.`**`has_delete_permission`**(*request*, *obj=None*)
Should return `True` if deleting obj is permitted, `False` otherwise. If obj is `None`, should return `True` or

`False` to indicate whether deleting objects of this type is permitted in general (e.g., `False` will be interpreted as meaning that the current user is not permitted to delete any object of this type).

`ModelAdmin.`**`has_module_permission`**(*request*)
Should return `True` if displaying the module on the admin index page and accessing the module's index page is permitted, `False` otherwise. Uses *`User.has_module_perms()`* by default. Overriding it does not restrict access to the add, change or delete views, *`has_add_permission()`*, *`has_change_permission()`*, and *`has_delete_permission()`* should be used for that.

`ModelAdmin.`**`get_queryset`**(*request*)
The `get_queryset` method on a `ModelAdmin` returns a *`QuerySet`* of all model instances that can be edited by the admin site. One use case for overriding this method is to show objects owned by the logged-in user:

```
class MyModelAdmin(admin.ModelAdmin):
    def get_queryset(self, request):
        qs = super(MyModelAdmin, self).get_queryset(request)
        if request.user.is_superuser:
            return qs
        return qs.filter(author=request.user)
```

`ModelAdmin.`**`message_user`**(*request*, *message*, *level=messages.INFO*, *extra_tags=''*, *fail_silently=False*)
Sends a message to the user using the *`django.contrib.messages`* backend. See the *custom ModelAdmin example*.

Keyword arguments allow you to change the message level, add extra CSS tags, or fail silently if the `contrib.messages` framework is not installed. These keyword arguments match those for *`django.contrib.messages.add_message()`*, see that function's documentation for more details. One difference is that the level may be passed as a string label in addition to integer/constant.

`ModelAdmin.`**`get_paginator`**(*request*, *queryset*, *per_page*, *orphans=0*, *allow_empty_first_page=True*)
Returns an instance of the paginator to use for this view. By default, instantiates an instance of *`paginator`*.

`ModelAdmin.`**`response_add`**(*request*, *obj*, *post_url_continue=None*)
Determines the *`HttpResponse`* for the *`add_view()`* stage.

`response_add` is called after the admin form is submitted and just after the object and all the related instances have been created and saved. You can override it to change the default behavior after the object has been created.

`ModelAdmin.`**`response_change`**(*request*, *obj*)
Determines the *`HttpResponse`* for the *`change_view()`* stage.

`response_change` is called after the admin form is submitted and just after the object and all the related instances have been saved. You can override it to change the default behavior after the object has been changed.

`ModelAdmin.`**`response_delete`**(*request*, *obj_display*, *obj_id*)
Determines the *`HttpResponse`* for the *`delete_view()`* stage.

`response_delete` is called after the object has been deleted. You can override it to change the default behavior after the object has been deleted.

`obj_display` is a string with the name of the deleted object.

`obj_id` is the serialized identifier used to retrieve the object to be deleted.

`ModelAdmin.`**`get_changeform_initial_data`**(*request*)
A hook for the initial data on admin change forms. By default, fields are given initial values from `GET` parameters. For instance, `?name=initial_value` will set the `name` field's initial value to be `initial_value`.

This method should return a dictionary in the form `{'fieldname': 'fieldval'}`:

```
    def get_changeform_initial_data(self, request):
        return {'name': 'custom_initial_value'}
```

Other methods

ModelAdmin.**add_view**(*request*, *form_url=''*, *extra_context=None*)
Django view for the model instance addition page. See note below.

ModelAdmin.**change_view**(*request*, *object_id*, *form_url=''*, *extra_context=None*)
Django view for the model instance editing page. See note below.

ModelAdmin.**changelist_view**(*request*, *extra_context=None*)
Django view for the model instances change list/actions page. See note below.

ModelAdmin.**delete_view**(*request*, *object_id*, *extra_context=None*)
Django view for the model instance(s) deletion confirmation page. See note below.

ModelAdmin.**history_view**(*request*, *object_id*, *extra_context=None*)
Django view for the page that shows the modification history for a given model instance.

Unlike the hook-type ModelAdmin methods detailed in the previous section, these five methods are in reality designed to be invoked as Django views from the admin application URL dispatching handler to render the pages that deal with model instances CRUD operations. As a result, completely overriding these methods will significantly change the behavior of the admin application.

One common reason for overriding these methods is to augment the context data that is provided to the template that renders the view. In the following example, the change view is overridden so that the rendered template is provided some extra mapping data that would not otherwise be available:

```
class MyModelAdmin(admin.ModelAdmin):

    # A template for a very customized change view:
    change_form_template = 'admin/myapp/extras/openstreetmap_change_form.html'

    def get_osm_info(self):
        # ...
        pass

    def change_view(self, request, object_id, form_url='', extra_context=None):
        extra_context = extra_context or {}
        extra_context['osm_data'] = self.get_osm_info()
        return super(MyModelAdmin, self).change_view(
            request, object_id, form_url, extra_context=extra_context,
        )
```

These views return *TemplateResponse* instances which allow you to easily customize the response data before rendering. For more details, see the TemplateResponse documentation.

ModelAdmin asset definitions

There are times where you would like add a bit of CSS and/or JavaScript to the add/change views. This can be accomplished by using a Media inner class on your ModelAdmin:

```
class ArticleAdmin(admin.ModelAdmin):
    class Media:
        css = {
            "all": ("my_styles.css",)
        }
        js = ("my_code.js",)
```

The staticfiles app prepends *STATIC_URL* (or *MEDIA_URL* if *STATIC_URL* is None) to any asset paths. The same rules apply as *regular asset definitions on forms*.

jQuery Django admin JavaScript makes use of the jQuery library.

To avoid conflicts with user-supplied scripts or libraries, Django's jQuery (version 2.2.3) is namespaced as `django.jQuery`. If you want to use jQuery in your own admin JavaScript without including a second copy, you can use the `django.jQuery` object on changelist and add/edit views.

The embedded jQuery was upgraded from 1.11.2 to 2.1.4. This drops support for Internet Explorer 8 and below. You can restore support by *including your own version of jQuery 1.X*.

The embedded jQuery was upgraded from 2.1.4 to 2.2.3.

The *ModelAdmin* class requires jQuery by default, so there is no need to add jQuery to your `ModelAdmin`'s list of media resources unless you have a specific need. For example, if you require the jQuery library to be in the global namespace (for example when using third-party jQuery plugins) or if you need a newer version of jQuery, you will have to include your own copy.

Django provides both uncompressed and 'minified' versions of jQuery, as `jquery.js` and `jquery.min.js` respectively.

ModelAdmin and *InlineModelAdmin* have a `media` property that returns a list of `Media` objects which store paths to the JavaScript files for the forms and/or formsets. If *DEBUG* is `True` it will return the uncompressed versions of the various JavaScript files, including `jquery.js`; if not, it will return the 'minified' versions.

Adding custom validation to the admin

Adding custom validation of data in the admin is quite easy. The automatic admin interface reuses *django.forms*, and the `ModelAdmin` class gives you the ability define your own form:

```
class ArticleAdmin(admin.ModelAdmin):
    form = MyArticleAdminForm
```

`MyArticleAdminForm` can be defined anywhere as long as you import where needed. Now within your form you can add your own custom validation for any field:

```
class MyArticleAdminForm(forms.ModelForm):
    def clean_name(self):
        # do something that validates your data
        return self.cleaned_data["name"]
```

It is important you use a `ModelForm` here otherwise things can break. See the forms documentation on custom validation and, more specifically, the *model form validation notes* for more information.

`InlineModelAdmin` objects

class `InlineModelAdmin`

class `TabularInline`

class `StackedInline`

The admin interface has the ability to edit models on the same page as a parent model. These are called inlines. Suppose you have these two models:

```
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    title = models.CharField(max_length=100)
```

You can edit the books authored by an author on the author page. You add inlines to a model by specifying them in a `ModelAdmin.inlines`:

```
from django.contrib import admin

class BookInline(admin.TabularInline):
    model = Book

class AuthorAdmin(admin.ModelAdmin):
    inlines = [
        BookInline,
    ]
```

Django provides two subclasses of `InlineModelAdmin` and they are:

- *TabularInline*
- *StackedInline*

The difference between these two is merely the template used to render them.

`InlineModelAdmin` options

`InlineModelAdmin` shares many of the same features as `ModelAdmin`, and adds some of its own (the shared features are actually defined in the `BaseModelAdmin` superclass). The shared features are:

- *form*
- *fieldsets*
- *fields*
- *formfield_overrides*
- *exclude*
- *filter_horizontal*
- *filter_vertical*
- *ordering*
- *prepopulated_fields*
- *get_queryset()*
- *radio_fields*
- *readonly_fields*
- *raw_id_fields*
- *formfield_for_choice_field()*
- *formfield_for_foreignkey()*

- *formfield_for_manytomany()*
- *has_add_permission()*
- *has_change_permission()*
- *has_delete_permission()*
- *has_module_permission()*

The `InlineModelAdmin` class adds:

`InlineModelAdmin.`**`model`**
The model which the inline is using. This is required.

`InlineModelAdmin.`**`fk_name`**
The name of the foreign key on the model. In most cases this will be dealt with automatically, but `fk_name` must be specified explicitly if there are more than one foreign key to the same parent model.

`InlineModelAdmin.`**`formset`**
This defaults to *BaseInlineFormSet*. Using your own formset can give you many possibilities of customization. Inlines are built around *model formsets*.

`InlineModelAdmin.`**`form`**
The value for `form` defaults to `ModelForm`. This is what is passed through to *inlineformset_factory()* when creating the formset for this inline.

: When writing custom validation for `InlineModelAdmin` forms, be cautious of writing validation that relies on features of the parent model. If the parent model fails to validate, it may be left in an inconsistent state as described in the warning in *Validation on a ModelForm*.

`InlineModelAdmin.`**`classes`**
A list or tuple containing extra CSS classes to apply to the fieldset that is rendered for the inlines. Defaults to `None`. As with classes configured in *fieldsets*, inlines with a `collapse` class will be initially collapsed and their header will have a small "show" link.

`InlineModelAdmin.`**`extra`**
This controls the number of extra forms the formset will display in addition to the initial forms. See the formsets documentation for more information.

For users with JavaScript-enabled browsers, an "Add another" link is provided to enable any number of additional inlines to be added in addition to those provided as a result of the `extra` argument.

The dynamic link will not appear if the number of currently displayed forms exceeds `max_num`, or if the user does not have JavaScript enabled.

InlineModelAdmin.get_extra() also allows you to customize the number of extra forms.

`InlineModelAdmin.`**`max_num`**
This controls the maximum number of forms to show in the inline. This doesn't directly correlate to the number of objects, but can if the value is small enough. See *Limiting the number of editable objects* for more information.

InlineModelAdmin.get_max_num() also allows you to customize the maximum number of extra forms.

`InlineModelAdmin.`**`min_num`**
This controls the minimum number of forms to show in the inline. See *modelformset_factory()* for more information.

InlineModelAdmin.get_min_num() also allows you to customize the minimum number of displayed forms.

InlineModelAdmin.**raw_id_fields**
By default, Django's admin uses a select-box interface (<select>) for fields that are `ForeignKey`. Sometimes you don't want to incur the overhead of having to select all the related instances to display in the drop-down.

`raw_id_fields` is a list of fields you would like to change into an `Input` widget for either a `ForeignKey` or `ManyToManyField`:

```
class BookInline(admin.TabularInline):
    model = Book
    raw_id_fields = ("pages",)
```

InlineModelAdmin.**template**
The template used to render the inline on the page.

InlineModelAdmin.**verbose_name**
An override to the `verbose_name` found in the model's inner `Meta` class.

InlineModelAdmin.**verbose_name_plural**
An override to the `verbose_name_plural` found in the model's inner `Meta` class.

InlineModelAdmin.**can_delete**
Specifies whether or not inline objects can be deleted in the inline. Defaults to `True`.

InlineModelAdmin.**show_change_link**
Specifies whether or not inline objects that can be changed in the admin have a link to the change form. Defaults to `False`.

InlineModelAdmin.**get_formset**(*request*, *obj=None*, ***kwargs*)
Returns a *BaseInlineFormSet* class for use in admin add/change views. See the example for *ModelAdmin.get_formsets_with_inlines*.

InlineModelAdmin.**get_extra**(*request*, *obj=None*, ***kwargs*)
Returns the number of extra inline forms to use. By default, returns the *InlineModelAdmin.extra* attribute.

Override this method to programmatically determine the number of extra inline forms. For example, this may be based on the model instance (passed as the keyword argument `obj`):

```
class BinaryTreeAdmin(admin.TabularInline):
    model = BinaryTree

    def get_extra(self, request, obj=None, **kwargs):
        extra = 2
        if obj:
            return extra - obj.binarytree_set.count()
        return extra
```

InlineModelAdmin.**get_max_num**(*request*, *obj=None*, ***kwargs*)
Returns the maximum number of extra inline forms to use. By default, returns the *InlineModelAdmin.max_num* attribute.

Override this method to programmatically determine the maximum number of inline forms. For example, this may be based on the model instance (passed as the keyword argument `obj`):

```
class BinaryTreeAdmin(admin.TabularInline):
    model = BinaryTree

    def get_max_num(self, request, obj=None, **kwargs):
        max_num = 10
        if obj and obj.parent:
```

```
            return max_num - 5
        return max_num
```

InlineModelAdmin.**get_min_num**(*request*, *obj=None*, ***kwargs*)

Returns the minimum number of inline forms to use. By default, returns the *InlineModelAdmin.min_num* attribute.

Override this method to programmatically determine the minimum number of inline forms. For example, this may be based on the model instance (passed as the keyword argument `obj`).

Working with a model with two or more foreign keys to the same parent model

It is sometimes possible to have more than one foreign key to the same model. Take this model for instance:

```
from django.db import models

class Friendship(models.Model):
    to_person = models.ForeignKey(Person, on_delete=models.CASCADE, related_name="friends")
    from_person = models.ForeignKey(Person, on_delete=models.CASCADE, related_name="from_friends")
```

If you wanted to display an inline on the `Person` admin add/change pages you need to explicitly define the foreign key since it is unable to do so automatically:

```
from django.contrib import admin
from myapp.models import Friendship

class FriendshipInline(admin.TabularInline):
    model = Friendship
    fk_name = "to_person"

class PersonAdmin(admin.ModelAdmin):
    inlines = [
        FriendshipInline,
    ]
```

Working with many-to-many models

By default, admin widgets for many-to-many relations will be displayed on whichever model contains the actual reference to the *ManyToManyField*. Depending on your `ModelAdmin` definition, each many-to-many field in your model will be represented by a standard HTML `<select multiple>`, a horizontal or vertical filter, or a `raw_id_admin` widget. However, it is also possible to replace these widgets with inlines.

Suppose we have the following models:

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=128)

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, related_name='groups')
```

If you want to display many-to-many relations using an inline, you can do so by defining an `InlineModelAdmin` object for the relationship:

```
from django.contrib import admin

class MembershipInline(admin.TabularInline):
    model = Group.members.through

class PersonAdmin(admin.ModelAdmin):
    inlines = [
        MembershipInline,
    ]

class GroupAdmin(admin.ModelAdmin):
    inlines = [
        MembershipInline,
    ]
    exclude = ('members',)
```

There are two features worth noting in this example.

Firstly - the `MembershipInline` class references `Group.members.through`. The `through` attribute is a reference to the model that manages the many-to-many relation. This model is automatically created by Django when you define a many-to-many field.

Secondly, the `GroupAdmin` must manually exclude the `members` field. Django displays an admin widget for a many-to-many field on the model that defines the relation (in this case, `Group`). If you want to use an inline model to represent the many-to-many relationship, you must tell Django's admin to *not* display this widget - otherwise you will end up with two widgets on your admin page for managing the relation.

Note that when using this technique the *`m2m_changed`* signals aren't triggered. This is because as far as the admin is concerned, `through` is just a model with two foreign key fields rather than a many-to-many relation.

In all other respects, the `InlineModelAdmin` is exactly the same as any other. You can customize the appearance using any of the normal `ModelAdmin` properties.

Working with many-to-many intermediary models

When you specify an intermediary model using the `through` argument to a *`ManyToManyField`*, the admin will not display a widget by default. This is because each instance of that intermediary model requires more information than could be displayed in a single widget, and the layout required for multiple widgets will vary depending on the intermediate model.

However, we still want to be able to edit that information inline. Fortunately, this is easy to do with inline admin models. Suppose we have the following models:

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=128)

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, through='Membership')

class Membership(models.Model):
    person = models.ForeignKey(Person, on_delete=models.CASCADE)
    group = models.ForeignKey(Group, on_delete=models.CASCADE)
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)
```

The first step in displaying this intermediate model in the admin is to define an inline class for the `Membership` model:

```
class MembershipInline(admin.TabularInline):
    model = Membership
    extra = 1
```

This simple example uses the default `InlineModelAdmin` values for the `Membership` model, and limits the extra add forms to one. This could be customized using any of the options available to `InlineModelAdmin` classes.

Now create admin views for the `Person` and `Group` models:

```
class PersonAdmin(admin.ModelAdmin):
    inlines = (MembershipInline,)

class GroupAdmin(admin.ModelAdmin):
    inlines = (MembershipInline,)
```

Finally, register your `Person` and `Group` models with the admin site:

```
admin.site.register(Person, PersonAdmin)
admin.site.register(Group, GroupAdmin)
```

Now your admin site is set up to edit `Membership` objects inline from either the `Person` or the `Group` detail pages.

Using generic relations as an inline

It is possible to use an inline with generically related objects. Let's say you have the following models:

```
from django.db import models
from django.contrib.contenttypes.fields import GenericForeignKey

class Image(models.Model):
    image = models.ImageField(upload_to="images")
    content_type = models.ForeignKey(ContentType, on_delete=models.CASCADE)
    object_id = models.PositiveIntegerField()
    content_object = GenericForeignKey("content_type", "object_id")

class Product(models.Model):
    name = models.CharField(max_length=100)
```

If you want to allow editing and creating an `Image` instance on the `Product`, add/change views you can use *`GenericTabularInline`* or *`GenericStackedInline`* (both subclasses of *`GenericInlineModelAdmin`*) provided by *`admin`*. They implement tabular and stacked visual layouts for the forms representing the inline objects, respectively, just like their non-generic counterparts. They behave just like any other inline. In your `admin.py` for this example app:

```
from django.contrib import admin
from django.contrib.contenttypes.admin import GenericTabularInline

from myproject.myapp.models import Image, Product

class ImageInline(GenericTabularInline):
    model = Image

class ProductAdmin(admin.ModelAdmin):
    inlines = [
        ImageInline,
    ]
```

```
admin.site.register(Product, ProductAdmin)
```

See the contenttypes documentation for more specific information.

Overriding admin templates

It is relatively easy to override many of the templates which the admin module uses to generate the various pages of an admin site. You can even override a few of these templates for a specific app, or a specific model.

Set up your projects admin template directories

The admin template files are located in the `contrib/admin/templates/admin` directory.

In order to override one or more of them, first create an `admin` directory in your project's `templates` directory. This can be any of the directories you specified in the *`DIRS`* option of the `DjangoTemplates` backend in the *`TEMPLATES`* setting. If you have customized the `'loaders'` option, be sure `'django.template.loaders.filesystem.Loader'` appears before `'django.template.loaders.app_directories.Loader'` so that your custom templates will be found by the template loading system before those that are included with *`django.contrib.admin`*.

Within this `admin` directory, create sub-directories named after your app. Within these app subdirectories create sub-directories named after your models. Note, that the admin app will lowercase the model name when looking for the directory, so make sure you name the directory in all lowercase if you are going to run your app on a case-sensitive filesystem.

To override an admin template for a specific app, copy and edit the template from the `django/contrib/admin/templates/admin` directory, and save it to one of the directories you just created.

For example, if we wanted to add a tool to the change list view for all the models in an app named `my_app`, we would copy `contrib/admin/templates/admin/change_list.html` to the `templates/admin/my_app/` directory of our project, and make any necessary changes.

If we wanted to add a tool to the change list view for only a specific model named 'Page', we would copy that same file to the `templates/admin/my_app/page` directory of our project.

Overriding vs. replacing an admin template

Because of the modular design of the admin templates, it is usually neither necessary nor advisable to replace an entire template. It is almost always better to override only the section of the template which you need to change.

To continue the example above, we want to add a new link next to the `History` tool for the `Page` model. After looking at `change_form.html` we determine that we only need to override the `object-tools-items` block. Therefore here is our new `change_form.html` :

```
{% extends "admin/change_form.html" %}
{% load i18n admin_urls %}
{% block object-tools-items %}
    <li>
        <a href="{% url opts|admin_urlname:'history' original.pk|admin_urlquote %}" class="historyli
    </li>
    <li>
        <a href="mylink/" class="historylink">My Link</a>
    </li>
    {% if has_absolute_url %}
```

```
        <li>
            <a href="{% url 'admin:view_on_site' content_type_id original.pk %}" class="viewsitelink"
        </li>
    {% endif %}
{% endblock %}
```

And that's it! If we placed this file in the `templates/admin/my_app` directory, our link would appear on the change form for all models within my_app.

Templates which may be overridden per app or model

Not every template in `contrib/admin/templates/admin` may be overridden per app or per model. The following can:

- `app_index.html`
- `change_form.html`
- `change_list.html`
- `delete_confirmation.html`
- `object_history.html`

For those templates that cannot be overridden in this way, you may still override them for your entire project. Just place the new version in your `templates/admin` directory. This is particularly useful to create custom 404 and 500 pages.

: Some of the admin templates, such as `change_list_results.html` are used to render custom inclusion tags. These may be overridden, but in such cases you are probably better off creating your own version of the tag in question and giving it a different name. That way you can use it selectively.

Root and login templates

If you wish to change the index, login or logout templates, you are better off creating your own `AdminSite` instance (see below), and changing the *`AdminSite.index_template`* , *`AdminSite.login_template`* or *`AdminSite.logout_template`* properties.

`AdminSite` objects

class `AdminSite`(*name='admin'*)

A Django administrative site is represented by an instance of `django.contrib.admin.sites.AdminSite`; by default, an instance of this class is created as `django.contrib.admin.site` and you can register your models and `ModelAdmin` instances with it.

When constructing an instance of an `AdminSite`, you can provide a unique instance name using the `name` argument to the constructor. This instance name is used to identify the instance, especially when *reversing admin URLs*. If no instance name is provided, a default instance name of `admin` will be used. See *Customizing the AdminSite class* for an example of customizing the *`AdminSite`* class.

AdminSite attributes

Templates can override or extend base admin templates as described in *Overriding admin templates*.

AdminSite.**site_header**
The text to put at the top of each admin page, as an <h1> (a string). By default, this is "Django administration".

AdminSite.**site_title**
The text to put at the end of each admin page's <title> (a string). By default, this is "Django site admin".

AdminSite.**site_url**
The URL for the "View site" link at the top of each admin page. By default, site_url is /. Set it to None to remove the link.

For sites running on a subpath, the *each_context()* method checks if the current request has request.META['SCRIPT_NAME'] set and uses that value if site_url isn't set to something other than /.

The SCRIPT_NAME support described in the previous paragraph was added.

AdminSite.**index_title**
The text to put at the top of the admin index page (a string). By default, this is "Site administration".

AdminSite.**index_template**
Path to a custom template that will be used by the admin site main index view.

AdminSite.**app_index_template**
Path to a custom template that will be used by the admin site app index view.

AdminSite.**empty_value_display**
The string to use for displaying empty values in the admin site's change list. Defaults to a dash. The value can also be overridden on a per ModelAdmin basis and on a custom field within a ModelAdmin by setting an empty_value_display attribute on the field. See *ModelAdmin.empty_value_display* for examples.

AdminSite.**login_template**
Path to a custom template that will be used by the admin site login view.

AdminSite.**login_form**
Subclass of *AuthenticationForm* that will be used by the admin site login view.

AdminSite.**logout_template**
Path to a custom template that will be used by the admin site logout view.

AdminSite.**password_change_template**
Path to a custom template that will be used by the admin site password change view.

AdminSite.**password_change_done_template**
Path to a custom template that will be used by the admin site password change done view.

AdminSite methods

AdminSite.**each_context**(*request*)
Returns a dictionary of variables to put in the template context for every page in the admin site.

Includes the following variables and values by default:

- site_header: *AdminSite.site_header*
- site_title: *AdminSite.site_title*
- site_url: *AdminSite.site_url*

- has_permission: *AdminSite.has_permission()*
- available_apps: a list of applications from the application registry available for the current user. Each entry in the list is a dict representing an application with the following keys:
  - app_label: the application label
  - app_url: the URL of the application index in the admin
  - has_module_perms: a boolean indicating if displaying and accessing of the module's index page is permitted for the current user
  - models: a list of the models available in the application

  Each model is a dict with the following keys:
  - object_name: class name of the model
  - name: plural name of the model
  - perms: a dict tracking add, change, and delete permissions
  - admin_url: admin changelist URL for the model
  - add_url: admin URL to add a new model instance

The available_apps variable was added.

AdminSite.**has_permission**(*request*)
Returns True if the user for the given HttpRequest has permission to view at least one page in the admin site. Defaults to requiring both *User.is_active* and *User.is_staff* to be True.

AdminSite.**register**(*model_or_iterable*, *admin_class=None*, ***options*)
Registers the given model class (or iterable of classes) with the given admin_class. admin_class defaults to *ModelAdmin* (the default admin options). If keyword arguments are given – e.g. list_display – they'll be applied as options to the admin class.

Raises *ImproperlyConfigured* if a model is abstract. and django.contrib.admin.sites.AlreadyRegistered if a model is already registered.

Hooking AdminSite instances into your URLconf

The last step in setting up the Django admin is to hook your AdminSite instance into your URLconf. Do this by pointing a given URL at the AdminSite.urls method. It is not necessary to use *include()*.

In this example, we register the default AdminSite instance django.contrib.admin.site at the URL /admin/

```
# urls.py
from django.conf.urls import url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', admin.site.urls),
]
```

In previous versions, you would pass admin.site.urls to *include()*.

Customizing the `AdminSite` class

If you'd like to set up your own admin site with custom behavior, you're free to subclass `AdminSite` and override or add anything you like. Then, simply create an instance of your `AdminSite` subclass (the same way you'd instantiate any other Python class) and register your models and `ModelAdmin` subclasses with it instead of with the default site. Finally, update `myproject/urls.py` to reference your *`AdminSite`* subclass.

`myapp/admin.py`

```
from django.contrib.admin import AdminSite

from .models import MyModel

class MyAdminSite(AdminSite):
    site_header = 'Monty Python administration'

admin_site = MyAdminSite(name='myadmin')
admin_site.register(MyModel)
```

`myproject/urls.py`

```
from django.conf.urls import url

from myapp.admin import admin_site

urlpatterns = [
    url(r'^myadmin/', admin_site.urls),
]
```

Note that you may not want autodiscovery of `admin` modules when using your own `AdminSite` instance since you will likely be importing all the per-app `admin` modules in your `myproject.admin` module. This means you need to put `'django.contrib.admin.apps.SimpleAdminConfig'` instead of `'django.contrib.admin'` in your *`INSTALLED_APPS`* setting.

Multiple admin sites in the same URLconf

It's easy to create multiple instances of the admin site on the same Django-powered website. Just create multiple instances of `AdminSite` and root each one at a different URL.

In this example, the URLs `/basic-admin/` and `/advanced-admin/` feature separate versions of the admin site – using the `AdminSite` instances `myproject.admin.basic_site` and `myproject.admin.advanced_site`, respectively:

```
# urls.py
from django.conf.urls import url
from myproject.admin import basic_site, advanced_site

urlpatterns = [
    url(r'^basic-admin/', basic_site.urls),
    url(r'^advanced-admin/', advanced_site.urls),
]
```

`AdminSite` instances take a single argument to their constructor, their name, which can be anything you like. This argument becomes the prefix to the URL names for the purposes of *reversing them*. This is only necessary if you are using more than one `AdminSite`.

Adding views to admin sites

Just like *ModelAdmin*, *AdminSite* provides a *get_urls()* method that can be overridden to define additional views for the site. To add a new view to your admin site, extend the base *get_urls()* method to include a pattern for your new view.

: Any view you render that uses the admin templates, or extends the base admin template, should set `request.current_app` before rendering the template. It should be set to either `self.name` if your view is on an `AdminSite` or `self.admin_site.name` if your view is on a `ModelAdmin`.

Adding a password-reset feature

You can add a password-reset feature to the admin site by adding a few lines to your URLconf. Specifically, add these four patterns:

```
from django.contrib.auth import views as auth_views

url(r'^admin/password_reset/$', auth_views.password_reset, name='admin_password_reset'),
url(r'^admin/password_reset/done/$', auth_views.password_reset_done, name='password_reset_done'),
url(r'^reset/(?P<uidb64>[0-9A-Za-z_\-]+)/(?P<token>.+)/$', auth_views.password_reset_confirm, name='p
url(r'^reset/done/$', auth_views.password_reset_complete, name='password_reset_complete'),
```

(This assumes you've added the admin at `admin/` and requires that you put the URLs starting with `^admin/` before the line that includes the admin app itself).

The presence of the `admin_password_reset` named URL will cause a "forgotten your password?" link to appear on the default admin log-in page under the password box.

`LogEntry` objects

class `models.`**`LogEntry`**

The `LogEntry` class tracks additions, changes, and deletions of objects done through the admin interface.

`LogEntry` attributes

`LogEntry.`**`action_time`**

The date and time of the action.

`LogEntry.`**`user`**

The user (an *AUTH_USER_MODEL* instance) who performed the action.

`LogEntry.`**`content_type`**

The *ContentType* of the modified object.

`LogEntry.`**`object_id`**

The textual representation of the modified object's primary key.

`LogEntry.`**`object_repr`**

The object's `repr()` after the modification.

`LogEntry.`**`action_flag`**

The type of action logged: `ADDITION`, `CHANGE`, `DELETION`.

For example, to get a list of all additions done through the admin:

```
    from django.contrib.admin.models import LogEntry, ADDITION

    LogEntry.objects.filter(action_flag=ADDITION)
```

LogEntry.**change_message**
The detailed description of the modification. In the case of an edit, for example, the message contains a list of the edited fields. The Django admin site formats this content as a JSON structure, so that *get_change_message()* can recompose a message translated in the current user language. Custom code might set this as a plain string though. You are advised to use the *get_change_message()* method to retrieve this value instead of accessing it directly.

Previously, this attribute was always a plain string. It is now JSON-structured so that the message can be translated in the current user language. Old messages are untouched.

LogEntry methods

LogEntry.**get_edited_object**()
A shortcut that returns the referenced object.

LogEntry.**get_change_message**()
Formats and translates *change_message* into the current user language. Messages created before Django 1.10 will always be displayed in the language in which they were logged.

Reversing admin URLs

When an *AdminSite* is deployed, the views provided by that site are accessible using Django's *URL reversing system*.

The *AdminSite* provides the following named URL patterns:

Page	URL name	Parameters
Index	index	
Login	login	
Logout	logout	
Password change	password_change	
Password change done	password_change_done	
i18n JavaScript	jsi18n	
Application index page	app_list	app_label
Redirect to object's page	view_on_site	content_type_id, object_id

Each *ModelAdmin* instance provides an additional set of named URLs:

Page	URL name	Parameters
Changelist	{{ app_label }}_{{ model_name }}_changelist	
Add	{{ app_label }}_{{ model_name }}_add	
History	{{ app_label }}_{{ model_name }}_history	object_id
Delete	{{ app_label }}_{{ model_name }}_delete	object_id
Change	{{ app_label }}_{{ model_name }}_change	object_id

The UserAdmin provides a named URL:

Page	URL name	Parameters
Password change	auth_user_password_change	user_id

These named URLs are registered with the application namespace admin, and with an instance namespace corresponding to the name of the Site instance.

So - if you wanted to get a reference to the Change view for a particular `Choice` object (from the polls application) in the default admin, you would call:

```
>>> from django.urls import reverse
>>> c = Choice.objects.get(...)
>>> change_url = reverse('admin:polls_choice_change', args=(c.id,))
```

This will find the first registered instance of the admin application (whatever the instance name), and resolve to the view for changing `poll.Choice` instances in that instance.

If you want to find a URL in a specific admin instance, provide the name of that instance as a `current_app` hint to the reverse call. For example, if you specifically wanted the admin view from the admin instance named `custom`, you would need to call:

```
>>> change_url = reverse('admin:polls_choice_change', args=(c.id,), current_app='custom')
```

For more details, see the documentation on *reversing namespaced URLs*.

To allow easier reversing of the admin urls in templates, Django provides an `admin_urlname` filter which takes an action as argument:

```
{% load admin_urls %}
<a href="{% url opts|admin_urlname:'add' %}">Add user</a>
<a href="{% url opts|admin_urlname:'delete' user.pk %}">Delete this user</a>
```

The action in the examples above match the last part of the URL names for *`ModelAdmin`* instances described above. The `opts` variable can be any object which has an `app_label` and `model_name` attributes and is usually supplied by the admin views for the current model.

The `staff_member_required` decorator

`staff_member_required`(*redirect_field_name='next'*, *login_url='admin:login'*)

This decorator is used on the admin views that require authorization. A view decorated with this function will having the following behavior:

- If the user is logged in, is a staff member (`User.is_staff=True`), and is active (`User.is_active=True`), execute the view normally.
- Otherwise, the request will be redirected to the URL specified by the `login_url` parameter, with the originally requested path in a query string variable specified by `redirect_field_name`. For example: `/admin/login/?next=/admin/polls/question/3/`.

Example usage:

```
from django.contrib.admin.views.decorators import staff_member_required

@staff_member_required
def my_view(request):
    ...
```

6.5.2 `django.contrib.auth`

This document provides API reference material for the components of Django's authentication system. For more details on the usage of these components or how to customize authentication and authorization see the authentication topic guide.

`User` model

Fields

class `models.`**`User`**

`User` objects have the following fields:

`username`

Required. 150 characters or fewer. Usernames may contain alphanumeric, `_`, `@`, `+`, `.` and `-` characters.

The `max_length` should be sufficient for many use cases. If you need a longer length, please use a *custom user model*. If you use MySQL with the `utf8mb4` encoding (recommended for proper Unicode support), specify at most `max_length=191` because MySQL can only create unique indexes with 191 characters in that case by default.

Usernames and Unicode

Django originally accepted only ASCII letters in usernames. Although it wasn't a deliberate choice, Unicode characters have always been accepted when using Python 3. Django 1.10 officially added Unicode support in usernames, keeping the ASCII-only behavior on Python 2, with the option to customize the behavior using *`User.username_validator`*.

The `max_length` increased from 30 to 150 characters.

`first_name`

Optional. 30 characters or fewer.

`last_name`

Optional. 30 characters or fewer.

`email`

Optional. Email address.

`password`

Required. A hash of, and metadata about, the password. (Django doesn't store the raw password.) Raw passwords can be arbitrarily long and can contain any character. See the password documentation.

`groups`

Many-to-many relationship to *`Group`*

`user_permissions`

Many-to-many relationship to *`Permission`*

`is_staff`

Boolean. Designates whether this user can access the admin site.

`is_active`

Boolean. Designates whether this user account should be considered active. We recommend that you set this flag to `False` instead of deleting accounts; that way, if your applications have any foreign keys to users, the foreign keys won't break.

This doesn't necessarily control whether or not the user can log in. Authentication backends aren't required to check for the `is_active` flag but the default backend (*`ModelBackend`*) and the *`RemoteUserBackend`* do. You can use *`AllowAllUsersModelBackend`* or *`AllowAllUsersRemoteUserBackend`* if you want to allow inactive users to login. In this case, you'll also want to customize the *`AuthenticationForm`* used by the *`login()`* view as it rejects inactive users. Be aware that the permission-checking methods such as *`has_perm()`* and the authentication in the Django admin all return `False` for inactive users.

In older versions, *ModelBackend* and *RemoteUserBackend* allowed inactive users to authenticate.

is_superuser
Boolean. Designates that this user has all permissions without explicitly assigning them.

last_login
A datetime of the user's last login.

date_joined
A datetime designating when the account was created. Is set to the current date/time by default when the account is created.

Attributes

class models.**User**

is_authenticated
Read-only attribute which is always `True` (as opposed to `AnonymousUser.is_authenticated` which is always `False`). This is a way to tell if the user has been authenticated. This does not imply any permissions and doesn't check if the user is active or has a valid session. Even though normally you will check this attribute on `request.user` to find out whether it has been populated by the *AuthenticationMiddleware* (representing the currently logged-in user), you should know this attribute is `True` for any *User* instance.

In older versions, this was a method. Backwards-compatibility support for using it as a method will be removed in Django 2.0.

is_anonymous
Read-only attribute which is always `False`. This is a way of differentiating *User* and *AnonymousUser* objects. Generally, you should prefer using *is_authenticated* to this attribute.

In older versions, this was a method. Backwards-compatibility support for using it as a method will be removed in Django 2.0.

username_validator
Points to a validator instance used to validate usernames. Defaults to *validators.UnicodeUsernameValidator* on Python 3 and *validators.ASCIIUsernameValidator* on Python 2.

To change the default username validator, you can subclass the `User` model and set this attribute to a different validator instance. For example, to use ASCII usernames on Python 3:

```
from django.contrib.auth.models import User
from django.contrib.auth.validators import ASCIIUsernameValidator

class CustomUser(User):
    username_validator = ASCIIUsernameValidator()

    class Meta:
        proxy = True  # If no new field is added.
```

Methods

class models.**User**

get_username()
Returns the username for the user. Since the User model can be swapped out, you should use this method instead of referencing the username attribute directly.

get_full_name()
Returns the *first_name* plus the *last_name*, with a space in between.

get_short_name()
Returns the *first_name*.

set_password(*raw_password*)
Sets the user's password to the given raw string, taking care of the password hashing. Doesn't save the *User* object.

When the raw_password is None, the password will be set to an unusable password, as if *set_unusable_password()* were used.

check_password(*raw_password*)
Returns True if the given raw string is the correct password for the user. (This takes care of the password hashing in making the comparison.)

set_unusable_password()
Marks the user as having no password set. This isn't the same as having a blank string for a password. *check_password()* for this user will never return True. Doesn't save the *User* object.

You may need this if authentication for your application takes place against an existing external source such as an LDAP directory.

has_usable_password()
Returns False if *set_unusable_password()* has been called for this user.

get_group_permissions(*obj=None*)
Returns a set of permission strings that the user has, through their groups.

If obj is passed in, only returns the group permissions for this specific object.

get_all_permissions(*obj=None*)
Returns a set of permission strings that the user has, both through group and user permissions.

If obj is passed in, only returns the permissions for this specific object.

has_perm(*perm*, *obj=None*)
Returns True if the user has the specified permission, where perm is in the format "<app label>.<permission codename>". (see documentation on *permissions*). If the user is inactive, this method will always return False.

If obj is passed in, this method won't check for a permission for the model, but for this specific object.

has_perms(*perm_list*, *obj=None*)
Returns True if the user has each of the specified permissions, where each perm is in the format "<app label>.<permission codename>". If the user is inactive, this method will always return False.

If obj is passed in, this method won't check for permissions for the model, but for the specific object.

has_module_perms(*package_name*)
Returns True if the user has any permissions in the given package (the Django app label). If the user is inactive, this method will always return False.

email_user(*subject*, *message*, *from_email=None*, ***kwargs*)
Sends an email to the user. If from_email is None, Django uses the *DEFAULT_FROM_EMAIL*. Any **kwargs are passed to the underlying *send_mail()* call.

Manager methods

class models.**UserManager**

The *User* model has a custom manager that has the following helper methods (in addition to the methods provided by *BaseUserManager*):

create_user(*username*, *email=None*, *password=None*, ***extra_fields*)

Creates, saves and returns a *User*.

The *username* and *password* are set as given. The domain portion of *email* is automatically converted to lowercase, and the returned *User* object will have *is_active* set to True.

If no password is provided, *set_unusable_password()* will be called.

The extra_fields keyword arguments are passed through to the *User*'s __init__ method to allow setting arbitrary fields on a *custom User model*.

See *Creating users* for example usage.

create_superuser(*username*, *email*, *password*, ***extra_fields*)

Same as *create_user()*, but sets *is_staff* and *is_superuser* to True.

AnonymousUser object

class models.**AnonymousUser**

django.contrib.auth.models.AnonymousUser is a class that implements the *django.contrib.auth.models.User* interface, with these differences:

- *id* is always None.
- *username* is always the empty string.
- *get_username()* always returns the empty string.
- *is_anonymous* is True instead of False.
- *is_authenticated* is False instead of True.
- *is_staff* and *is_superuser* are always False.
- *is_active* is always False.
- *groups* and *user_permissions* are always empty.
- *set_password()*, *check_password()*, *save()* and *delete()* raise NotImplementedError.

In practice, you probably won't need to use *AnonymousUser* objects on your own, but they're used by Web requests, as explained in the next section.

Permission model

class models.**Permission**

Fields

Permission objects have the following fields:

class models.**Permission**

name
Required. 255 characters or fewer. Example: `'Can vote'`.

content_type
Required. A reference to the `django_content_type` database table, which contains a record for each installed model.

codename
Required. 100 characters or fewer. Example: `'can_vote'`.

Methods

Permission objects have the standard data-access methods like any other Django model.

Group model

class `models.`**Group**

Fields

Group objects have the following fields:

class `models.`**Group**

name
Required. 80 characters or fewer. Any characters are permitted. Example: `'Awesome Users'`.

permissions
Many-to-many field to *Permission*:

```
group.permissions.set([permission_list])
group.permissions.add(permission, permission, ...)
group.permissions.remove(permission, permission, ...)
group.permissions.clear()
```

Validators

class `validators.`**ASCIIUsernameValidator**
A field validator allowing only ASCII letters, in addition to `@`, `.`, `+`, `-`, and `_`. The default validator for `User.username` on Python 2.

class `validators.`**UnicodeUsernameValidator**
A field validator allowing Unicode letters, in addition to `@`, `.`, `+`, `-`, and `_`. The default validator for `User.username` on Python 3.

Login and logout signals

The auth framework uses the following signals that can be used for notification when a user logs in or out.

user_logged_in`()`
Sent when a user logs in successfully.

Arguments sent with this signal:

sender The class of the user that just logged in.

request The current *HttpRequest* instance.

user The user instance that just logged in.

user_logged_out()
Sent when the logout method is called.

sender As above: the class of the user that just logged out or None if the user was not authenticated.

request The current *HttpRequest* instance.

user The user instance that just logged out or None if the user was not authenticated.

user_login_failed()
Sent when the user failed to login successfully

sender The name of the module used for authentication.

credentials A dictionary of keyword arguments containing the user credentials that were passed to *authenticate()* or your own custom authentication backend. Credentials matching a set of 'sensitive' patterns, (including password) will not be sent in the clear as part of the signal.

Authentication backends

This section details the authentication backends that come with Django. For information on how to use them and how to write your own authentication backends, see the *Other authentication sources section* of the User authentication guide.

Available authentication backends

The following backends are available in *django.contrib.auth.backends*:

class ModelBackend
This is the default authentication backend used by Django. It authenticates using credentials consisting of a user identifier and password. For Django's default user model, the user identifier is the username, for custom user models it is the field specified by USERNAME_FIELD (see Customizing Users and authentication).

It also handles the default permissions model as defined for *User* and *PermissionsMixin*.

has_perm(), *get_all_permissions()*, *get_user_permissions()*, and *get_group_permissions()* allow an object to be passed as a parameter for object-specific permissions, but this backend does not implement them other than returning an empty set of permissions if `obj is not None`.

authenticate(*username=None*, *password=None*, ***kwargs*)
Tries to authenticate `username` with `password` by calling *User.check_password*. If no `username` is provided, it tries to fetch a username from `kwargs` using the key *CustomUser.USERNAME_FIELD*. Returns an authenticated user or `None`.

get_user_permissions(*user_obj*, *obj=None*)
Returns the set of permission strings the `user_obj` has from their own user permissions. Returns an empty set if *is_anonymous* or *is_active* is `False`.

get_group_permissions(*user_obj*, *obj=None*)
Returns the set of permission strings the `user_obj` has from the permissions of the groups they belong. Returns an empty set if *is_anonymous* or *is_active* is `False`.

get_all_permissions(*user_obj*, *obj=None*)
Returns the set of permission strings the `user_obj` has, including both user permissions and group permissions. Returns an empty set if *is_anonymous* or *is_active* is `False`.

has_perm(*user_obj*, *perm*, *obj=None*)
Uses *get_all_permissions()* to check if `user_obj` has the permission string `perm`. Returns `False` if the user is not *is_active*.

has_module_perms(*self*, *user_obj*, *app_label*)
Returns whether the `user_obj` has any permissions on the app `app_label`.

user_can_authenticate()
Returns whether the user is allowed to authenticate. To match the behavior of *AuthenticationForm* which *prohibits inactive users from logging in*, this method returns `False` for users with *is_active=False*. Custom user models that don't have an *is_active* field are allowed.

class AllowAllUsersModelBackend
Same as *ModelBackend* except that it doesn't reject inactive users because *user_can_authenticate()* always returns `True`.

When using this backend, you'll likely want to customize the *AuthenticationForm* used by the *login()* view by overriding the *confirm_login_allowed()* method as it rejects inactive users.

class RemoteUserBackend
Use this backend to take advantage of external-to-Django-handled authentication. It authenticates using usernames passed in *request.META['REMOTE_USER']*. See the Authenticating against REMOTE_USER documentation.

If you need more control, you can create your own authentication backend that inherits from this class and override these attributes or methods:

RemoteUserBackend.**create_unknown_user**
`True` or `False`. Determines whether or not a *User* object is created if not already in the database. Defaults to `True`.

RemoteUserBackend.**authenticate**(*remote_user*)
The username passed as `remote_user` is considered trusted. This method simply returns the `User` object with the given username, creating a new `User` object if *create_unknown_user* is `True`.

Returns `None` if *create_unknown_user* is `False` and a `User` object with the given username is not found in the database.

RemoteUserBackend.**clean_username**(*username*)
Performs any cleaning on the `username` (e.g. stripping LDAP DN information) prior to using it to get or create a *User* object. Returns the cleaned username.

RemoteUserBackend.**configure_user**(*user*)
Configures a newly created user. This method is called immediately after a new user is created, and can be used to perform custom setup actions, such as setting the user's groups based on attributes in an LDAP directory. Returns the user object.

RemoteUserBackend.**user_can_authenticate**()
Returns whether the user is allowed to authenticate. This method returns `False` for users with *is_active=False*. Custom user models that don't have an *is_active* field are allowed.

class AllowAllUsersRemoteUserBackend
Same as *RemoteUserBackend* except that it doesn't reject inactive users because *user_can_authenticate* always returns `True`.

Utility functions

get_user(*request*)

Returns the user model instance associated with the given `request`'s session.

It checks if the authentication backend stored in the session is present in *AUTHENTICATION_BACKENDS*. If so, it uses the backend's `get_user()` method to retrieve the user model instance and then verifies the session by calling the user model's *get_session_auth_hash()* method.

Returns an instance of *AnonymousUser* if the authentication backend stored in the session is no longer in *AUTHENTICATION_BACKENDS*, if a user isn't returned by the backend's `get_user()` method, or if the session auth hash doesn't validate.

6.5.3 The contenttypes framework

Django includes a *contenttypes* application that can track all of the models installed in your Django-powered project, providing a high-level, generic interface for working with your models.

Overview

At the heart of the contenttypes application is the *ContentType* model, which lives at `django.contrib.contenttypes.models.ContentType`. Instances of *ContentType* represent and store information about the models installed in your project, and new instances of *ContentType* are automatically created whenever new models are installed.

Instances of *ContentType* have methods for returning the model classes they represent and for querying objects from those models. *ContentType* also has a *custom manager* that adds methods for working with *ContentType* and for obtaining instances of *ContentType* for a particular model.

Relations between your models and *ContentType* can also be used to enable "generic" relationships between an instance of one of your models and instances of any model you have installed.

Installing the contenttypes framework

The contenttypes framework is included in the default *INSTALLED_APPS* list created by `django-admin startproject`, but if you've removed it or if you manually set up your *INSTALLED_APPS* list, you can enable it by adding `'django.contrib.contenttypes'` to your *INSTALLED_APPS* setting.

It's generally a good idea to have the contenttypes framework installed; several of Django's other bundled applications require it:

- The admin application uses it to log the history of each object added or changed through the admin interface.
- Django's *authentication framework* uses it to tie user permissions to specific models.

The `ContentType` model

class ContentType

Each instance of *ContentType* has two fields which, taken together, uniquely describe an installed model:

app_label

The name of the application the model is part of. This is taken from the *app_label* attribute of the model, and includes only the *last* part of the application's Python import path; "django.contrib.contenttypes", for example, becomes an *app_label* of "contenttypes".

model
The name of the model class.

Additionally, the following property is available:

name
The human-readable name of the content type. This is taken from the *verbose_name* attribute of the model.

Let's look at an example to see how this works. If you already have the *contenttypes* application installed, and then add *the sites application* to your *INSTALLED_APPS* setting and run `manage.py migrate` to install it, the model *django.contrib.sites.models.Site* will be installed into your database. Along with it a new instance of *ContentType* will be created with the following values:

- *app_label* will be set to `'sites'` (the last part of the Python path "django.contrib.sites").
- *model* will be set to `'site'`.

Methods on `ContentType` instances

Each *ContentType* instance has methods that allow you to get from a *ContentType* instance to the model it represents, or to retrieve objects from that model:

`ContentType.`**`get_object_for_this_type`**`(`***kwargs*`)`
Takes a set of valid *lookup arguments* for the model the *ContentType* represents, and does *a get() lookup* on that model, returning the corresponding object.

`ContentType.`**`model_class`**`()`
Returns the model class represented by this *ContentType* instance.

For example, we could look up the *ContentType* for the *User* model:

```
>>> from django.contrib.contenttypes.models import ContentType
>>> ContentType.objects.get(app_label="auth", model="user")
<ContentType: user>
```

And then use it to query for a particular *User*, or to get access to the `User` model class:

```
>>> user_type.model_class()
<class 'django.contrib.auth.models.User'>
>>> user_type.get_object_for_this_type(username='Guido')
<User: Guido>
```

Together, *get_object_for_this_type()* and *model_class()* enable two extremely important use cases:

1. Using these methods, you can write high-level generic code that performs queries on any installed model – instead of importing and using a single specific model class, you can pass an `app_label` and `model` into a *ContentType* lookup at runtime, and then work with the model class or retrieve objects from it.
2. You can relate another model to *ContentType* as a way of tying instances of it to particular model classes, and use these methods to get access to those model classes.

Several of Django's bundled applications make use of the latter technique. For example, *the permissions system* in Django's authentication framework uses a *Permission* model with a foreign key to *ContentType*; this lets *Permission* represent concepts like "can add blog entry" or "can delete news story".

The `ContentTypeManager`

class `ContentTypeManager`
ContentType also has a custom manager, *ContentTypeManager*, which adds the following methods:

clear_cache()
Clears an internal cache used by *ContentType* to keep track of models for which it has created *ContentType* instances. You probably won't ever need to call this method yourself; Django will call it automatically when it's needed.

get_for_id(*id*)
Lookup a *ContentType* by ID. Since this method uses the same shared cache as *get_for_model()*, it's preferred to use this method over the usual `ContentType.objects.get(pk=id)`

get_for_model(*model*, *for_concrete_model=True*)
Takes either a model class or an instance of a model, and returns the *ContentType* instance representing that model. `for_concrete_model=False` allows fetching the *ContentType* of a proxy model.

get_for_models(**models*, *for_concrete_models=True*)
Takes a variadic number of model classes, and returns a dictionary mapping the model classes to the *ContentType* instances representing them. `for_concrete_models=False` allows fetching the *ContentType* of proxy models.

get_by_natural_key(*app_label*, *model*)
Returns the *ContentType* instance uniquely identified by the given application label and model name. The primary purpose of this method is to allow *ContentType* objects to be referenced via a *natural key* during deserialization.

The *get_for_model()* method is especially useful when you know you need to work with a *ContentType* but don't want to go to the trouble of obtaining the model's metadata to perform a manual lookup:

```
>>> from django.contrib.auth.models import User
>>> ContentType.objects.get_for_model(User)
<ContentType: user>
```

Generic relations

Adding a foreign key from one of your own models to *ContentType* allows your model to effectively tie itself to another model class, as in the example of the *Permission* model above. But it's possible to go one step further and use *ContentType* to enable truly generic (sometimes called "polymorphic") relationships between models.

A simple example is a tagging system, which might look like this:

```
from django.db import models
from django.contrib.contenttypes.fields import GenericForeignKey
from django.contrib.contenttypes.models import ContentType

class TaggedItem(models.Model):
    tag = models.SlugField()
    content_type = models.ForeignKey(ContentType, on_delete=models.CASCADE)
    object_id = models.PositiveIntegerField()
    content_object = GenericForeignKey('content_type', 'object_id')

    def __str__(self):              # __unicode__ on Python 2
        return self.tag
```

A normal *ForeignKey* can only "point to" one other model, which means that if the `TaggedItem` model used a *ForeignKey* it would have to choose one and only one model to store tags for. The contenttypes application provides a special field type (`GenericForeignKey`) which works around this and allows the relationship to be with any model:

class GenericForeignKey
There are three parts to setting up a *GenericForeignKey*:

1. Give your model a *ForeignKey* to *ContentType*. The usual name for this field is "content_type".
2. Give your model a field that can store primary key values from the models you'll be relating to. For most models, this means a *PositiveIntegerField*. The usual name for this field is "object_id".
3. Give your model a *GenericForeignKey*, and pass it the names of the two fields described above. If these fields are named "content_type" and "object_id", you can omit this – those are the default field names *GenericForeignKey* will look for.

for_concrete_model

If `False`, the field will be able to reference proxy models. Default is `True`. This mirrors the `for_concrete_model` argument to *get_for_model()*.

Primary key type compatibility

The "object_id" field doesn't have to be the same type as the primary key fields on the related models, but their primary key values must be coercible to the same type as the "object_id" field by its *get_db_prep_value()* method.

For example, if you want to allow generic relations to models with either *IntegerField* or *CharField* primary key fields, you can use *CharField* for the "object_id" field on your model since integers can be coerced to strings by *get_db_prep_value()*.

For maximum flexibility you can use a *TextField* which doesn't have a maximum length defined, however this may incur significant performance penalties depending on your database backend.

There is no one-size-fits-all solution for which field type is best. You should evaluate the models you expect to be pointing to and determine which solution will be most effective for your use case.

Serializing references to `ContentType` objects

If you're serializing data (for example, when generating *fixtures*) from a model that implements generic relations, you should probably be using a natural key to uniquely identify related *ContentType* objects. See *natural keys* and *dumpdata --natural-foreign* for more information.

This will enable an API similar to the one used for a normal *ForeignKey*; each `TaggedItem` will have a `content_object` field that returns the object it's related to, and you can also assign to that field or use it when creating a `TaggedItem`:

```
>>> from django.contrib.auth.models import User
>>> guido = User.objects.get(username='Guido')
>>> t = TaggedItem(content_object=guido, tag='bdfl')
>>> t.save()
>>> t.content_object
<User: Guido>
```

Due to the way *GenericForeignKey* is implemented, you cannot use such fields directly with filters (`filter()` and `exclude()`, for example) via the database API. Because a *GenericForeignKey* isn't a normal field object, these examples will *not* work:

```
# This will fail
>>> TaggedItem.objects.filter(content_object=guido)
# This will also fail
>>> TaggedItem.objects.get(content_object=guido)
```

Likewise, *GenericForeignKey*s does not appear in *ModelForm*s.

Reverse generic relations

class GenericRelation

related_query_name
The relation on the related object back to this object doesn't exist by default. Setting `related_query_name` creates a relation from the related object back to this one. This allows querying and filtering from the related object.

If you know which models you'll be using most often, you can also add a "reverse" generic relationship to enable an additional API. For example:

```
from django.db import models
from django.contrib.contenttypes.fields import GenericRelation

class Bookmark(models.Model):
    url = models.URLField()
    tags = GenericRelation(TaggedItem)
```

`Bookmark` instances will each have a `tags` attribute, which can be used to retrieve their associated `TaggedItems`:

```
>>> b = Bookmark(url='https://www.djangoproject.com/')
>>> b.save()
>>> t1 = TaggedItem(content_object=b, tag='django')
>>> t1.save()
>>> t2 = TaggedItem(content_object=b, tag='python')
>>> t2.save()
>>> b.tags.all()
<QuerySet [<TaggedItem: django>, <TaggedItem: python>]>
```

Defining *GenericRelation* with `related_query_name` set allows querying from the related object:

```
tags = GenericRelation(TaggedItem, related_query_name='bookmarks')
```

This enables filtering, ordering, and other query operations on `Bookmark` from `TaggedItem`:

```
>>> # Get all tags belonging to bookmarks containing `django` in the url
>>> TaggedItem.objects.filter(bookmarks__url__contains='django')
<QuerySet [<TaggedItem: django>, <TaggedItem: python>]>
```

Of course, if you don't add the reverse relationship, you can do the same types of lookups manually:

```
>>> b = Bookmark.objects.get(url='https://www.djangoproject.com/')
>>> bookmark_type = ContentType.objects.get_for_model(b)
>>> TaggedItem.objects.filter(content_type__pk=bookmark_type.id, object_id=b.id)
<QuerySet [<TaggedItem: django>, <TaggedItem: python>]>
```

Just as *GenericForeignKey* accepts the names of the content-type and object-ID fields as arguments, so too does *GenericRelation*; if the model which has the generic foreign key is using non-default names for those fields, you must pass the names of the fields when setting up a *GenericRelation* to it. For example, if the `TaggedItem` model referred to above used fields named `content_type_fk` and `object_primary_key` to create its generic foreign key, then a *GenericRelation* back to it would need to be defined like so:

```
tags = GenericRelation(
    TaggedItem,
    content_type_field='content_type_fk',
    object_id_field='object_primary_key',
)
```

Note also, that if you delete an object that has a *GenericRelation*, any objects which have a *GenericForeignKey* pointing at it will be deleted as well. In the example above, this means that if a `Bookmark` object were deleted, any `TaggedItem` objects pointing at it would be deleted at the same time.

Unlike *ForeignKey*, *GenericForeignKey* does not accept an *on_delete* argument to customize this behavior; if desired, you can avoid the cascade-deletion simply by not using *GenericRelation*, and alternate behavior can be provided via the *pre_delete* signal.

Generic relations and aggregation

Django's database aggregation API works with a *GenericRelation*. For example, you can find out how many tags all the bookmarks have:

```
>>> Bookmark.objects.aggregate(Count('tags'))
{'tags__count': 3}
```

Generic relation in forms

The *django.contrib.contenttypes.forms* module provides:

- *BaseGenericInlineFormSet*
- A formset factory, *generic_inlineformset_factory()*, for use with *GenericForeignKey*.

class BaseGenericInlineFormSet

generic_inlineformset_factory(*model*, *form=ModelForm*, *formset=BaseGenericInlineFormSet*, *ct_field="content_type"*, *fk_field="object_id"*, *fields=None*, *exclude=None*, *extra=3*, *can_order=False*, *can_delete=True*, *max_num=None*, *formfield_callback=None*, *validate_max=False*, *for_concrete_model=True*, *min_num=None*, *validate_min=False*)

Returns a `GenericInlineFormSet` using *modelformset_factory()*.

You must provide `ct_field` and `fk_field` if they are different from the defaults, `content_type` and `object_id` respectively. Other parameters are similar to those documented in *modelformset_factory()* and *inlineformset_factory()*.

The `for_concrete_model` argument corresponds to the *for_concrete_model* argument on `GenericForeignKey`.

Generic relations in admin

The *django.contrib.contenttypes.admin* module provides *GenericTabularInline* and *GenericStackedInline* (subclasses of *GenericInlineModelAdmin*)

These classes and functions enable the use of generic relations in forms and the admin. See the model formset and *admin* documentation for more information.

class GenericInlineModelAdmin

The *GenericInlineModelAdmin* class inherits all properties from an *InlineModelAdmin* class. However, it adds a couple of its own for working with the generic relation:

ct_field

The name of the *ContentType* foreign key field on the model. Defaults to `content_type`.

ct_fk_field
The name of the integer field that represents the ID of the related object. Defaults to `object_id`.

class GenericTabularInline

class GenericStackedInline
Subclasses of *GenericInlineModelAdmin* with stacked and tabular layouts, respectively.

6.5.4 The flatpages app

Django comes with an optional "flatpages" application. It lets you store simple "flat" HTML content in a database and handles the management for you via Django's admin interface and a Python API.

A flatpage is a simple object with a URL, title and content. Use it for one-off, special-case pages, such as "About" or "Privacy Policy" pages, that you want to store in a database but for which you don't want to develop a custom Django application.

A flatpage can use a custom template or a default, systemwide flatpage template. It can be associated with one, or multiple, sites.

The content field may optionally be left blank if you prefer to put your content in a custom template.

Here are some examples of flatpages on Django-powered sites:

- http://www.lawrence.com/about/contact/
- http://www2.ljworld.com/site/rules/

Installation

To install the flatpages app, follow these steps:

1. Install the *sites framework* by adding `'django.contrib.sites'` to your *INSTALLED_APPS* setting, if it's not already in there.

   Also make sure you've correctly set *SITE_ID* to the ID of the site the settings file represents. This will usually be `1` (i.e. `SITE_ID = 1`, but if you're using the sites framework to manage multiple sites, it could be the ID of a different site.

2. Add `'django.contrib.flatpages'` to your *INSTALLED_APPS* setting.

Then either:

3. Add an entry in your URLconf. For example:

```
urlpatterns = [
    url(r'^pages/', include('django.contrib.flatpages.urls')),
]
```

or:

3. Add `'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware'` to your *MIDDLEWARE* setting.

4. Run the command *manage.py migrate*.

How it works

`manage.py migrate` creates two tables in your database: `django_flatpage` and `django_flatpage_sites`. `django_flatpage` is a simple lookup table that simply maps a URL to a title and bunch of text content. `django_flatpage_sites` associates a flatpage with a site.

Using the URLconf

There are several ways to include the flat pages in your URLconf. You can dedicate a particular path to flat pages:

```
urlpatterns = [
    url(r'^pages/', include('django.contrib.flatpages.urls')),
]
```

You can also set it up as a "catchall" pattern. In this case, it is important to place the pattern at the end of the other urlpatterns:

```
from django.contrib.flatpages import views

# Your other patterns here
urlpatterns += [
    url(r'^(?P<url>.*/)$', views.flatpage),
]
```

: If you set *APPEND_SLASH* to `False`, you must remove the slash in the catchall pattern or flatpages without a trailing slash will not be matched.

Another common setup is to use flat pages for a limited set of known pages and to hard code the urls, so you can reference them with the *url* template tag:

```
from django.contrib.flatpages import views

urlpatterns += [
    url(r'^about-us/$', views.flatpage, {'url': '/about-us/'}, name='about'),
    url(r'^license/$', views.flatpage, {'url': '/license/'}, name='license'),
]
```

Using the middleware

The *FlatpageFallbackMiddleware* can do all of the work.

class `FlatpageFallbackMiddleware`

Each time any Django application raises a 404 error, this middleware checks the flatpages database for the requested URL as a last resort. Specifically, it checks for a flatpage with the given URL with a site ID that corresponds to the *SITE_ID* setting.

If it finds a match, it follows this algorithm:

- If the flatpage has a custom template, it loads that template. Otherwise, it loads the template `flatpages/default.html`.
- It passes that template a single context variable, `flatpage`, which is the flatpage object. It uses *RequestContext* in rendering the template.

The middleware will only add a trailing slash and redirect (by looking at the *APPEND_SLASH* setting) if the resulting URL refers to a valid flatpage. Redirects are permanent (301 status code).

If it doesn't find a match, the request continues to be processed as usual.

The middleware only gets activated for 404s – not for 500s or responses of any other status code.

Flatpages will not apply view middleware

Because the `FlatpageFallbackMiddleware` is applied only after URL resolution has failed and produced a 404, the response it returns will not apply any *view middleware* methods. Only requests which are successfully routed to a view via normal URL resolution apply view middleware.

Note that the order of *`MIDDLEWARE`* matters. Generally, you can put *`FlatpageFallbackMiddleware`* at the end of the list. This means it will run first when processing the response, and ensures that any other response-processing middlewares see the real flatpage response rather than the 404.

For more on middleware, read the middleware docs.

Ensure that your 404 template works

Note that the *`FlatpageFallbackMiddleware`* only steps in once another view has successfully produced a 404 response. If another view or middleware class attempts to produce a 404 but ends up raising an exception instead, the response will become an HTTP 500 ("Internal Server Error") and the *`FlatpageFallbackMiddleware`* will not attempt to serve a flat page.

How to add, change and delete flatpages

Via the admin interface

If you've activated the automatic Django admin interface, you should see a "Flatpages" section on the admin index page. Edit flatpages as you edit any other object in the system.

The `FlatPage` model has an `enable_comments` field that isn't used by `contrib.flatpages`, but that could be useful for your project or third-party apps. It doesn't appear in the admin interface, but you can add it by registering a custom `ModelAdmin` for `FlatPage`:

```
from django.contrib import admin
from django.contrib.flatpages.admin import FlatPageAdmin
from django.contrib.flatpages.models import FlatPage
from django.utils.translation import ugettext_lazy as _

# Define a new FlatPageAdmin
class FlatPageAdmin(FlatPageAdmin):
    fieldsets = (
        (None, {'fields': ('url', 'title', 'content', 'sites')}),
        (_('Advanced options'), {
            'classes': ('collapse', ),
            'fields': (
                'enable_comments',
                'registration_required',
                'template_name',
            ),
        }),
    )

# Re-register FlatPageAdmin
```

```
admin.site.unregister(FlatPage)
admin.site.register(FlatPage, FlatPageAdmin)
```

The `enable_comments` field was removed from `FlatPageAdmin`.

Via the Python API

class `FlatPage`

Flatpages are represented by a standard Django model, which lives in django/contrib/flatpages/models.py. You can access flatpage objects via the Django database API.

Check for duplicate flatpage URLs.

If you add or modify flatpages via your own code, you will likely want to check for duplicate flatpage URLs within the same site. The flatpage form used in the admin performs this validation check, and can be imported from `django.contrib.flatpages.forms.FlatpageForm` and used in your own views.

Flatpage templates

By default, flatpages are rendered via the template `flatpages/default.html`, but you can override that for a particular flatpage: in the admin, a collapsed fieldset titled "Advanced options" (clicking will expand it) contains a field for specifying a template name. If you're creating a flat page via the Python API you can simply set the template name as the field `template_name` on the `FlatPage` object.

Creating the `flatpages/default.html` template is your responsibility; in your template directory, just create a `flatpages` directory containing a file `default.html`.

Flatpage templates are passed a single context variable, `flatpage`, which is the flatpage object.

Here's a sample `flatpages/default.html` template:

```
<!DOCTYPE html>
<html>
<head>
<title>{{ flatpage.title }}</title>
</head>
<body>
{{ flatpage.content }}
</body>
</html>
```

Since you're already entering raw HTML into the admin page for a flatpage, both `flatpage.title` and `flatpage.content` are marked as **not** requiring *automatic HTML escaping* in the template.

Getting a list of `FlatPage` objects in your templates

The flatpages app provides a template tag that allows you to iterate over all of the available flatpages on the *current site*.

Like all custom template tags, you'll need to *load its custom tag library* before you can use it. After loading the library, you can retrieve all current flatpages via the *`get_flatpages`* tag:

```
{% load flatpages %}
{% get_flatpages as flatpages %}
<ul>
    {% for page in flatpages %}
        <li><a href="{{ page.url }}">{{ page.title }}</a></li>
    {% endfor %}
</ul>
```

Displaying `registration_required` flatpages

By default, the *get_flatpages* templatetag will only show flatpages that are marked `registration_required = False`. If you want to display registration-protected flatpages, you need to specify an authenticated user using a `for` clause.

For example:

```
{% get_flatpages for someuser as about_pages %}
```

If you provide an anonymous user, *get_flatpages* will behave the same as if you hadn't provided a user – i.e., it will only show you public flatpages.

Limiting flatpages by base URL

An optional argument, `starts_with`, can be applied to limit the returned pages to those beginning with a particular base URL. This argument may be passed as a string, or as a variable to be resolved from the context.

For example:

```
{% get_flatpages '/about/' as about_pages %}
{% get_flatpages about_prefix as about_pages %}
{% get_flatpages '/about/' for someuser as about_pages %}
```

Integrating with `django.contrib.sitemaps`

class `FlatPageSitemap`

The *sitemaps.FlatPageSitemap* class looks at all publicly visible *flatpages* defined for the current *SITE_ID* (see the *sites documentation*) and creates an entry in the sitemap. These entries include only the *location* attribute – not *lastmod*, *changefreq* or *priority*.

Example

Here's an example of a URLconf using *FlatPageSitemap*:

```
from django.conf.urls import url
from django.contrib.flatpages.sitemaps import FlatPageSitemap
from django.contrib.sitemaps.views import sitemap

urlpatterns = [
    # ...

    # the sitemap
    url(r'^sitemap\.xml$', sitemap,
        {'sitemaps': {'flatpages': FlatPageSitemap}},
```

```
        name='django.contrib.sitemaps.views.sitemap'),
]
```

6.5.5 GeoDjango

GeoDjango intends to be a world-class geographic Web framework. Its goal is to make it as easy as possible to build GIS Web applications and harness the power of spatially enabled data.

GeoDjango Tutorial

Introduction

GeoDjango is an included contrib module for Django that turns it into a world-class geographic Web framework. GeoDjango strives to make it as simple as possible to create geographic Web applications, like location-based services. Its features include:

- Django model fields for OGC geometries and raster data.
- Extensions to Django's ORM for querying and manipulating spatial data.
- Loosely-coupled, high-level Python interfaces for GIS geometry and raster operations and data manipulation in different formats.
- Editing geometry fields from the admin.

This tutorial assumes familiarity with Django; thus, if you're brand new to Django, please read through the regular tutorial to familiarize yourself with Django first.

: GeoDjango has additional requirements beyond what Django requires – please consult the installation documentation for more details.

This tutorial will guide you through the creation of a geographic web application for viewing the world borders. [1] Some of the code used in this tutorial is taken from and/or inspired by the GeoDjango basic apps project. [2]

: Proceed through the tutorial sections sequentially for step-by-step instructions.

Setting Up

Create a Spatial Database Typically no special setup is required, so you can create a database as you would for any other project. We provide some tips for selected databases:

- Installing PostGIS
- Installing SpatiaLite

[1] Special thanks to Bjørn Sandvik of thematicmapping.org for providing and maintaining this dataset.
[2] GeoDjango basic apps was written by Dane Springmeyer, Josh Livni, and Christopher Schmidt.

Create a New Project Use the standard `django-admin` script to create a project called `geodjango`:

```
$ django-admin startproject geodjango
```

This will initialize a new project. Now, create a `world` Django application within the `geodjango` project:

```
$ cd geodjango
$ python manage.py startapp world
```

Configure `settings.py` The `geodjango` project settings are stored in the `geodjango/settings.py` file. Edit the database connection settings to match your setup:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.contrib.gis.db.backends.postgis',
        'NAME': 'geodjango',
        'USER': 'geo',
    },
}
```

In addition, modify the *INSTALLED_APPS* setting to include *django.contrib.admin*, *django.contrib.gis*, and `world` (your newly created application):

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.gis',
    'world',
]
```

Geographic Data

World Borders The world borders data is available in this zip file. Create a `data` directory in the `world` application, download the world borders data, and unzip. On GNU/Linux platforms, use the following commands:

```
$ mkdir world/data
$ cd world/data
$ wget http://thematicmapping.org/downloads/TM_WORLD_BORDERS-0.3.zip
$ unzip TM_WORLD_BORDERS-0.3.zip
$ cd ../..
```

The world borders ZIP file contains a set of data files collectively known as an ESRI Shapefile, one of the most popular geospatial data formats. When unzipped, the world borders dataset includes files with the following extensions:

- `.shp`: Holds the vector data for the world borders geometries.
- `.shx`: Spatial index file for geometries stored in the `.shp`.
- `.dbf`: Database file for holding non-geometric attribute data (e.g., integer and character fields).
- `.prj`: Contains the spatial reference information for the geographic data stored in the shapefile.

Use `ogrinfo` to examine spatial data The GDAL `ogrinfo` utility allows examining the metadata of shapefiles or other vector data sources:

```
$ ogrinfo world/data/TM_WORLD_BORDERS-0.3.shp
INFO: Open of `world/data/TM_WORLD_BORDERS-0.3.shp'
      using driver `ESRI Shapefile' successful.
1: TM_WORLD_BORDERS-0.3 (Polygon)
```

`ogrinfo` tells us that the shapefile has one layer, and that this layer contains polygon data. To find out more, we'll specify the layer name and use the `-so` option to get only the important summary information:

```
$ ogrinfo -so world/data/TM_WORLD_BORDERS-0.3.shp TM_WORLD_BORDERS-0.3
INFO: Open of `world/data/TM_WORLD_BORDERS-0.3.shp'
      using driver `ESRI Shapefile' successful.

Layer name: TM_WORLD_BORDERS-0.3
Geometry: Polygon
Feature Count: 246
Extent: (-180.000000, -90.000000) - (180.000000, 83.623596)
Layer SRS WKT:
GEOGCS["GCS_WGS_1984",
    DATUM["WGS_1984",
        SPHEROID["WGS_1984",6378137.0,298.257223563]],
    PRIMEM["Greenwich",0.0],
    UNIT["Degree",0.0174532925199433]]
FIPS: String (2.0)
ISO2: String (2.0)
ISO3: String (3.0)
UN: Integer (3.0)
NAME: String (50.0)
AREA: Integer (7.0)
POP2005: Integer (10.0)
REGION: Integer (3.0)
SUBREGION: Integer (3.0)
LON: Real (8.3)
LAT: Real (7.3)
```

This detailed summary information tells us the number of features in the layer (246), the geographic bounds of the data, the spatial reference system ("SRS WKT"), as well as type information for each attribute field. For example, `FIPS: String (2.0)` indicates that the `FIPS` character field has a maximum length of 2. Similarly, `LON: Real (8.3)` is a floating-point field that holds a maximum of 8 digits up to three decimal places.

Geographic Models

Defining a Geographic Model Now that you've examined your dataset using `ogrinfo`, create a GeoDjango model to represent this data:

```
from django.contrib.gis.db import models

class WorldBorder(models.Model):
    # Regular Django fields corresponding to the attributes in the
    # world borders shapefile.
    name = models.CharField(max_length=50)
    area = models.IntegerField()
    pop2005 = models.IntegerField('Population 2005')
    fips = models.CharField('FIPS Code', max_length=2)
    iso2 = models.CharField('2 Digit ISO', max_length=2)
    iso3 = models.CharField('3 Digit ISO', max_length=3)
```

```
    un = models.IntegerField('United Nations Code')
    region = models.IntegerField('Region Code')
    subregion = models.IntegerField('Sub-Region Code')
    lon = models.FloatField()
    lat = models.FloatField()

    # GeoDjango-specific: a geometry field (MultiPolygonField)
    mpoly = models.MultiPolygonField()

    # Returns the string representation of the model.
    def __str__(self):              # __unicode__ on Python 2
        return self.name
```

Note that the `models` module is imported from `django.contrib.gis.db`.

The default spatial reference system for geometry fields is WGS84 (meaning the SRID is 4326) – in other words, the field coordinates are in longitude, latitude pairs in units of degrees. To use a different coordinate system, set the SRID of the geometry field with the `srid` argument. Use an integer representing the coordinate system's EPSG code.

Run `migrate` After defining your model, you need to sync it with the database. First, create a database migration:

```
$ python manage.py makemigrations
Migrations for 'world':
  world/migrations/0001_initial.py:
    - Create model WorldBorder
```

Let's look at the SQL that will generate the table for the `WorldBorder` model:

```
$ python manage.py sqlmigrate world 0001
```

This command should produce the following output:

```
BEGIN;
--
-- Create model WorldBorder
--
CREATE TABLE "world_worldborder" (
    "id" serial NOT NULL PRIMARY KEY,
    "name" varchar(50) NOT NULL,
    "area" integer NOT NULL,
    "pop2005" integer NOT NULL,
    "fips" varchar(2) NOT NULL,
    "iso2" varchar(2) NOT NULL,
    "iso3" varchar(3) NOT NULL,
    "un" integer NOT NULL,
    "region" integer NOT NULL,
    "subregion" integer NOT NULL,
    "lon" double precision NOT NULL,
    "lat" double precision NOT NULL
    "mpoly" geometry(MULTIPOLYGON,4326) NOT NULL
)
;
CREATE INDEX "world_worldborder_mpoly_id" ON "world_worldborder" USING GIST ( "mpoly" );
COMMIT;
```

If this looks correct, run *`migrate`* to create this table in the database:

```
$ python manage.py migrate
Operations to perform:
```

```
  Apply all migrations: admin, auth, contenttypes, sessions, world
Running migrations:
  ...
  Applying world.0001_initial... OK
```

Importing Spatial Data

This section will show you how to import the world borders shapefile into the database via GeoDjango models using the LayerMapping data import utility.

There are many different ways to import data into a spatial database – besides the tools included within GeoDjango, you may also use the following:

- ogr2ogr: A command-line utility included with GDAL that can import many vector data formats into PostGIS, MySQL, and Oracle databases.
- shp2pgsql: This utility included with PostGIS imports ESRI shapefiles into PostGIS.

GDAL Interface Earlier, you used `ogrinfo` to examine the contents of the world borders shapefile. GeoDjango also includes a Pythonic interface to GDAL's powerful OGR library that can work with all the vector data sources that OGR supports.

First, invoke the Django shell:

```
$ python manage.py shell
```

If you downloaded the *World Borders* data earlier in the tutorial, then you can determine its path using Python's built-in `os` module:

```
>>> import os
>>> import world
>>> world_shp = os.path.abspath(os.path.join(os.path.dirname(world.__file__),
...                             'data', 'TM_WORLD_BORDERS-0.3.shp'))
```

Now, open the world borders shapefile using GeoDjango's *`DataSource`* interface:

```
>>> from django.contrib.gis.gdal import DataSource
>>> ds = DataSource(world_shp)
>>> print(ds)
/ ... /geodjango/world/data/TM_WORLD_BORDERS-0.3.shp (ESRI Shapefile)
```

Data source objects can have different layers of geospatial features; however, shapefiles are only allowed to have one layer:

```
>>> print(len(ds))
1
>>> lyr = ds[0]
>>> print(lyr)
TM_WORLD_BORDERS-0.3
```

You can see the layer's geometry type and how many features it contains:

```
>>> print(lyr.geom_type)
Polygon
>>> print(len(lyr))
246
```

: Unfortunately, the shapefile data format does not allow for greater specificity with regards to geometry types. This shapefile, like many others, actually includes `MultiPolygon` geometries, not Polygons. It's important to use a more general field type in models: a GeoDjango `MultiPolygonField` will accept a `Polygon` geometry, but a `PolygonField` will not accept a `MultiPolygon` type geometry. This is why the `WorldBorder` model defined above uses a `MultiPolygonField`.

The *`Layer`* may also have a spatial reference system associated with it. If it does, the `srs` attribute will return a *`SpatialReference`* object:

```
>>> srs = lyr.srs
>>> print(srs)
GEOGCS["GCS_WGS_1984",
    DATUM["WGS_1984",
        SPHEROID["WGS_1984",6378137.0,298.257223563]],
    PRIMEM["Greenwich",0.0],
    UNIT["Degree",0.0174532925199433]]
>>> srs.proj4 # PROJ.4 representation
'+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs '
```

This shapefile is in the popular WGS84 spatial reference system – in other words, the data uses longitude, latitude pairs in units of degrees.

In addition, shapefiles also support attribute fields that may contain additional data. Here are the fields on the World Borders layer:

```
>>> print(lyr.fields)
['FIPS', 'ISO2', 'ISO3', 'UN', 'NAME', 'AREA', 'POP2005', 'REGION', 'SUBREGION', 'LON', 'LAT']
```

The following code will let you examine the OGR types (e.g. integer or string) associated with each of the fields:

```
>>> [fld.__name__ for fld in lyr.field_types]
['OFTString', 'OFTString', 'OFTString', 'OFTInteger', 'OFTString', 'OFTInteger', 'OFTInteger', 'OFTIn
```

You can iterate over each feature in the layer and extract information from both the feature's geometry (accessed via the `geom` attribute) as well as the feature's attribute fields (whose **values** are accessed via `get()` method):

```
>>> for feat in lyr:
...     print(feat.get('NAME'), feat.geom.num_points)
...
Guernsey 18
Jersey 26
South Georgia South Sandwich Islands 338
Taiwan 363
```

`Layer` objects may be sliced:

```
>>> lyr[0:2]
[<django.contrib.gis.gdal.feature.Feature object at 0x2f47690>, <django.contrib.gis.gdal.feature.Feat
```

And individual features may be retrieved by their feature ID:

```
>>> feat = lyr[234]
>>> print(feat.get('NAME'))
San Marino
```

Boundary geometries may be exported as WKT and GeoJSON:

```
>>> geom = feat.geom
>>> print(geom.wkt)
POLYGON ((12.415798 43.957954,12.450554 ...
>>> print(geom.json)
{ "type": "Polygon", "coordinates": [ [ [ 12.415798, 43.957954 ], [ 12.450554, 43.979721 ], ...
```

LayerMapping To import the data, use a LayerMapping in a Python script. Create a file called `load.py` inside the `world` application, with the following code:

```
import os
from django.contrib.gis.utils import LayerMapping
from .models import WorldBorder

world_mapping = {
    'fips' : 'FIPS',
    'iso2' : 'ISO2',
    'iso3' : 'ISO3',
    'un' : 'UN',
    'name' : 'NAME',
    'area' : 'AREA',
    'pop2005' : 'POP2005',
    'region' : 'REGION',
    'subregion' : 'SUBREGION',
    'lon' : 'LON',
    'lat' : 'LAT',
    'mpoly' : 'MULTIPOLYGON',
}

world_shp = os.path.abspath(
    os.path.join(os.path.dirname(__file__), 'data', 'TM_WORLD_BORDERS-0.3.shp'),
)

def run(verbose=True):
    lm = LayerMapping(
        WorldBorder, world_shp, world_mapping,
        transform=False, encoding='iso-8859-1',
    )
    lm.save(strict=True, verbose=verbose)
```

A few notes about what's going on:

- Each key in the `world_mapping` dictionary corresponds to a field in the `WorldBorder` model. The value is the name of the shapefile field that data will be loaded from.
- The key `mpoly` for the geometry field is `MULTIPOLYGON`, the geometry type GeoDjango will import the field as. Even simple polygons in the shapefile will automatically be converted into collections prior to insertion into the database.
- The path to the shapefile is not absolute – in other words, if you move the `world` application (with `data` subdirectory) to a different location, the script will still work.
- The `transform` keyword is set to `False` because the data in the shapefile does not need to be converted – it's already in WGS84 (SRID=4326).
- The `encoding` keyword is set to the character encoding of the string values in the shapefile. This ensures that string values are read and saved correctly from their original encoding system.

Afterwards, invoke the Django shell from the `geodjango` project directory:

```
$ python manage.py shell
```

Next, import the `load` module, call the `run` routine, and watch `LayerMapping` do the work:

```
>>> from world import load
>>> load.run()
```

Try `ogrinspect` Now that you've seen how to define geographic models and import data with the LayerMapping data import utility, it's possible to further automate this process with use of the *`ogrinspect`* management command. The *`ogrinspect`* command introspects a GDAL-supported vector data source (e.g., a shapefile) and generates a model definition and `LayerMapping` dictionary automatically.

The general usage of the command goes as follows:

```
$ python manage.py ogrinspect [options] <data_source> <model_name> [options]
```

`data_source` is the path to the GDAL-supported data source and `model_name` is the name to use for the model. Command-line options may be used to further define how the model is generated.

For example, the following command nearly reproduces the `WorldBorder` model and mapping dictionary created above, automatically:

```
$ python manage.py ogrinspect world/data/TM_WORLD_BORDERS-0.3.shp WorldBorder \
    --srid=4326 --mapping --multi
```

A few notes about the command-line options given above:

- The `--srid=4326` option sets the SRID for the geographic field.
- The `--mapping` option tells `ogrinspect` to also generate a mapping dictionary for use with *`LayerMapping`*.
- The `--multi` option is specified so that the geographic field is a *`MultiPolygonField`* instead of just a *`PolygonField`*.

The command produces the following output, which may be copied directly into the `models.py` of a GeoDjango application:

```
# This is an auto-generated Django model module created by ogrinspect.
from django.contrib.gis.db import models

class WorldBorder(models.Model):
    fips = models.CharField(max_length=2)
    iso2 = models.CharField(max_length=2)
    iso3 = models.CharField(max_length=3)
    un = models.IntegerField()
    name = models.CharField(max_length=50)
    area = models.IntegerField()
    pop2005 = models.IntegerField()
    region = models.IntegerField()
    subregion = models.IntegerField()
    lon = models.FloatField()
    lat = models.FloatField()
    geom = models.MultiPolygonField(srid=4326)

# Auto-generated `LayerMapping` dictionary for WorldBorder model
worldborders_mapping = {
    'fips' : 'FIPS',
    'iso2' : 'ISO2',
```

```
    'iso3' : 'ISO3',
    'un' : 'UN',
    'name' : 'NAME',
    'area' : 'AREA',
    'pop2005' : 'POP2005',
    'region' : 'REGION',
    'subregion' : 'SUBREGION',
    'lon' : 'LON',
    'lat' : 'LAT',
    'geom' : 'MULTIPOLYGON',
}
```

Spatial Queries

Spatial Lookups GeoDjango adds spatial lookups to the Django ORM. For example, you can find the country in the `WorldBorder` table that contains a particular point. First, fire up the management shell:

```
$ python manage.py shell
```

Now, define a point of interest [3]:

```
>>> pnt_wkt = 'POINT(-95.3385 29.7245)'
```

The `pnt_wkt` string represents the point at -95.3385 degrees longitude, 29.7245 degrees latitude. The geometry is in a format known as Well Known Text (WKT), a standard issued by the Open Geospatial Consortium (OGC). [4] Import the `WorldBorder` model, and perform a `contains` lookup using the `pnt_wkt` as the parameter:

```
>>> from world.models import WorldBorder
>>> WorldBorder.objects.filter(mpoly__contains=pnt_wkt)
<QuerySet [<WorldBorder: United States>]>
```

Here, you retrieved a `QuerySet` with only one model: the border of the United States (exactly what you would expect).

Similarly, you may also use a GEOS geometry object. Here, you can combine the `intersects` spatial lookup with the `get` method to retrieve only the `WorldBorder` instance for San Marino instead of a queryset:

```
>>> from django.contrib.gis.geos import Point
>>> pnt = Point(12.4604, 43.9420)
>>> WorldBorder.objects.get(mpoly__intersects=pnt)
<WorldBorder: San Marino>
```

The `contains` and `intersects` lookups are just a subset of the available queries – the GeoDjango Database API documentation has more.

Automatic Spatial Transformations When doing spatial queries, GeoDjango automatically transforms geometries if they're in a different coordinate system. In the following example, coordinates will be expressed in EPSG SRID 32140, a coordinate system specific to south Texas **only** and in units of **meters**, not degrees:

```
>>> from django.contrib.gis.geos import Point, GEOSGeometry
>>> pnt = Point(954158.1, 4215137.1, srid=32140)
```

Note that `pnt` may also be constructed with EWKT, an "extended" form of WKT that includes the SRID:

[3] This point is the University of Houston Law Center.
[4] Open Geospatial Consortium, Inc., OpenGIS Simple Feature Specification For SQL.

```
>>> pnt = GEOSGeometry('SRID=32140;POINT(954158.1 4215137.1)')
```

GeoDjango's ORM will automatically wrap geometry values in transformation SQL, allowing the developer to work at a higher level of abstraction:

```
>>> qs = WorldBorder.objects.filter(mpoly__intersects=pnt)
>>> print(qs.query) # Generating the SQL
SELECT "world_worldborder"."id", "world_worldborder"."name", "world_worldborder"."area",
"world_worldborder"."pop2005", "world_worldborder"."fips", "world_worldborder"."iso2",
"world_worldborder"."iso3", "world_worldborder"."un", "world_worldborder"."region",
"world_worldborder"."subregion", "world_worldborder"."lon", "world_worldborder"."lat",
"world_worldborder"."mpoly" FROM "world_worldborder"
WHERE ST_Intersects("world_worldborder"."mpoly", ST_Transform(%s, 4326))
>>> qs # printing evaluates the queryset
<QuerySet [<WorldBorder: United States>]>
```

Raw queries

When using raw queries, you should generally wrap your geometry fields with the `asText()` SQL function (or `ST_AsText` for PostGIS) so that the field value will be recognized by GEOS:

```
City.objects.raw('SELECT id, name, asText(point) from myapp_city')
```

This is not absolutely required by PostGIS, but generally you should only use raw queries when you know exactly what you are doing.

Lazy Geometries GeoDjango loads geometries in a standardized textual representation. When the geometry field is first accessed, GeoDjango creates a *GEOS geometry object <ref-geos>*, exposing powerful functionality, such as serialization properties for popular geospatial formats:

```
>>> sm = WorldBorder.objects.get(name='San Marino')
>>> sm.mpoly
<MultiPolygon object at 0x24c6798>
>>> sm.mpoly.wkt # WKT
MULTIPOLYGON (((12.4157980000000006 43.9579540000000009, 12.4505540000000003 43.9797209999999978, ...
>>> sm.mpoly.wkb # WKB (as Python binary buffer)
<read-only buffer for 0x1fe2c70, size -1, offset 0 at 0x2564c40>
>>> sm.mpoly.geojson # GeoJSON (requires GDAL)
'{ "type": "MultiPolygon", "coordinates": [ [ [ [ 12.415798, 43.957954 ], [ 12.450554, 43.979721 ], ...
```

This includes access to all of the advanced geometric operations provided by the GEOS library:

```
>>> pnt = Point(12.4604, 43.9420)
>>> sm.mpoly.contains(pnt)
True
>>> pnt.contains(sm.mpoly)
False
```

Geographic annotations GeoDjango also offers a set of geographic annotations to compute distances and several other operations (intersection, difference, etc.). See the Geographic Database Functions documentation.

Putting your data on the map

Geographic Admin GeoDjango extends Django's admin application with support for editing geometry fields.

Basics GeoDjango also supplements the Django admin by allowing users to create and modify geometries on a JavaScript slippy map (powered by OpenLayers).

Let's dive right in. Create a file called `admin.py` inside the `world` application with the following code:

```
from django.contrib.gis import admin
from .models import WorldBorder

admin.site.register(WorldBorder, admin.GeoModelAdmin)
```

Next, edit your `urls.py` in the `geodjango` application folder as follows:

```
from django.conf.urls import url, include
from django.contrib.gis import admin

urlpatterns = [
    url(r'^admin/', admin.site.urls),
]
```

Create an admin user:

```
$ python manage.py createsuperuser
```

Next, start up the Django development server:

```
$ python manage.py runserver
```

Finally, browse to `http://localhost:8000/admin/`, and log in with the user you just created. Browse to any of the `WorldBorder` entries – the borders may be edited by clicking on a polygon and dragging the vertexes to the desired position.

`OSMGeoAdmin` With the *`OSMGeoAdmin`*, GeoDjango uses a Open Street Map layer in the admin. This provides more context (including street and thoroughfare details) than available with the *`GeoModelAdmin`* (which uses the Vector Map Level 0 WMS dataset hosted at OSGeo).

First, there are some important requirements:

- *`OSMGeoAdmin`* requires that GDAL is installed.
- The PROJ.4 datum shifting files must be installed (see the *PROJ.4 installation instructions* for more details).

If you meet this requirement, then just substitute the `OSMGeoAdmin` option class in your `admin.py` file:

```
admin.site.register(WorldBorder, admin.OSMGeoAdmin)
```

GeoDjango Installation

Overview

In general, GeoDjango installation requires:

1. *Python and Django*
2. *Spatial database*
3. Installing Geospatial libraries

Details for each of the requirements and installation instructions are provided in the sections below. In addition, platform-specific instructions are available for:

- *Mac OS X*
- *Windows*

Use the Source

Because GeoDjango takes advantage of the latest in the open source geospatial software technology, recent versions of the libraries are necessary. If binary packages aren't available for your platform, installation from source may be required. When compiling the libraries from source, please follow the directions closely, especially if you're a beginner.

Requirements

Python and Django Because GeoDjango is included with Django, please refer to Django's *installation instructions* for details on how to install.

Spatial database PostgreSQL (with PostGIS), MySQL (mostly with MyISAM engine), Oracle, and SQLite (with SpatiaLite) are the spatial databases currently supported.

: PostGIS is recommended, because it is the most mature and feature-rich open source spatial database.

The geospatial libraries required for a GeoDjango installation depends on the spatial database used. The following lists the library requirements, supported versions, and any notes for each of the supported database backends:

Database	Library Requirements	Supported Versions	Notes
PostgreSQL	GEOS, PROJ.4, PostGIS	9.2+	Requires PostGIS.
MySQL	GEOS	5.5+	Not OGC-compliant; *limited functionality*.
Oracle	GEOS	11.2+	XE not supported.
SQLite	GEOS, GDAL, PROJ.4, SpatiaLite	3.6.+	Requires SpatiaLite 3.0+, pysqlite2 2.5+

See also this comparison matrix on the OSGeo Wiki for PostgreSQL/PostGIS/GEOS/GDAL possible combinations.

Installation

Geospatial libraries

Installing Geospatial libraries GeoDjango uses and/or provides interfaces for the following open source geospatial libraries:

Program	Description	Required	Supported Versions
GEOS	Geometry Engine Open Source	Yes	3.4, 3.3
PROJ.4	Cartographic Projections library	Yes (PostgreSQL and SQLite only)	4.9, 4.8, 4.7, 4.6, 4.5, 4.4
GDAL	Geospatial Data Abstraction Library	Yes (SQLite only)	2.1, 2.0, 1.11, 1.10, 1.9, 1.8, 1.7
GeoIP	IP-based geolocation library	No	1.4
PostGIS	Spatial extensions for PostgreSQL	Yes (PostgreSQL only)	2.3, 2.2, 2.1, 2.0
SpatiaLite	Spatial extensions for SQLite	Yes (SQLite only)	4.3, 4.2, 4.1, 4.0, 3.0

Note that older or more recent versions of these libraries *may* also work totally fine with GeoDjango. Your mileage may vary.

Install GDAL

While *GDAL* is technically not required, it is *recommended*. Important features of GeoDjango (including the LayerMapping data import utility, geometry reprojection, and the geographic admin) depend on its functionality.

: The GeoDjango interfaces to GEOS, GDAL, and GeoIP may be used independently of Django. In other words, no database or settings file required – just import them as normal from `django.contrib.gis`.

On Debian/Ubuntu, you are advised to install the following packages which will install, directly or by dependency, the required geospatial libraries:

```
$ sudo apt-get install binutils libproj-dev gdal-bin
```

Optional packages to consider:

- `libgeoip1`: for GeoIP support
- `gdal-bin`: for GDAL command line programs like `ogr2ogr`
- `python-gdal` for GDAL's own Python bindings – includes interfaces for raster manipulation

Please also consult platform-specific instructions if you are on *Mac OS X* or *Windows*.

Building from source When installing from source on UNIX and GNU/Linux systems, please follow the installation instructions carefully, and install the libraries in the given order. If using MySQL or Oracle as the spatial database, only GEOS is required.

: On Linux platforms, it may be necessary to run the `ldconfig` command after installing each library. For example:

```
$ sudo make install
$ sudo ldconfig
```

: OS X users are required to install Apple Developer Tools in order to compile software from source. This is typically included on your OS X installation DVDs.

GEOS GEOS is a C++ library for performing geometric operations, and is the default internal geometry representation used by GeoDjango (it's behind the "lazy" geometries). Specifically, the C API library is called (e.g., `libgeos_c.so`) directly from Python using ctypes.

First, download GEOS 3.4.2 from the GEOS website and untar the source archive:

```
$ wget http://download.osgeo.org/geos/geos-3.4.2.tar.bz2
$ tar xjf geos-3.4.2.tar.bz2
```

Next, change into the directory where GEOS was unpacked, run the configure script, compile, and install:

```
$ cd geos-3.4.2
$ ./configure
$ make
$ sudo make install
$ cd ..
```

Troubleshooting

Can't find GEOS library When GeoDjango can't find GEOS, this error is raised:

```
ImportError: Could not find the GEOS library (tried "geos_c"). Try setting GEOS_LIBRARY_PATH in your
```

The most common solution is to properly configure your *Library environment settings* *or* set *GEOS_LIBRARY_PATH* in your settings.

If using a binary package of GEOS (e.g., on Ubuntu), you may need to *Install binutils*.

`GEOS_LIBRARY_PATH` If your GEOS library is in a non-standard location, or you don't want to modify the system's library path then the *`GEOS_LIBRARY_PATH`* setting may be added to your Django settings file with the full path to the GEOS C library. For example:

```
GEOS_LIBRARY_PATH = '/home/bob/local/lib/libgeos_c.so'
```

: The setting must be the *full* path to the **C** shared library; in other words you want to use `libgeos_c.so`, not `libgeos.so`.

See also *My logs are filled with GEOS-related errors*.

PROJ.4 PROJ.4 is a library for converting geospatial data to different coordinate reference systems.

First, download the PROJ.4 source code and datum shifting files [1]:

```
$ wget http://download.osgeo.org/proj/proj-4.9.1.tar.gz
$ wget http://download.osgeo.org/proj/proj-datumgrid-1.5.tar.gz
```

Next, untar the source code archive, and extract the datum shifting files in the `nad` subdirectory. This must be done *prior* to configuration:

[1] The datum shifting files are needed for converting data to and from certain projections. For example, the PROJ.4 string for the Google projection (900913 or 3857) requires the `null` grid file only included in the extra datum shifting files. It is easier to install the shifting files now, then to have debug a problem caused by their absence later.

```
$ tar xzf proj-4.9.1.tar.gz
$ cd proj-4.9.1/nad
$ tar xzf ../../proj-datumgrid-1.5.tar.gz
$ cd ..
```

Finally, configure, make and install PROJ.4:

```
$ ./configure
$ make
$ sudo make install
$ cd ..
```

GDAL GDAL is an excellent open source geospatial library that has support for reading most vector and raster spatial data formats. Currently, GeoDjango only supports GDAL's vector data capabilities [2]. *GEOS* and *PROJ.4* should be installed prior to building GDAL.

First download the latest GDAL release version and untar the archive:

```
$ wget http://download.osgeo.org/gdal/1.11.2/gdal-1.11.2.tar.gz
$ tar xzf gdal-1.11.2.tar.gz
$ cd gdal-1.11.2
```

Configure, make and install:

```
$ ./configure
$ make # Go get some coffee, this takes a while.
$ sudo make install
$ cd ..
```

: Because GeoDjango has its own Python interface, the preceding instructions do not build GDAL's own Python bindings. The bindings may be built by adding the `--with-python` flag when running `configure`. See GDAL/OGR In Python for more information on GDAL's bindings.

If you have any problems, please see the troubleshooting section below for suggestions and solutions.

Troubleshooting

Can't find GDAL library When GeoDjango can't find the GDAL library, the `HAS_GDAL` flag will be false:

```
>>> from django.contrib.gis import gdal
>>> gdal.HAS_GDAL
False
```

The solution is to properly configure your *Library environment settings* *or* set *GDAL_LIBRARY_PATH* in your settings.

`GDAL_LIBRARY_PATH` If your GDAL library is in a non-standard location, or you don't want to modify the system's library path then the *`GDAL_LIBRARY_PATH`* setting may be added to your Django settings file with the full path to the GDAL library. For example:

```
GDAL_LIBRARY_PATH = '/home/sue/local/lib/libgdal.so'
```

[2] Specifically, GeoDjango provides support for the OGR library, a component of GDAL.

Database installation

Installing PostGIS PostGIS adds geographic object support to PostgreSQL, turning it into a spatial database. *GEOS*, *PROJ.4* and *GDAL* should be installed prior to building PostGIS. You might also need additional libraries, see PostGIS requirements.

The psycopg2 module is required for use as the database adapter when using GeoDjango with PostGIS.

On Debian/Ubuntu, you are advised to install the following packages: postgresql-x.x, postgresql-x.x-postgis, postgresql-server-dev-x.x, python-psycopg2 (x.x matching the PostgreSQL version you want to install). Alternately, you can build from source. Consult the platform-specific instructions if you are on *Mac OS X* or *Windows*.

Post-installation

Creating a spatial database PostGIS 2 includes an extension for PostgreSQL that's used to enable spatial functionality:

```
$ createdb  <db name>
$ psql <db name>
> CREATE EXTENSION postgis;
```

The database user must be a superuser in order to run `CREATE EXTENSION postgis;`. The command is run during the *`migrate`* process. An alternative is to use a migration operation in your project:

```
from django.contrib.postgres.operations import CreateExtension
from django.db import migrations

class Migration(migrations.Migration):

    operations = [
        CreateExtension('postgis'),
        ...
    ]
```

GeoDjango does not currently leverage any PostGIS topology functionality. If you plan to use those features at some point, you can also install the `postgis_topology` extension by issuing `CREATE EXTENSION postgis_topology;`.

Managing the database To administer the database, you can either use the pgAdmin III program (*Start* → *PostgreSQL 9.x* → *pgAdmin III*) or the SQL Shell (*Start* → *PostgreSQL 9.x* → *SQL Shell*). For example, to create a `geodjango` spatial database and user, the following may be executed from the SQL Shell as the `postgres` user:

```
postgres# CREATE USER geodjango PASSWORD 'my_passwd';
postgres# CREATE DATABASE geodjango OWNER geodjango;
```

Installing SpatiaLite SpatiaLite adds spatial support to SQLite, turning it into a full-featured spatial database.

First, check if you can install SpatiaLite from system packages or binaries.

For example, on Debian-based distributions, try to install the `spatialite-bin` package. For distributions that package SpatiaLite 4.2+, install `libsqlite3-mod-spatialite`.

For Mac OS X, follow the *instructions below*.

For Windows, you may find binaries on the Gaia-SINS home page.

In any case, you should always be able to *install from source*.

`SPATIALITE_LIBRARY_PATH` setting required for SpatiaLite 4.2+

If you're using SpatiaLite 4.2+, you must put this in your settings:

```
SPATIALITE_LIBRARY_PATH = 'mod_spatialite'
```

Installing from source GEOS and PROJ.4 should be installed prior to building SpatiaLite.

SQLite Check first if SQLite is compiled with the R*Tree module. Run the sqlite3 command line interface and enter the following query:

```
sqlite> CREATE VIRTUAL TABLE testrtree USING rtree(id,minX,maxX,minY,maxY);
```

If you obtain an error, you will have to recompile SQLite from source. Otherwise, just skip this section.

To install from sources, download the latest amalgamation source archive from the SQLite download page, and extract:

```
$ wget https://sqlite.org/sqlite-amalgamation-3.6.23.1.tar.gz
$ tar xzf sqlite-amalgamation-3.6.23.1.tar.gz
$ cd sqlite-3.6.23.1
```

Next, run the `configure` script – however the `CFLAGS` environment variable needs to be customized so that SQLite knows to build the R*Tree module:

```
$ CFLAGS="-DSQLITE_ENABLE_RTREE=1" ./configure
$ make
$ sudo make install
$ cd ..
```

SpatiaLite library (`libspatialite`) Get the latest SpatiaLite library source bundle from the download page:

```
$ wget http://www.gaia-gis.it/gaia-sins/libspatialite-sources/libspatialite-4.1.0.tar.gz
$ tar xaf libspatialite-4.1.0.tar.gz
$ cd libspatialite-4.1.0
$ ./configure
$ make
$ sudo make install
```

: For Mac OS X users building from source, the SpatiaLite library *and* tools need to have their `target` configured:

```
$ ./configure --target=macosx
```

Mac OS X-specific instructions To install the SpatiaLite library and tools, Mac OS X users can choose between *KyngChaos packages* and Homebrew.

KyngChaos First, follow the instructions in the *KyngChaos packages* section.

When creating a SpatiaLite database, the `spatialite` program is required. However, instead of attempting to compile the SpatiaLite tools from source, download the SpatiaLite Binaries for OS X, and install `spatialite` in a location available in your `PATH`. For example:

```
$ curl -O http://www.gaia-gis.it/spatialite/spatialite-tools-osx-x86-2.3.1.tar.gz
$ tar xzf spatialite-tools-osx-x86-2.3.1.tar.gz
$ cd spatialite-tools-osx-x86-2.3.1/bin
$ sudo cp spatialite /Library/Frameworks/SQLite3.framework/Programs
```

Finally, for GeoDjango to be able to find the KyngChaos SpatiaLite library, add the following to your `settings.py`:

```
SPATIALITE_LIBRARY_PATH='/Library/Frameworks/SQLite3.framework/SQLite3'
```

Homebrew Homebrew handles all the SpatiaLite related packages on your behalf, including SQLite3, SpatiaLite, PROJ, and GEOS. Install them like this:

```
$ brew update
$ brew install spatialite-tools
$ brew install gdal
```

Finally, for GeoDjango to be able to find the SpatiaLite library, add the following to your `settings.py`:

```
SPATIALITE_LIBRARY_PATH='/usr/local/lib/mod_spatialite.dylib'
```

Add `django.contrib.gis` to `INSTALLED_APPS` Like other Django contrib applications, you will *only* need to add *`django.contrib.gis`* to *`INSTALLED_APPS`* in your settings. This is so that the `gis` templates can be located – if not done, then features such as the geographic admin or KML sitemaps will not function properly.

Troubleshooting

If you can't find the solution to your problem here then participate in the community! You can:

- Join the `#geodjango` IRC channel on Freenode. Please be patient and polite – while you may not get an immediate response, someone will attempt to answer your question as soon as they see it.
- Ask your question on the GeoDjango mailing list.
- File a ticket on the Django trac if you think there's a bug. Make sure to provide a complete description of the problem, versions used, and specify the component as "GIS".

Library environment settings By far, the most common problem when installing GeoDjango is that the external shared libraries (e.g., for GEOS and GDAL) cannot be located. [1] Typically, the cause of this problem is that the operating system isn't aware of the directory where the libraries built from source were installed.

In general, the library path may be set on a per-user basis by setting an environment variable, or by configuring the library path for the entire system.

`LD_LIBRARY_PATH` environment variable A user may set this environment variable to customize the library paths they want to use. The typical library directory for software built from source is `/usr/local/lib`. Thus, `/usr/local/lib` needs to be included in the `LD_LIBRARY_PATH` variable. For example, the user could place the following in their bash profile:

```
export LD_LIBRARY_PATH=/usr/local/lib
```

[1] GeoDjango uses the `find_library()` routine from `ctypes.util` to locate shared libraries.

Setting system library path On GNU/Linux systems, there is typically a file in `/etc/ld.so.conf`, which may include additional paths from files in another directory, such as `/etc/ld.so.conf.d`. As the root user, add the custom library path (like `/usr/local/lib`) on a new line in `ld.so.conf`. This is *one* example of how to do so:

```
$ sudo echo /usr/local/lib >> /etc/ld.so.conf
$ sudo ldconfig
```

For OpenSolaris users, the system library path may be modified using the `crle` utility. Run `crle` with no options to see the current configuration and use `crle -l` to set with the new library path. Be *very* careful when modifying the system library path:

```
# crle -l $OLD_PATH:/usr/local/lib
```

Install `binutils` GeoDjango uses the `find_library` function (from the `ctypes.util` Python module) to discover libraries. The `find_library` routine uses a program called `objdump` (part of the `binutils` package) to verify a shared library on GNU/Linux systems. Thus, if `binutils` is not installed on your Linux system then Python's ctypes may not be able to find your library even if your library path is set correctly and geospatial libraries were built perfectly.

The `binutils` package may be installed on Debian and Ubuntu systems using the following command:

```
$ sudo apt-get install binutils
```

Similarly, on Red Hat and CentOS systems:

```
$ sudo yum install binutils
```

Platform-specific instructions

Mac OS X Because of the variety of packaging systems available for OS X, users have several different options for installing GeoDjango. These options are:

- *Postgres.app* (easiest and recommended)
- *Homebrew*
- *KyngChaos packages*
- *Fink*
- *MacPorts*
- *Building from source*

This section also includes instructions for installing an upgraded version of *Python* from packages provided by the Python Software Foundation, however, this is not required.

Python Although OS X comes with Python installed, users can use framework installers provided by the Python Software Foundation. An advantage to using the installer is that OS X's Python will remain "pristine" for internal operating system use.

: You will need to modify the `PATH` environment variable in your `.profile` file so that the new version of Python is used when `python` is entered at the command-line:

```
export PATH=/Library/Frameworks/Python.framework/Versions/Current/bin:$PATH
```

Postgres.app Postgres.app is a standalone PostgreSQL server that includes the PostGIS extension. You will also need to install `gdal` and `libgeoip` with *Homebrew*.

After installing Postgres.app, add the following to your `.bash_profile` so you can run the package's programs from the command-line. Replace `X.Y` with the version of PostgreSQL in the Postgres.app you installed:

```
export PATH=$PATH:/Applications/Postgres.app/Contents/Versions/X.Y/bin
```

You can check if the path is set up correctly by typing `which psql` at a terminal prompt.

Homebrew Homebrew provides "recipes" for building binaries and packages from source. It provides recipes for the GeoDjango prerequisites on Macintosh computers running OS X. Because Homebrew still builds the software from source, the Apple Developer Tools are required.

Summary:

```
$ brew install postgresql
$ brew install postgis
$ brew install gdal
$ brew install libgeoip
```

KyngChaos packages William Kyngesburye provides a number of geospatial library binary packages that make it simple to get GeoDjango installed on OS X without compiling them from source. However, the Apple Developer Tools are still necessary for compiling the Python database adapters *psycopg2* (for PostGIS) and pysqlite2 (for SpatiaLite).

: SpatiaLite users should consult the *Mac OS X-specific instructions* section after installing the packages for additional instructions.

Download the framework packages for:

- UnixImageIO
- PROJ
- GEOS
- SQLite3 (includes the SpatiaLite library)
- GDAL

Install the packages in the order they are listed above, as the GDAL and SQLite packages require the packages listed before them.

Afterwards, you can also install the KyngChaos binary packages for PostgreSQL and PostGIS.

After installing the binary packages, you'll want to add the following to your `.profile` to be able to run the package programs from the command-line:

```
export PATH=/Library/Frameworks/UnixImageIO.framework/Programs:$PATH
export PATH=/Library/Frameworks/PROJ.framework/Programs:$PATH
export PATH=/Library/Frameworks/GEOS.framework/Programs:$PATH
export PATH=/Library/Frameworks/SQLite3.framework/Programs:$PATH
export PATH=/Library/Frameworks/GDAL.framework/Programs:$PATH
export PATH=/usr/local/pgsql/bin:$PATH
```

psycopg2 After you've installed the KyngChaos binaries and modified your `PATH`, as described above, `psycopg2` may be installed using the following command:

```
$ pip install psycopg2
```

: If you don't have `pip`, follow the *installation instructions* to install it.

Fink Kurt Schwehr has been gracious enough to create GeoDjango packages for users of the Fink package system. Different packages are available (starting with "django-gis"), depending on which version of Python you want to use.

MacPorts MacPorts may be used to install GeoDjango prerequisites on Macintosh computers running OS X. Because MacPorts still builds the software from source, the Apple Developer Tools are required.

Summary:

```
$ sudo port install postgresql93-server
$ sudo port install geos
$ sudo port install proj
$ sudo port install postgis
$ sudo port install gdal +geos
$ sudo port install libgeoip
```

: You will also have to modify the `PATH` in your `.profile` so that the MacPorts programs are accessible from the command-line:

```
export PATH=/opt/local/bin:/opt/local/lib/postgresql93/bin
```

In addition, add the `DYLD_FALLBACK_LIBRARY_PATH` setting so that the libraries can be found by Python:

```
export DYLD_FALLBACK_LIBRARY_PATH=/opt/local/lib:/opt/local/lib/postgresql93
```

Windows Proceed through the following sections sequentially in order to install GeoDjango on Windows.

: These instructions assume that you are using 32-bit versions of all programs. While 64-bit versions of Python and PostgreSQL 9.x are available, 64-bit versions of spatial libraries, like GEOS and GDAL, are not yet provided by the *OSGeo4W* installer.

Python First, download the latest Python 2.7 installer from the Python website. Next, run the installer and keep the defaults – for example, keep 'Install for all users' checked and the installation path set as `C:\Python27`.

: You may already have a version of Python installed in `C:\python` as ESRI products sometimes install a copy there. *You should still install a fresh version of Python 2.7.*

PostgreSQL First, download the latest PostgreSQL 9.x installer from the EnterpriseDB website. After downloading, simply run the installer, follow the on-screen directions, and keep the default options unless you know the consequences of changing them.

: The PostgreSQL installer creates both a new Windows user to be the 'postgres service account' and a `postgres` database superuser You will be prompted once to set the password for both accounts – make sure to remember it!

When the installer completes, it will ask to launch the Application Stack Builder (ASB) on exit – keep this checked, as it is necessary to install *PostGIS*.

: If installed successfully, the PostgreSQL server will run in the background each time the system as started as a Windows service. A *PostgreSQL 9.x* start menu group will created and contains shortcuts for the ASB as well as the 'SQL Shell', which will launch a `psql` command window.

PostGIS From within the Application Stack Builder (to run outside of the installer, *Start → Programs → PostgreSQL 9.x*), select *PostgreSQL Database Server 9.x on port 5432* from the drop down menu. Next, expand the *Categories → Spatial Extensions* menu tree and select *PostGIS X.Y for PostgreSQL 9.x*.

After clicking next, you will be prompted to select your mirror, PostGIS will be downloaded, and the PostGIS installer will begin. Select only the default options during install (e.g., do not uncheck the option to create a default PostGIS database).

: You will be prompted to enter your `postgres` database superuser password in the 'Database Connection Information' dialog.

psycopg2 The `psycopg2` Python module provides the interface between Python and the PostgreSQL database. Download the latest Windows installer for your version of Python and PostgreSQL and run using the default settings. [2]

OSGeo4W The OSGeo4W installer makes it simple to install the PROJ.4, GDAL, and GEOS libraries required by GeoDjango. First, download the OSGeo4W installer, and run it. Select *Express Web-GIS Install* and click next. In the 'Select Packages' list, ensure that GDAL is selected; MapServer and Apache are also enabled by default, but are not required by GeoDjango and may be unchecked safely. After clicking next, the packages will be automatically downloaded and installed, after which you may exit the installer.

Modify Windows environment In order to use GeoDjango, you will need to add your Python and OSGeo4W directories to your Windows system `Path`, as well as create `GDAL_DATA` and `PROJ_LIB` environment variables. The following set of commands, executable with `cmd.exe`, will set this up:

```
set OSGEO4W_ROOT=C:\OSGeo4W
set PYTHON_ROOT=C:\Python27
set GDAL_DATA=%OSGEO4W_ROOT%\share\gdal
set PROJ_LIB=%OSGEO4W_ROOT%\share\proj
set PATH=%PATH%;%PYTHON_ROOT%;%OSGEO4W_ROOT%\bin
reg ADD "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Environment" /v Path /t REG_EXPAND_SZ
reg ADD "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Environment" /v GDAL_DATA /t REG_EXPAN
reg ADD "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Environment" /v PROJ_LIB /t REG_EXPANI
```

For your convenience, these commands are available in the executable batch script, `geodjango_setup.bat`.

[2] The `psycopg2` Windows installers are packaged and maintained by Jason Erickson.

: Administrator privileges are required to execute these commands. To do this, right-click on `geodjango_setup.bat` and select *Run as administrator*. You need to log out and log back in again for the settings to take effect.

: If you customized the Python or OSGeo4W installation directories, then you will need to modify the `OSGEO4W_ROOT` and/or `PYTHON_ROOT` variables accordingly.

Install Django and set up database Finally, *install Django* on your system.

GeoDjango Model API

This document explores the details of the GeoDjango Model API. Throughout this section, we'll be using the following geographic model of a ZIP code and of a Digital Elevation Model as our examples:

```
from django.contrib.gis.db import models

class Zipcode(models.Model):
    code = models.CharField(max_length=5)
    poly = models.PolygonField()

class Elevation(models.Model):
    name = models.CharField(max_length=100)
    rast = models.RasterField()
```

Spatial Field Types

Spatial fields consist of a series of geometry field types and one raster field type. Each of the geometry field types correspond to the OpenGIS Simple Features specification [1]. There is no such standard for raster data.

`GeometryField`
class `GeometryField`

`PointField`
class `PointField`

`LineStringField`
class `LineStringField`

`PolygonField`
class `PolygonField`

`MultiPointField`
class `MultiPointField`

[1] OpenGIS Consortium, Inc., Simple Feature Specification For SQL.

MultiLineStringField
class MultiLineStringField

MultiPolygonField
class MultiPolygonField

GeometryCollectionField
class GeometryCollectionField

RasterField
class RasterField
`RasterField` is currently only implemented for the PostGIS backend.

Spatial Field Options

The geometry field options `srid` and `spatial_index` are now shared by `GeometryField` and `RasterField` through the `BaseSpatialField`.

In addition to the regular *Field options* available for Django model fields, spatial fields have the following additional options. All are optional.

srid
`BaseSpatialField.`**srid**
Sets the SRID [2] (Spatial Reference System Identity) of the geometry field to the given value. Defaults to 4326 (also known as WGS84, units are in degrees of longitude and latitude).

Selecting an SRID Choosing an appropriate SRID for your model is an important decision that the developer should consider carefully. The SRID is an integer specifier that corresponds to the projection system that will be used to interpret the data in the spatial database. [3] Projection systems give the context to the coordinates that specify a location. Although the details of geodesy are beyond the scope of this documentation, the general problem is that the earth is spherical and representations of the earth (e.g., paper maps, Web maps) are not.

Most people are familiar with using latitude and longitude to reference a location on the earth's surface. However, latitude and longitude are angles, not distances. In other words, while the shortest path between two points on a flat surface is a straight line, the shortest path between two points on a curved surface (such as the earth) is an *arc* of a great circle. [4] Thus, additional computation is required to obtain distances in planar units (e.g., kilometers and miles). Using a geographic coordinate system may introduce complications for the developer later on. For example, SpatiaLite does not have the capability to perform distance calculations between geometries using geographic coordinate systems, e.g. constructing a query to find all points within 5 miles of a county boundary stored as WGS84. [5]

Portions of the earth's surface may projected onto a two-dimensional, or Cartesian, plane. Projected coordinate systems are especially convenient for region-specific applications, e.g., if you know that your database will only cover geometries in North Kansas, then you may consider using projection system specific to that region. Moreover, projected coordinate systems are defined in Cartesian units (such as meters or feet), easing distance calculations.

[2] *See id.* at Ch. 2.3.8, p. 39 (Geometry Values and Spatial Reference Systems).
[3] Typically, SRID integer corresponds to an EPSG (European Petroleum Survey Group) identifier. However, it may also be associated with custom projections defined in spatial database's spatial reference systems table.
[4] Terry A. Slocum, Robert B. McMaster, Fritz C. Kessler, & Hugh H. Howard, *Thematic Cartography and Geographic Visualization* (Prentice Hall, 2nd edition), at Ch. 7.1.3.
[5] This limitation does not apply to PostGIS.

: If you wish to perform arbitrary distance queries using non-point geometries in WGS84 in PostGIS and you want decent performance, enable the *GeometryField.geography* keyword so that *geography database type* is used instead.

Additional Resources:

- spatialreference.org: A Django-powered database of spatial reference systems.
- The State Plane Coordinate System: A website covering the various projection systems used in the United States. Much of the U.S. spatial data encountered will be in one of these coordinate systems rather than in a geographic coordinate system such as WGS84.

spatial_index
BaseSpatialField.**spatial_index**
Defaults to True. Creates a spatial index for the given geometry field.

: This is different from the db_index field option because spatial indexes are created in a different manner than regular database indexes. Specifically, spatial indexes are typically created using a variant of the R-Tree, while regular database indexes typically use B-Trees.

Geometry Field Options

There are additional options available for Geometry fields. All the following options are optional.

dim
GeometryField.**dim**
This option may be used for customizing the coordinate dimension of the geometry field. By default, it is set to 2, for representing two-dimensional geometries. For spatial backends that support it, it may be set to 3 for three-dimensional support.

: At this time 3D support is limited to the PostGIS spatial backend.

geography
GeometryField.**geography**
If set to True, this option will create a database column of type geography, rather than geometry. Please refer to the *geography type* section below for more details.

: Geography support is limited to PostGIS and will force the SRID to be 4326.

Geography Type The geography type provides native support for spatial features represented with geographic coordinates (e.g., WGS84 longitude/latitude). [6] Unlike the plane used by a geometry type, the geography type uses a spherical representation of its data. Distance and measurement operations performed on a geography column automatically employ great circle arc calculations and return linear units. In other words, when ST_Distance is called on two geographies, a value in meters is returned (as opposed to degrees if called on a geometry column in WGS84).

[6] Please refer to the PostGIS Geography Type documentation for more details.

Because geography calculations involve more mathematics, only a subset of the PostGIS spatial lookups are available for the geography type. Practically, this means that in addition to the *distance lookups* only the following additional *spatial lookups* are available for geography columns:

- *bboverlaps*
- *coveredby*
- *covers*
- *intersects*

If you need to use a spatial lookup or aggregate that doesn't support the geography type as input, you can use the *Cast* database function to convert the geography column to a geometry type in the query:

```
from django.contrib.gis.db.models import PointField
from django.db.models.functions import Cast

Zipcode.objects.annotate(
    geom=Cast('geography_field', PointField())
).filter(geom__within=poly)
```

For more information, the PostGIS documentation contains a helpful section on determining when to use geography data type over geometry data type.

GeoManager

class GeoManager

The `GeoManager` is required in order to use the legacy *GeoQuerySet Methods*.

1.9 : All `GeoQuerySet` methods have been deprecated and replaced by equivalent database functions. As soon as the legacy methods have been replaced in your code, you should be able to remove the special `GeoManager` from your GIS-enabled classes.

In older versions, the manager was required to conduct geographic queries. Without it, all geographic filters failed.

`GeoManager` was required even if the model did not have a geographic field itself, e.g., in the case of a `ForeignKey` relation to a model with a geographic field. For example, if we had an `Address` model with a `ForeignKey` to our `Zipcode` model:

```
from django.contrib.gis.db import models

class Address(models.Model):
    num = models.IntegerField()
    street = models.CharField(max_length=100)
    city = models.CharField(max_length=100)
    state = models.CharField(max_length=2)
    zipcode = models.ForeignKey(Zipcode, on_delete=models.CASCADE)
    objects = models.GeoManager()
```

The geographic manager was needed to do spatial queries on related `Zipcode` objects, for example:

```
qs = Address.objects.filter(zipcode__poly__contains='POINT(-104.590948 38.319914)')
```

GeoDjango Database API

Spatial Backends

GeoDjango currently provides the following spatial database backends:

- `django.contrib.gis.db.backends.postgis`
- `django.contrib.gis.db.backends.mysql`
- `django.contrib.gis.db.backends.oracle`
- `django.contrib.gis.db.backends.spatialite`

MySQL Spatial Limitations MySQL's spatial extensions only support bounding box operations (what MySQL calls minimum bounding rectangles, or MBR). Specifically, MySQL does not conform to the OGC standard:

> Currently, MySQL does not implement these functions [`Contains`, `Crosses`, `Disjoint`, `Intersects`, `Overlaps`, `Touches`, `Within`] according to the specification. Those that are implemented return the same result as the corresponding MBR-based functions.

In other words, while spatial lookups such as *`contains`* are available in GeoDjango when using MySQL, the results returned are really equivalent to what would be returned when using *`bbcontains`* on a different spatial backend.

: True spatial indexes (R-trees) are only supported with MyISAM tables on MySQL. [e] In other words, when using MySQL spatial extensions you have to choose between fast spatial lookups and the integrity of your data – MyISAM tables do not support transactions or foreign key constraints.

[e] *See* Creating Spatial Indexes in the MySQL Reference Manual:

> For MyISAM tables, `SPATIAL INDEX` creates an R-tree index. For storage engines that support nonspatial indexing of spatial columns, the engine creates a B-tree index. A B-tree index on spatial values will be useful for exact-value lookups, but not for range scans.

Raster Support `RasterField` is currently only implemented for the PostGIS backend. Spatial lookups are available for raster fields, but spatial database functions and aggregates aren't implemented for raster fields.

`RasterField` now supports spatial lookups.

Creating and Saving Models with Geometry Fields

Here is an example of how to create a geometry object (assuming the `Zipcode` model):

```
>>> from zipcode.models import Zipcode
>>> z = Zipcode(code=77096, poly='POLYGON(( 10 10, 10 20, 20 20, 20 15, 10 10))')
>>> z.save()
```

`GEOSGeometry` objects may also be used to save geometric models:

```
>>> from django.contrib.gis.geos import GEOSGeometry
>>> poly = GEOSGeometry('POLYGON(( 10 10, 10 20, 20 20, 20 15, 10 10))')
>>> z = Zipcode(code=77096, poly=poly)
>>> z.save()
```

Moreover, if the `GEOSGeometry` is in a different coordinate system (has a different SRID value) than that of the field, then it will be implicitly transformed into the SRID of the model's field, using the spatial database's transform procedure:

```
>>> poly_3084 = GEOSGeometry('POLYGON(( 10 10, 10 20, 20 20, 20 15, 10 10))', srid=3084)  # SRID 308
>>> z = Zipcode(code=78212, poly=poly_3084)
>>> z.save()
>>> from django.db import connection
>>> print(connection.queries[-1]['sql']) # printing the last SQL statement executed (requires DEBUG=
INSERT INTO "geoapp_zipcode" ("code", "poly") VALUES (78212, ST_Transform(ST_GeomFromWKB('\\001 ...
```

Thus, geometry parameters may be passed in using the `GEOSGeometry` object, WKT (Well Known Text [1]), HEX-EWKB (PostGIS specific – a WKB geometry in hexadecimal [2]), and GeoJSON [3] (requires GDAL). Essentially, if the input is not a `GEOSGeometry` object, the geometry field will attempt to create a `GEOSGeometry` instance from the input.

For more information creating *`GEOSGeometry`* objects, refer to the *GEOS tutorial*.

Creating and Saving Models with Raster Fields

When creating raster models, the raster field will implicitly convert the input into a *`GDALRaster`* using lazy-evaluation. The raster field will therefore accept any input that is accepted by the *`GDALRaster`* constructor.

Here is an example of how to create a raster object from a raster file `volcano.tif` (assuming the `Elevation` model):

```
>>> from elevation.models import Elevation
>>> dem = Elevation(name='Volcano', rast='/path/to/raster/volcano.tif')
>>> dem.save()
```

`GDALRaster` objects may also be used to save raster models:

```
>>> from django.contrib.gis.gdal import GDALRaster
>>> rast = GDALRaster({'width': 10, 'height': 10, 'name': 'Canyon', 'srid': 4326,
...                    'scale': [0.1, -0.1], 'bands': [{"data": range(100)}]})
>>> dem = Elevation(name='Canyon', rast=rast)
>>> dem.save()
```

Note that this equivalent to:

```
>>> dem = Elevation.objects.create(
...     name='Canyon',
...     rast={'width': 10, 'height': 10, 'name': 'Canyon', 'srid': 4326,
...           'scale': [0.1, -0.1], 'bands': [{"data": range(100)}]},
... )
```

Spatial Lookups

GeoDjango's lookup types may be used with any manager method like `filter()`, `exclude()`, etc. However, the lookup types unique to GeoDjango are only available on spatial fields.

Filters on 'normal' fields (e.g. *`CharField`*) may be chained with those on geographic fields. Geographic lookups accept geometry and raster input on both sides and input types can be mixed freely.

The general structure of geographic lookups is described below. A complete reference can be found in the *spatial lookup reference*.

[1] *See* Open Geospatial Consortium, Inc., OpenGIS Simple Feature Specification For SQL, Document 99-049 (May 5, 1999), at Ch. 3.2.5, p. 3-11 (SQL Textual Representation of Geometry).

[2] *See* PostGIS EWKB, EWKT and Canonical Forms, PostGIS documentation at Ch. 4.1.2.

[3] *See* Howard Butler, Martin Daly, Allan Doyle, Tim Schaub, & Christopher Schmidt, The GeoJSON Format Specification, Revision 1.0 (June 16, 2008).

Geometry Lookups Geographic queries with geometries take the following general form (assuming the `Zipcode` model used in the GeoDjango Model API):

```
>>> qs = Zipcode.objects.filter(<field>__<lookup_type>=<parameter>)
>>> qs = Zipcode.objects.exclude(...)
```

For example:

```
>>> qs = Zipcode.objects.filter(poly__contains=pnt)
>>> qs = Elevation.objects.filter(poly__contains=rst)
```

In this case, `poly` is the geographic field, *`contains`* is the spatial lookup type, `pnt` is the parameter (which may be a *`GEOSGeometry`* object or a string of GeoJSON , WKT, or HEXEWKB), and `rst` is a *`GDALRaster`* object.

Raster Lookups The raster lookup syntax is similar to the syntax for geometries. The only difference is that a band index can be specified as additional input. If no band index is specified, the first band is used by default (index `0`). In that case the syntax is identical to the syntax for geometry lookups.

To specify the band index, an additional parameter can be specified on both sides of the lookup. On the left hand side, the double underscore syntax is used to pass a band index. On the right hand side, a tuple of the raster and band index can be specified.

This results in the following general form for lookups involving rasters (assuming the `Elevation` model used in the GeoDjango Model API):

```
>>> qs = Elevation.objects.filter(<field>__<lookup_type>=<parameter>)
>>> qs = Elevation.objects.filter(<field>__<band_index>__<lookup_type>=<parameter>)
>>> qs = Elevation.objects.filter(<field>__<lookup_type>=(<raster_input, <band_index>)
```

For example:

```
>>> qs = Elevation.objects.filter(rast__contains=geom)
>>> qs = Elevation.objects.filter(rast__contains=rst)
>>> qs = Elevation.objects.filter(rast__1__contains=geom)
>>> qs = Elevation.objects.filter(rast__contains=(rst, 1))
>>> qs = Elevation.objects.filter(rast__1__contains=(rst, 1))
```

On the left hand side of the example, `rast` is the geographic raster field and *`contains`* is the spatial lookup type. On the right hand side, `geom` is a geometry input and `rst` is a *`GDALRaster`* object. The band index defaults to `0` in the first two queries and is set to `1` on the others.

While all spatial lookups can be used with raster objects on both sides, not all underlying operators natively accept raster input. For cases where the operator expects geometry input, the raster is automatically converted to a geometry. It's important to keep this in mind when interpreting the lookup results.

The type of raster support is listed for all lookups in the *compatibility table*. Lookups involving rasters are currently only available for the PostGIS backend.

Distance Queries

Introduction Distance calculations with spatial data is tricky because, unfortunately, the Earth is not flat. Some distance queries with fields in a geographic coordinate system may have to be expressed differently because of limitations in PostGIS. Please see the *Selecting an SRID* section in the GeoDjango Model API documentation for more details.

Distance Lookups *Availability*: PostGIS, Oracle, SpatiaLite, PGRaster (Native)

The following distance lookups are available:

- *distance_lt*
- *distance_lte*
- *distance_gt*
- *distance_gte*
- *dwithin*

: For *measuring*, rather than querying on distances, use the *Distance* function.

Distance lookups take a tuple parameter comprising:

1. A geometry or raster to base calculations from; and
2. A number or *Distance* object containing the distance.

If a *Distance* object is used, it may be expressed in any units (the SQL generated will use units converted to those of the field); otherwise, numeric parameters are assumed to be in the units of the field.

: In PostGIS, `ST_Distance_Sphere` does *not* limit the geometry types geographic distance queries are performed with. [4] However, these queries may take a long time, as great-circle distances must be calculated on the fly for *every* row in the query. This is because the spatial index on traditional geometry fields cannot be used.

For much better performance on WGS84 distance queries, consider using *geography columns* in your database instead because they are able to use their spatial index in distance queries. You can tell GeoDjango to use a geography column by setting `geography=True` in your field definition.

For example, let's say we have a `SouthTexasCity` model (from the GeoDjango distance tests) on a *projected* coordinate system valid for cities in southern Texas:

```
from django.contrib.gis.db import models

class SouthTexasCity(models.Model):
    name = models.CharField(max_length=30)
    # A projected coordinate system (only valid for South Texas!)
    # is used, units are in meters.
    point = models.PointField(srid=32140)
```

Then distance queries may be performed as follows:

```
>>> from django.contrib.gis.geos import GEOSGeometry
>>> from django.contrib.gis.measure import D # ``D`` is a shortcut for ``Distance``
>>> from geoapp.models import SouthTexasCity
# Distances will be calculated from this point, which does not have to be projected.
>>> pnt = GEOSGeometry('POINT(-96.876369 29.905320)', srid=4326)
# If numeric parameter, units of field (meters in this case) are assumed.
>>> qs = SouthTexasCity.objects.filter(point__distance_lte=(pnt, 7000))
# Find all Cities within 7 km, > 20 miles away, and > 100 chains away (an obscure unit)
>>> qs = SouthTexasCity.objects.filter(point__distance_lte=(pnt, D(km=7)))
>>> qs = SouthTexasCity.objects.filter(point__distance_gte=(pnt, D(mi=20)))
>>> qs = SouthTexasCity.objects.filter(point__distance_gte=(pnt, D(chain=100)))
```

Raster queries work the same way by simply replacing the geometry field `point` with a raster field, or the `pnt` object with a raster object, or both. To specify the band index of a raster input on the right hand side, a 3-tuple can be passed to the lookup as follows:

[4] *See* PostGIS documentation on `ST_DistanceSphere`.

```
>>> qs = SouthTexasCity.objects.filter(point__distance_gte=(rst, 2, D(km=7)))
```

Where the band with index 2 (the third band) of the raster `rst` would be used for the lookup.

Compatibility Tables

Spatial Lookups The following table provides a summary of what spatial lookups are available for each spatial database backend. The PostGIS Raster (PGRaster) lookups are divided into the three categories described in the *raster lookup details*: native support `N`, bilateral native support `B`, and geometry conversion support `C`.

Lookup Type	PostGIS	Oracle	MySQL [6]	SpatiaLite	PGRaster
bbcontains	X		X	X	N
bboverlaps	X		X	X	N
contained	X		X	X	N
contains	X	X	X	X	B
contains_properly	X				B
coveredby	X	X			B
covers	X	X			B
crosses	X			X	C
disjoint	X	X	X	X	B
distance_gt	X	X		X	N
distance_gte	X	X		X	N
distance_lt	X	X		X	N
distance_lte	X	X		X	N
dwithin	X	X			B
equals	X	X	X	X	C
exact	X	X	X	X	B
intersects	X	X	X	X	B
isvalid	X				
overlaps	X	X	X	X	B
relate	X	X		X	C
same_as	X	X	X	X	B
touches	X	X	X	X	B
within	X	X	X	X	B
left	X				C
right	X				C
overlaps_left	X				B
overlaps_right	X				B
overlaps_above	X				C
overlaps_below	X				C
strictly_above	X				C
strictly_below	X				C

Database functions The following table provides a summary of what geography-specific database functions are available on each spatial backend.

[6] Refer *MySQL Spatial Limitations* section for more details.

Function	PostGIS	Oracle	MySQL	SpatiaLite
Area	X	X	X	X
AsGeoJSON	X			X
AsGML	X	X		X
AsKML	X			X
AsSVG	X			X
BoundingCircle	X			
Centroid	X	X	X	X
Difference	X	X	X (≥ 5.6.1)	X
Distance	X	X	X (≥ 5.6.1)	X
Envelope	X		X	X
ForceRHR	X			
GeoHash	X			X (≥ 4.0, LWGEOM)
Intersection	X	X	X (≥ 5.6.1)	X
IsValid	X			
Length	X	X	X	X
MakeValid	X			
MemSize	X			
NumGeometries	X	X	X	X
NumPoints	X	X	X	X
Perimeter	X	X		X (≥ 4.0)
PointOnSurface	X	X		X
Reverse	X	X		X (≥ 4.0)
Scale	X			X
SnapToGrid	X			X (≥ 3.1)
SymDifference	X	X	X (≥ 5.6.1)	X
Transform	X	X		X
Translate	X			X
Union	X	X	X (≥ 5.6.1)	X

Aggregate Functions The following table provides a summary of what GIS-specific aggregate functions are available on each spatial backend. Please note that MySQL does not support any of these aggregates, and is thus excluded from the table.

Aggregate	PostGIS	Oracle	SpatiaLite
Collect	X		X
Extent	X	X	X
Extent3D	X		
MakeLine	X		X
Union	X	X	X

GeoDjango Forms API

GeoDjango provides some specialized form fields and widgets in order to visually display and edit geolocalized data on a map. By default, they use OpenLayers-powered maps, with a base WMS layer provided by Metacarta.

Field arguments

In addition to the regular *form field arguments*, GeoDjango form fields take the following optional arguments.

srid
Field.**srid**
This is the SRID code that the field value should be transformed to. For example, if the map widget SRID is different from the SRID more generally used by your application or database, the field will automatically convert input values into that SRID.

geom_type
Field.**geom_type**
You generally shouldn't have to set or change that attribute which should be setup depending on the field class. It matches the OpenGIS standard geometry name.

Form field classes

GeometryField
class GeometryField

PointField
class PointField

LineStringField
class LineStringField

PolygonField
class PolygonField

MultiPointField
class MultiPointField

MultiLineStringField
class MultiLineStringField

MultiPolygonField
class MultiPolygonField

GeometryCollectionField
class GeometryCollectionField

Form widgets

GeoDjango form widgets allow you to display and edit geographic data on a visual map. Note that none of the currently available widgets supports 3D geometries, hence geometry fields will fallback using a simple `Textarea` widget for such data.

Widget attributes GeoDjango widgets are template-based, so their attributes are mostly different from other Django widget attributes.

`BaseGeometryWidget.`**`geom_type`**
The OpenGIS geometry type, generally set by the form field.

`BaseGeometryWidget.`**`map_height`**

`BaseGeometryWidget.`**`map_width`**
Height and width of the widget map (default is 400x600).

`BaseGeometryWidget.`**`map_srid`**
SRID code used by the map (default is 4326).

`BaseGeometryWidget.`**`display_raw`**
Boolean value specifying if a textarea input showing the serialized representation of the current geometry is visible, mainly for debugging purposes (default is `False`).

`BaseGeometryWidget.`**`supports_3d`**
Indicates if the widget supports edition of 3D data (default is `False`).

`BaseGeometryWidget.`**`template_name`**
The template used to render the map widget.

You can pass widget attributes in the same manner that for any other Django widget. For example:

```
from django.contrib.gis import forms

class MyGeoForm(forms.Form):
    point = forms.PointField(widget=
        forms.OSMWidget(attrs={'map_width': 800, 'map_height': 500}))
```

Widget classes `BaseGeometryWidget`

class `BaseGeometryWidget`
This is an abstract base widget containing the logic needed by subclasses. You cannot directly use this widget for a geometry field. Note that the rendering of GeoDjango widgets is based on a template, identified by the *`template_name`* class attribute.

`OpenLayersWidget`

class `OpenLayersWidget`
This is the default widget used by all GeoDjango form fields. `template_name` is `gis/openlayers.html`.

`OpenLayersWidget` and *`OSMWidget`* use the `openlayers.js` file hosted on the `openlayers.org` website. This works for basic usage during development, but isn't appropriate for a production deployment as `openlayers.org/api/` has no guaranteed uptime and runs on a slow server. You are therefore advised to subclass these widgets in order to specify your own version of the `openlayers.js` file in the `js` property of the inner `Media` class (see *Assets as a static definition*). You can host a copy of `openlayers.js` tailored to your needs on your own server or refer to a copy from a content-delivery network like https://cdnjs.com/. This will also allow you to serve the JavaScript file(s) using the `https` protocol if needed.

`OSMWidget`

class `OSMWidget`
This widget uses an OpenStreetMap base layer (Mapnik) to display geographic objects on. `template_name` is `gis/openlayers-osm.html`.

The *`OpenLayersWidget`* note about JavaScript file hosting above also applies here. See also this FAQ answer about `https` access to map tiles.

GeoQuerySet API Reference

class GeoQuerySet(*model=None*)

Spatial Lookups

The spatial lookups in this section are available for *GeometryField* and *RasterField*.

For an introduction, see the *spatial lookups introduction*. For an overview of what lookups are compatible with a particular spatial backend, refer to the *spatial lookup compatibility table*.

Spatial lookups now support raster input.

Lookups with rasters All examples in the reference below are given for geometry fields and inputs, but the lookups can be used the same way with rasters on both sides. Whenever a lookup doesn't support raster input, the input is automatically converted to a geometry where necessary using the ST_Polygon function. See also the *introduction to raster lookups*.

The database operators used by the lookups can be divided into three categories:

- Native raster support `N`: the operator accepts rasters natively on both sides of the lookup, and raster input can be mixed with geometry inputs.
- Bilateral raster support `B`: the operator supports rasters only if both sides of the lookup receive raster inputs. Raster data is automatically converted to geometries for mixed lookups.
- Geometry conversion support `C`. The lookup does not have native raster support, all raster data is automatically converted to geometries.

The examples below show the SQL equivalent for the lookups in the different types of raster support. The same pattern applies to all spatial lookups.

Case	Lookup	SQL Equivalent
N, B	`rast__contains=rst`	`ST_Contains(rast, rst)`
N, B	`rast__1__contains=(rst, 2)`	`ST_Contains(rast, 1, rst, 2)`
B, C	`rast__contains=geom`	`ST_Contains(ST_Polygon(rast), geom)`
B, C	`rast__1__contains=geom`	`ST_Contains(ST_Polygon(rast, 1), geom)`
B, C	`poly__contains=rst`	`ST_Contains(poly, ST_Polygon(rst))`
B, C	`poly__contains=(rst, 1)`	`ST_Contains(poly, ST_Polygon(rst, 1))`
C	`rast__crosses=rst`	`ST_Crosses(ST_Polygon(rast), ST_Polygon(rst))`
C	`rast__1__crosses=(rst, 2)`	`ST_Crosses(ST_Polygon(rast, 1), ST_Polygon(rst, 2))`
C	`rast__crosses=geom`	`ST_Crosses(ST_Polygon(rast), geom)`
C	`poly__crosses=rst`	`ST_Crosses(poly, ST_Polygon(rst))`

Spatial lookups with rasters are only supported for PostGIS backends (denominated as PGRaster in this section).

bbcontains *Availability*: PostGIS, MySQL, SpatiaLite, PGRaster (Native)

Tests if the geometry or raster field's bounding box completely contains the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__bbcontains=geom)
```

Backend	SQL Equivalent
PostGIS	`poly ~ geom`
MySQL	`MBRContains(poly, geom)`
SpatiaLite	`MbrContains(poly, geom)`

`bboverlaps` *Availability*: PostGIS, MySQL, SpatiaLite, PGRaster (Native)

Tests if the geometry field's bounding box overlaps the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__bboverlaps=geom)
```

Backend	SQL Equivalent
PostGIS	`poly && geom`
MySQL	`MBROverlaps(poly, geom)`
SpatiaLite	`MbrOverlaps(poly, geom)`

`contained` *Availability*: PostGIS, MySQL, SpatiaLite, PGRaster (Native)

Tests if the geometry field's bounding box is completely contained by the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__contained=geom)
```

Backend	SQL Equivalent
PostGIS	`poly @ geom`
MySQL	`MBRWithin(poly, geom)`
SpatiaLite	`MbrWithin(poly, geom)`

`contains` *Availability*: PostGIS, Oracle, MySQL, SpatiaLite, PGRaster (Bilateral)

Tests if the geometry field spatially contains the lookup geometry.

Example:

```
Zipcode.objects.filter(poly__contains=geom)
```

Backend	SQL Equivalent
PostGIS	`ST_Contains(poly, geom)`
Oracle	`SDO_CONTAINS(poly, geom)`
MySQL	`MBRContains(poly, geom)`
SpatiaLite	`Contains(poly, geom)`

`contains_properly` *Availability*: PostGIS, PGRaster (Bilateral)

Returns true if the lookup geometry intersects the interior of the geometry field, but not the boundary (or exterior). [4]

Example:

```
Zipcode.objects.filter(poly__contains_properly=geom)
```

Backend	SQL Equivalent
PostGIS	`ST_ContainsProperly(poly, geom)`

[4] Refer to the PostGIS `ST_ContainsProperly` documentation for more details.

coveredby *Availability*: PostGIS, Oracle, PGRaster (Bilateral)

Tests if no point in the geometry field is outside the lookup geometry. [3]

Example:

```
Zipcode.objects.filter(poly__coveredby=geom)
```

Backend	SQL Equivalent
PostGIS	`ST_CoveredBy(poly, geom)`
Oracle	`SDO_COVEREDBY(poly, geom)`

covers *Availability*: PostGIS, Oracle, PGRaster (Bilateral)

Tests if no point in the lookup geometry is outside the geometry field. [3]

Example:

```
Zipcode.objects.filter(poly__covers=geom)
```

Backend	SQL Equivalent
PostGIS	`ST_Covers(poly, geom)`
Oracle	`SDO_COVERS(poly, geom)`

crosses *Availability*: PostGIS, SpatiaLite, PGRaster (Conversion)

Tests if the geometry field spatially crosses the lookup geometry.

Example:

```
Zipcode.objects.filter(poly__crosses=geom)
```

Backend	SQL Equivalent
PostGIS	`ST_Crosses(poly, geom)`
SpatiaLite	`Crosses(poly, geom)`

disjoint *Availability*: PostGIS, Oracle, MySQL, SpatiaLite, PGRaster (Bilateral)

Tests if the geometry field is spatially disjoint from the lookup geometry.

Example:

```
Zipcode.objects.filter(poly__disjoint=geom)
```

Backend	SQL Equivalent
PostGIS	`ST_Disjoint(poly, geom)`
Oracle	`SDO_GEOM.RELATE(poly, 'DISJOINT', geom, 0.05)`
MySQL	`MBRDisjoint(poly, geom)`
SpatiaLite	`Disjoint(poly, geom)`

equals *Availability*: PostGIS, Oracle, MySQL, SpatiaLite, PGRaster (Conversion)

exact, same_as *Availability*: PostGIS, Oracle, MySQL, SpatiaLite, PGRaster (Bilateral)

[3] For an explanation of this routine, read Quirks of the "Contains" Spatial Predicate by Martin Davis (a PostGIS developer).

intersects *Availability*: PostGIS, Oracle, MySQL, SpatiaLite, PGRaster (Bilateral)

Tests if the geometry field spatially intersects the lookup geometry.

Example:

```
Zipcode.objects.filter(poly__intersects=geom)
```

Backend	SQL Equivalent
PostGIS	`ST_Intersects(poly, geom)`
Oracle	`SDO_OVERLAPBDYINTERSECT(poly, geom)`
MySQL	`MBRIntersects(poly, geom)`
SpatiaLite	`Intersects(poly, geom)`

isvalid *Availability*: PostGIS

Tests if the geometry is valid.

Example:

```
Zipcode.objects.filter(poly__isvalid=True)
```

PostGIS equivalent:

```
SELECT ... WHERE ST_IsValid(poly)
```

overlaps *Availability*: PostGIS, Oracle, MySQL, SpatiaLite, PGRaster (Bilateral)

relate *Availability*: PostGIS, Oracle, SpatiaLite, PGRaster (Conversion)

Tests if the geometry field is spatially related to the lookup geometry by the values given in the given pattern. This lookup requires a tuple parameter, `(geom, pattern)`; the form of `pattern` will depend on the spatial backend:

PostGIS & SpatiaLite On these spatial backends the intersection pattern is a string comprising nine characters, which define intersections between the interior, boundary, and exterior of the geometry field and the lookup geometry. The intersection pattern matrix may only use the following characters: `1`, `2`, `T`, `F`, or `*`. This lookup type allows users to "fine tune" a specific geometric relationship consistent with the DE-9IM model. [1]

Geometry example:

```
# A tuple lookup parameter is used to specify the geometry and
# the intersection pattern (the pattern here is for 'contains').
Zipcode.objects.filter(poly__relate=(geom, 'T*T***FF*'))
```

PostGIS SQL equivalent:

```
SELECT ... WHERE ST_Relate(poly, geom, 'T*T***FF*')
```

SpatiaLite SQL equivalent:

```
SELECT ... WHERE Relate(poly, geom, 'T*T***FF*')
```

Raster example:

```
Zipcode.objects.filter(poly__relate=(rast, 1, 'T*T***FF*'))
Zipcode.objects.filter(rast__2__relate=(rast, 1, 'T*T***FF*'))
```

[1] *See* OpenGIS Simple Feature Specification For SQL, at Ch. 2.1.13.2, p. 2-13 (The Dimensionally Extended Nine-Intersection Model).

PostGIS SQL equivalent:

```
SELECT ... WHERE ST_Relate(poly, ST_Polygon(rast, 1), 'T*T***FF*')
SELECT ... WHERE ST_Relate(ST_Polygon(rast, 2), ST_Polygon(rast, 1), 'T*T***FF*')
```

Oracle Here the relation pattern is comprised of at least one of the nine relation strings: `TOUCH`, `OVERLAPBDYDISJOINT`, `OVERLAPBDYINTERSECT`, `EQUAL`, `INSIDE`, `COVEREDBY`, `CONTAINS`, `COVERS`, `ON`, and `ANYINTERACT`. Multiple strings may be combined with the logical Boolean operator OR, for example, `'inside+touch'`. [2] The relation strings are case-insensitive.

Example:

```
Zipcode.objects.filter(poly__relate=(geom, 'anyinteract'))
```

Oracle SQL equivalent:

```
SELECT ... WHERE SDO_RELATE(poly, geom, 'anyinteract')
```

`touches` *Availability*: PostGIS, Oracle, MySQL, SpatiaLite

Tests if the geometry field spatially touches the lookup geometry.

Example:

```
Zipcode.objects.filter(poly__touches=geom)
```

Backend	SQL Equivalent
PostGIS	`ST_Touches(poly, geom)`
MySQL	`MBRTouches(poly, geom)`
Oracle	`SDO_TOUCH(poly, geom)`
SpatiaLite	`Touches(poly, geom)`

`within` *Availability*: PostGIS, Oracle, MySQL, SpatiaLite, PGRaster (Bilateral)

Tests if the geometry field is spatially within the lookup geometry.

Example:

```
Zipcode.objects.filter(poly__within=geom)
```

Backend	SQL Equivalent
PostGIS	`ST_Within(poly, geom)`
MySQL	`MBRWithin(poly, geom)`
Oracle	`SDO_INSIDE(poly, geom)`
SpatiaLite	`Within(poly, geom)`

`left` *Availability*: PostGIS, PGRaster (Conversion)

Tests if the geometry field's bounding box is strictly to the left of the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__left=geom)
```

PostGIS equivalent:

[2] *See* SDO_RELATE documentation, from Ch. 11 of the Oracle Spatial User's Guide and Manual.

```
SELECT ... WHERE poly << geom
```

right *Availability*: PostGIS, PGRaster (Conversion)

Tests if the geometry field's bounding box is strictly to the right of the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__right=geom)
```

PostGIS equivalent:

```
SELECT ... WHERE poly >> geom
```

overlaps_left *Availability*: PostGIS, PGRaster (Bilateral)

Tests if the geometry field's bounding box overlaps or is to the left of the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__overlaps_left=geom)
```

PostGIS equivalent:

```
SELECT ... WHERE poly &< geom
```

overlaps_right *Availability*: PostGIS, PGRaster (Bilateral)

Tests if the geometry field's bounding box overlaps or is to the right of the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__overlaps_right=geom)
```

PostGIS equivalent:

```
SELECT ... WHERE poly &> geom
```

overlaps_above *Availability*: PostGIS, PGRaster (Conversion)

Tests if the geometry field's bounding box overlaps or is above the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__overlaps_above=geom)
```

PostGIS equivalent:

```
SELECT ... WHERE poly |&> geom
```

overlaps_below *Availability*: PostGIS, PGRaster (Conversion)

Tests if the geometry field's bounding box overlaps or is below the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__overlaps_below=geom)
```

PostGIS equivalent:

```
SELECT ... WHERE poly &<| geom
```

strictly_above *Availability*: PostGIS, PGRaster (Conversion)

Tests if the geometry field's bounding box is strictly above the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__strictly_above=geom)
```

PostGIS equivalent:

```
SELECT ... WHERE poly |>> geom
```

strictly_below *Availability*: PostGIS, PGRaster (Conversion)

Tests if the geometry field's bounding box is strictly below the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__strictly_below=geom)
```

PostGIS equivalent:

```
SELECT ... WHERE poly <<| geom
```

Distance Lookups

Availability: PostGIS, Oracle, SpatiaLite, PGRaster (Native)

For an overview on performing distance queries, please refer to the *distance queries introduction*.

Distance lookups take the following form:

```
<field>__<distance lookup>=(<geometry/raster>, <distance value>[, 'spheroid'])
<field>__<distance lookup>=(<raster>, <band_index>, <distance value>[, 'spheroid'])
<field>__<band_index>__<distance lookup>=(<raster>, <band_index>, <distance value>[, 'spheroid'])
```

The value passed into a distance lookup is a tuple; the first two values are mandatory, and are the geometry to calculate distances to, and a distance value (either a number in units of the field, a `Distance` object, or a *query expression* <*ref/models/expressions*>). To pass a band index to the lookup, use a 3-tuple where the second entry is the band index.

With PostGIS, on every distance lookup but *dwithin*, an optional element, `'spheroid'`, may be included to tell GeoDjango to use the more accurate spheroid distance calculation functions on fields with a geodetic coordinate system (e.g., `ST_Distance_Spheroid` would be used instead of `ST_Distance_Sphere`). The simpler `ST_Distance` function is used with projected coordinate systems. Rasters are converted to geometries for spheroid based lookups.

The ability to pass an expression as the distance value was added.

distance_gt Returns models where the distance to the geometry field from the lookup geometry is greater than the given distance value.

Example:

```
Zipcode.objects.filter(poly__distance_gt=(geom, D(m=5)))
```

Backend	SQL Equivalent
PostGIS	`ST_Distance/ST_Distance_Sphere(poly, geom) > 5`
Oracle	`SDO_GEOM.SDO_DISTANCE(poly, geom, 0.05) > 5`
SpatiaLite	`Distance(poly, geom) > 5`

distance_gte Returns models where the distance to the geometry field from the lookup geometry is greater than or equal to the given distance value.

Example:

```
Zipcode.objects.filter(poly__distance_gte=(geom, D(m=5)))
```

Backend	SQL Equivalent
PostGIS	`ST_Distance/ST_Distance_Sphere(poly, geom) >= 5`
Oracle	`SDO_GEOM.SDO_DISTANCE(poly, geom, 0.05) >= 5`
SpatiaLite	`Distance(poly, geom) >= 5`

distance_lt Returns models where the distance to the geometry field from the lookup geometry is less than the given distance value.

Example:

```
Zipcode.objects.filter(poly__distance_lt=(geom, D(m=5)))
```

Backend	SQL Equivalent
PostGIS	`ST_Distance/ST_Distance_Sphere(poly, geom) < 5`
Oracle	`SDO_GEOM.SDO_DISTANCE(poly, geom, 0.05) < 5`
SpatiaLite	`Distance(poly, geom) < 5`

distance_lte Returns models where the distance to the geometry field from the lookup geometry is less than or equal to the given distance value.

Example:

```
Zipcode.objects.filter(poly__distance_lte=(geom, D(m=5)))
```

Backend	SQL Equivalent
PostGIS	`ST_Distance/ST_Distance_Sphere(poly, geom) <= 5`
Oracle	`SDO_GEOM.SDO_DISTANCE(poly, geom, 0.05) <= 5`
SpatiaLite	`Distance(poly, geom) <= 5`

dwithin Returns models where the distance to the geometry field from the lookup geometry are within the given distance from one another. Note that you can only provide *Distance* objects if the targeted geometries are in a projected system. For geographic geometries, you should use units of the geometry field (e.g. degrees for `WGS84`) .

Example:

```
Zipcode.objects.filter(poly__dwithin=(geom, D(m=5)))
```

Backend	SQL Equivalent
PostGIS	`ST_DWithin(poly, geom, 5)`
Oracle	`SDO_WITHIN_DISTANCE(poly, geom, 5)`

: This lookup is not available on SpatiaLite.

`GeoQuerySet` Methods

1.9 : Using `GeoQuerySet` methods is now deprecated in favor of the new Geographic Database Functions. Albeit a little more verbose, they are much more powerful in how it is possible to combine them to build more complex queries.

`GeoQuerySet` methods specify that a spatial operation be performed on each spatial operation on each geographic field in the queryset and store its output in a new attribute on the model (which is generally the name of the `GeoQuerySet` method).

There are also aggregate `GeoQuerySet` methods which return a single value instead of a queryset. This section will describe the API and availability of every `GeoQuerySet` method available in GeoDjango.

: What methods are available depend on your spatial backend. See the *compatibility table* for more details.

With a few exceptions, the following keyword arguments may be used with all `GeoQuerySet` methods:

Keyword Argument	Description
`field_name`	By default, `GeoQuerySet` methods use the first geographic field encountered in the model. This keyword should be used to specify another geographic field (e.g., `field_name='point2'`) when there are multiple geographic fields in a model. On PostGIS, the `field_name` keyword may also be used on geometry fields in models that are related via a `ForeignKey` relation (e.g., `field_name='related__point'`).
`model_att`	By default, `GeoQuerySet` methods typically attach their output in an attribute with the same name as the `GeoQuerySet` method. Setting this keyword with the desired attribute name will override this default behavior. For example, `qs = Zipcode.objects.centroid(model_att='c')` will attach the centroid of the `Zipcode` geometry field in a `c` attribute on every model rather than in a `centroid` attribute. This keyword is required if a method name clashes with an existing `GeoQuerySet` method – if you wanted to use the `area()` method on model with a `PolygonField` named `area`, for example.

Measurement *Availability*: PostGIS, Oracle, SpatiaLite

`area`

`GeoQuerySet.`**`area`**`(`***kwargs*`)`
1.9 : Use the *`Area`* function instead.

Returns the area of the geographic field in an `area` attribute on each element of this GeoQuerySet.

`distance`

`GeoQuerySet.`**`distance`**`(`*geom*, ***kwargs*`)`
1.9 : Use the *`Distance`* function instead.

This method takes a geometry as a parameter, and attaches a `distance` attribute to every model in the returned queryset that contains the distance (as a *`Distance`* object) to the given geometry.

In the following example (taken from the GeoDjango distance tests), the distance from the Tasmanian city of Hobart to every other *`PointField`* in the `AustraliaCity` queryset is calculated:

```
>>> pnt = AustraliaCity.objects.get(name='Hobart').point
>>> for city in AustraliaCity.objects.distance(pnt): print(city.name, city.distance)
Wollongong 990071.220408 m
Shellharbour 972804.613941 m
Thirroul 1002334.36351 m
Mittagong 975691.632637 m
Batemans Bay 834342.185561 m
Canberra 598140.268959 m
Melbourne 575337.765042 m
Sydney 1056978.87363 m
Hobart 0.0 m
Adelaide 1162031.83522 m
Hillsdale 1049200.46122 m
```

: Because the `distance` attribute is a *`Distance`* object, you can easily express the value in the units of your choice. For example, `city.distance.mi` is the distance value in miles and `city.distance.km` is the distance value in kilometers. See Measurement Objects for usage details and the list of *Supported units*.

`length`
`GeoQuerySet.`**`length`**`(`***kwargs*`)`
1.9 : Use the *`Length`* function instead.

Returns the length of the geometry field in a `length` attribute (a *`Distance`* object) on each model in the queryset.

`perimeter`
`GeoQuerySet.`**`perimeter`**`(`***kwargs*`)`
1.9 : Use the *`Perimeter`* function instead.

Returns the perimeter of the geometry field in a `perimeter` attribute (a *`Distance`* object) on each model in the queryset.

Geometry Relationships The following methods take no arguments, and attach geometry objects each element of the *`GeoQuerySet`* that is the result of relationship function evaluated on the geometry field.

`centroid`
`GeoQuerySet.`**`centroid`**`(`***kwargs*`)`
1.9 : Use the *`Centroid`* function instead.

Availability: PostGIS, Oracle, SpatiaLite

Returns the `centroid` value for the geographic field in a `centroid` attribute on each element of the `GeoQuerySet`.

`envelope`
`GeoQuerySet.`**`envelope`**`(`***kwargs*`)`

1.9 : Use the *Envelope* function instead.

Availability: PostGIS, SpatiaLite

Returns a geometry representing the bounding box of the geometry field in an `envelope` attribute on each element of the `GeoQuerySet`.

`point_on_surface`
`GeoQuerySet.`**`point_on_surface`**`(**kwargs)`
1.9 : Use the *PointOnSurface* function instead.

Availability: PostGIS, Oracle, SpatiaLite

Returns a Point geometry guaranteed to lie on the surface of the geometry field in a `point_on_surface` attribute on each element of the queryset; otherwise sets with None.

Geometry Editors

`force_rhr`
`GeoQuerySet.`**`force_rhr`**`(**kwargs)`
1.9 : Use the *ForceRHR* function instead.

Availability: PostGIS

Returns a modified version of the polygon/multipolygon in which all of the vertices follow the Right-Hand-Rule, and attaches as a `force_rhr` attribute on each element of the queryset.

`reverse_geom`
`GeoQuerySet.`**`reverse_geom`**`(**kwargs)`
1.9 : Use the *Reverse* function instead.

Availability: PostGIS, Oracle

Reverse the coordinate order of the geometry field, and attaches as a `reverse` attribute on each element of the queryset.

`scale`
`GeoQuerySet.`**`scale`**`(x, y, z=0.0, **kwargs)`
1.9 : Use the *Scale* function instead.

Availability: PostGIS, SpatiaLite

`snap_to_grid`
`GeoQuerySet.`**`snap_to_grid`**`(*args, **kwargs)`
1.9 : Use the *SnapToGrid* function instead.

Snap all points of the input geometry to the grid. How the geometry is snapped to the grid depends on how many numeric (either float, integer, or long) arguments are given.

Number of Arguments	Description
1	A single size to snap bot the X and Y grids to.
2	X and Y sizes to snap the grid to.
4	X, Y sizes and the corresponding X, Y origins.

transform

GeoQuerySet.**transform**(*srid=4326*, ***kwargs*)
1.9 : Use the *Transform* function instead.

Availability: PostGIS, Oracle, SpatiaLite

The `transform` method transforms the geometry field of a model to the spatial reference system specified by the `srid` parameter. If no `srid` is given, then 4326 (WGS84) is used by default.

: Unlike other `GeoQuerySet` methods, `transform` stores its output "in-place". In other words, no new attribute for the transformed geometry is placed on the models.

: What spatial reference system an integer SRID corresponds to may depend on the spatial database used. In other words, the SRID numbers used for Oracle are not necessarily the same as those used by PostGIS.

Example:

```
>>> qs = Zipcode.objects.all().transform() # Transforms to WGS84
>>> qs = Zipcode.objects.all().transform(32140) # Transforming to "NAD83 / Texas South Central"
>>> print(qs[0].poly.srid)
32140
>>> print(qs[0].poly)
POLYGON ((234055.1698884720099159 4937796.9232223574072123 ...
```

translate

GeoQuerySet.**translate**(*x*, *y*, *z=0.0*, ***kwargs*)
1.9 : Use the *Translate* function instead.

Availability: PostGIS, SpatiaLite

Translates the geometry field to a new location using the given numeric parameters as offsets.

Geometry Operations *Availability*: PostGIS, Oracle, SpatiaLite

The following methods all take a geometry as a parameter and attach a geometry to each element of the `GeoQuerySet` that is the result of the operation.

difference

GeoQuerySet.**difference**(*geom*)
1.9 : Use the *Difference* function instead.

Returns the spatial difference of the geographic field with the given geometry in a `difference` attribute on each element of the `GeoQuerySet`.

intersection

GeoQuerySet.**intersection**(*geom*)
1.9 : Use the *Intersection* function instead.

Returns the spatial intersection of the geographic field with the given geometry in an `intersection` attribute on each element of the `GeoQuerySet`.

sym_difference

GeoQuerySet.**sym_difference**(*geom*)
1.9 : Use the *SymDifference* function instead.

Returns the symmetric difference of the geographic field with the given geometry in a `sym_difference` attribute on each element of the `GeoQuerySet`.

union

GeoQuerySet.**union**(*geom*)
1.9 : Use the *Union* function instead.

Returns the union of the geographic field with the given geometry in an `union` attribute on each element of the `GeoQuerySet`.

Geometry Output The following `GeoQuerySet` methods will return an attribute that has the value of the geometry field in each model converted to the requested output format.

geohash

GeoQuerySet.**geohash**(*precision=20*, ***kwargs*)
1.9 : Use the *GeoHash* function instead.

Attaches a `geohash` attribute to every model the queryset containing the GeoHash representation of the geometry.

geojson

GeoQuerySet.**geojson**(***kwargs*)
1.9 : Use the *AsGeoJSON* function instead.

Availability: PostGIS, SpatiaLite

Attaches a `geojson` attribute to every model in the queryset that contains the GeoJSON representation of the geometry.

Keyword Argument	Description
`precision`	It may be used to specify the number of significant digits for the coordinates in the GeoJSON representation – the default value is 8.
`crs`	Set this to `True` if you want the coordinate reference system to be included in the returned GeoJSON.
`bbox`	Set this to `True` if you want the bounding box to be included in the returned GeoJSON.

gml

GeoQuerySet.**gml**(***kwargs*)
1.9 : Use the *AsGML* function instead.

Availability: PostGIS, Oracle, SpatiaLite

Attaches a `gml` attribute to every model in the queryset that contains the Geographic Markup Language (GML) representation of the geometry.

Example:

```
>>> qs = Zipcode.objects.all().gml()
>>> print(qs[0].gml)
<gml:Polygon srsName="EPSG:4326"><gml:OuterBoundaryIs>-147.78711,70.245363 ...  -147.78711,70.245363<
```

Keyword Argument	Description
`precision`	This keyword is for PostGIS only. It may be used to specify the number of significant digits for the coordinates in the GML representation – the default value is 8.
`version`	This keyword is for PostGIS only. It may be used to specify the GML version used, and may only be values of 2 or 3. The default value is 2.

kml

`GeoQuerySet.`**`kml`**`(**kwargs)`
1.9 : Use the *AsKML* function instead.

Availability: PostGIS, SpatiaLite

Attaches a `kml` attribute to every model in the queryset that contains the Keyhole Markup Language (KML) representation of the geometry fields. It should be noted that the contents of the KML are transformed to WGS84 if necessary.

Example:

```
>>> qs = Zipcode.objects.all().kml()
>>> print(qs[0].kml)
<Polygon><outerBoundaryIs><LinearRing><coordinates>-103.04135,36.217596,0 ... -103.04135,36.217596,0
```

Keyword Argument	Description
`precision`	This keyword may be used to specify the number of significant digits for the coordinates in the KML representation – the default value is 8.

svg

`GeoQuerySet.`**`svg`**`(**kwargs)`
1.9 : Use the *AsSVG* function instead.

Availability: PostGIS, SpatiaLite

Attaches a `svg` attribute to every model in the queryset that contains the Scalable Vector Graphics (SVG) path data of the geometry fields.

Keyword Argument	Description
`relative`	If set to `True`, the path data will be implemented in terms of relative moves. Defaults to `False`, meaning that absolute moves are used instead.
`precision`	This keyword may be used to specify the number of significant digits for the coordinates in the SVG representation – the default value is 8.

Miscellaneous

mem_size

`GeoQuerySet.`**`mem_size`**`(**kwargs)`
1.9 : Use the *MemSize* function instead.

Availability: PostGIS

Returns the memory size (number of bytes) that the geometry field takes in a `mem_size` attribute on each element of the `GeoQuerySet`.

num_geom

GeoQuerySet.**num_geom**(***kwargs*)
1.9 : Use the *NumGeometries* function instead.

Availability: PostGIS, Oracle, SpatiaLite

Returns the number of geometries in a `num_geom` attribute on each element of the `GeoQuerySet` if the geometry field is a collection (e.g., a `GEOMETRYCOLLECTION` or `MULTI*` field); otherwise sets with `None`.

num_points

GeoQuerySet.**num_points**(***kwargs*)
1.9 : Use the *NumPoints* function instead.

Availability: PostGIS, Oracle, SpatiaLite

Returns the number of points in the first linestring in the geometry field in a `num_points` attribute on each element of the `GeoQuerySet`; otherwise sets with `None`.

Aggregate Functions Django provides some GIS-specific aggregate functions. For details on how to use these aggregate functions, see the topic guide on aggregation.

Keyword Argument	Description
`tolerance`	This keyword is for Oracle only. It is for the tolerance value used by the `SDOAGGRTYPE` procedure; the Oracle documentation has more details.

Example:

```
>>> from django.contrib.gis.db.models import Extent, Union
>>> WorldBorder.objects.aggregate(Extent('mpoly'), Union('mpoly'))
```

Collect

class Collect(*geo_field*)
Availability: PostGIS, SpatiaLite

Returns a `GEOMETRYCOLLECTION` or a `MULTI` geometry object from the geometry column. This is analogous to a simplified version of the *Union* aggregate, except it can be several orders of magnitude faster than performing a union because it simply rolls up geometries into a collection or multi object, not caring about dissolving boundaries.

Extent

class Extent(*geo_field*)
Availability: PostGIS, Oracle, SpatiaLite

Returns the extent of all `geo_field` in the `QuerySet` as a four-tuple, comprising the lower left coordinate and the upper right coordinate.

Example:

```
>>> qs = City.objects.filter(name__in=('Houston', 'Dallas')).aggregate(Extent('poly'))
>>> print(qs['poly__extent'])
(-96.8016128540039, 29.7633724212646, -95.3631439208984, 32.782058715820)
```

Extent3D

class Extent3D(*geo_field*)

Availability: PostGIS

Returns the 3D extent of all `geo_field` in the `QuerySet` as a six-tuple, comprising the lower left coordinate and upper right coordinate (each with x, y, and z coordinates).

Example:

```
>>> qs = City.objects.filter(name__in=('Houston', 'Dallas')).aggregate(Extent3D('poly'))
>>> print(qs['poly__extent3d'])
(-96.8016128540039, 29.7633724212646, 0, -95.3631439208984, 32.782058715820, 0)
```

MakeLine

class MakeLine(*geo_field*)

Availability: PostGIS, SpatiaLite

Returns a `LineString` constructed from the point field geometries in the `QuerySet`. Currently, ordering the queryset has no effect.

SpatiaLite support was added.

Example:

```
>>> qs = City.objects.filter(name__in=('Houston', 'Dallas')).aggregate(MakeLine('poly'))
>>> print(qs['poly__makeline'])
LINESTRING (-95.3631510000000020 29.7633739999999989, -96.8016109999999941 32.7820570000000018)
```

Union

class Union(*geo_field*)

Availability: PostGIS, Oracle, SpatiaLite

This method returns a *GEOSGeometry* object comprising the union of every geometry in the queryset. Please note that use of `Union` is processor intensive and may take a significant amount of time on large querysets.

: If the computation time for using this method is too expensive, consider using *Collect* instead.

Example:

```
>>> u = Zipcode.objects.aggregate(Union(poly))  # This may take a long time.
>>> u = Zipcode.objects.filter(poly__within=bbox).aggregate(Union(poly))  # A more sensible approach
```

Geographic Database Functions

The functions documented on this page allow users to access geographic database functions to be used in annotations, aggregations, or filters in Django.

Example:

```
>>> from django.contrib.gis.db.models.functions import Length
>>> Track.objects.annotate(length=Length('line')).filter(length__gt=100)
```

Not all backends support all functions, so refer to the documentation of each function to see if your database backend supports the function you want to use. If you call a geographic function on a backend that doesn't support it, you'll get a `NotImplementedError` exception.

Function's summary:

Measure-ment	Relation-ships	Operations	Editors	Output format	Miscella-neous
Area	*BoundingCircle*	*Difference*	*ForceRHR*	*AsGeoJSON*	*IsValid*
Distance	*Centroid*	*Intersection*	*MakeValid*	*AsGML*	*MemSize*
Length	*Envelope*	*SymDifference*	*Reverse*	*AsKML*	*NumGeometries*
Perimeter	*PointOnSurface*	*Union*	*Scale* *SnapToGrid* *Transform* *Translate*	*AsSVG* *GeoHash*	*NumPoints*

Area

class Area(*expression*, ***extra*)

Availability: MySQL, Oracle, PostGIS, SpatiaLite

Accepts a single geographic field or expression and returns the area of the field as an *Area* measure. On MySQL, a raw float value is returned, as it's not possible to automatically determine the unit of the field.

AsGeoJSON

class AsGeoJSON(*expression*, *bbox=False*, *crs=False*, *precision=8*, ***extra*)

Availability: PostGIS, SpatiaLite

Accepts a single geographic field or expression and returns a GeoJSON representation of the geometry. Note that the result is not a complete GeoJSON structure but only the `geometry` key content of a GeoJSON structure. See also GeoJSON Serializer.

Example:

```
>>> City.objects.annotate(json=AsGeoJSON('point')).get(name='Chicago').json
{"type":"Point","coordinates":[-87.65018,41.85039]}
```

Keyword Argument	Description
`bbox`	Set this to `True` if you want the bounding box to be included in the returned GeoJSON.
`crs`	Set this to `True` if you want the coordinate reference system to be included in the returned GeoJSON.
`precision`	It may be used to specify the number of significant digits for the coordinates in the GeoJSON representation – the default value is 8.

AsGML

class AsGML(*expression*, *version=2*, *precision=8*, ***extra*)

Availability: Oracle, PostGIS, SpatiaLite

Accepts a single geographic field or expression and returns a Geographic Markup Language (GML) representation of the geometry.

Example:

```
>>> qs = Zipcode.objects.annotate(gml=AsGML('poly'))
>>> print(qs[0].gml)
<gml:Polygon srsName="EPSG:4326"><gml:OuterBoundaryIs>-147.78711,70.245363 ...
-147.78711,70.245363</gml:OuterBoundaryIs></gml:Polygon>
```

Keyword Argument	Description
`precision`	Not used on Oracle. It may be used to specify the number of significant digits for the coordinates in the GML representation – the default value is 8.
`version`	Not used on Oracle. It may be used to specify the GML version used, and may only be values of 2 or 3. The default value is 2.

AsKML

class AsKML(*expression*, *precision=8*, ***extra*)

Availability: PostGIS, SpatiaLite

Accepts a single geographic field or expression and returns a Keyhole Markup Language (KML) representation of the geometry.

Example:

```
>>> qs = Zipcode.objects.annotate(kml=AsKML('poly'))
>>> print(qs[0].kml)
<Polygon><outerBoundaryIs><LinearRing><coordinates>-103.04135,36.217596,0 ...
-103.04135,36.217596,0</coordinates></LinearRing></outerBoundaryIs></Polygon>
```

Keyword Argument	Description
`precision`	This keyword may be used to specify the number of significant digits for the coordinates in the KML representation – the default value is 8.

AsSVG

class AsSVG(*expression*, *relative=False*, *precision=8*, ***extra*)

Availability: PostGIS, SpatiaLite

Accepts a single geographic field or expression and returns a Scalable Vector Graphics (SVG) representation of the geometry.

Keyword Argument	Description
`relative`	If set to `True`, the path data will be implemented in terms of relative moves. Defaults to `False`, meaning that absolute moves are used instead.
`precision`	This keyword may be used to specify the number of significant digits for the coordinates in the SVG representation – the default value is 8.

BoundingCircle

class BoundingCircle(*expression*, *num_seg=48*, ***extra*)

Availability: PostGIS

Accepts a single geographic field or expression and returns the smallest circle polygon that can fully contain the geometry.

Centroid

class Centroid(*expression*, ***extra*)

Availability: MySQL, PostGIS, Oracle, SpatiaLite

Accepts a single geographic field or expression and returns the `centroid` value of the geometry.

Difference

class Difference(*expr1*, *expr2*, ***extra*)

Availability: MySQL (≥ 5.6.1), PostGIS, Oracle, SpatiaLite

Accepts two geographic fields or expressions and returns the geometric difference, that is the part of geometry A that does not intersect with geometry B.

MySQL support was added.

Distance

class Distance(*expr1*, *expr2*, *spheroid=None*, ***extra*)

Availability: MySQL (≥ 5.6.1), PostGIS, Oracle, SpatiaLite

Accepts two geographic fields or expressions and returns the distance between them, as a *Distance* object. On MySQL, a raw float value is returned, as it's not possible to automatically determine the unit of the field.

On backends that support distance calculation on geodetic coordinates, the proper backend function is automatically chosen depending on the SRID value of the geometries (e.g. `ST_Distance_Sphere` on PostGIS).

When distances are calculated with geodetic (angular) coordinates, as is the case with the default WGS84 (4326) SRID, you can set the `spheroid` keyword argument to decide if the calculation should be based on a simple sphere (less accurate, less resource-intensive) or on a spheroid (more accurate, more resource-intensive).

In the following example, the distance from the city of Hobart to every other *PointField* in the `AustraliaCity` queryset is calculated:

```
>>> from django.contrib.gis.db.models.functions import Distance
>>> pnt = AustraliaCity.objects.get(name='Hobart').point
>>> for city in AustraliaCity.objects.annotate(distance=Distance('point', pnt)):
...     print(city.name, city.distance)
Wollongong 990071.220408 m
Shellharbour 972804.613941 m
Thirroul 1002334.36351 m
...
```

: Because the `distance` attribute is a *Distance* object, you can easily express the value in the units of your choice. For example, `city.distance.mi` is the distance value in miles and `city.distance.km` is the distance value in kilometers. See Measurement Objects for usage details and the list of *Supported units*.

Envelope

class Envelope(*expression*, ***extra*)

Availability: MySQL, PostGIS, SpatiaLite

Accepts a single geographic field or expression and returns the geometry representing the bounding box of the geometry.

ForceRHR

class ForceRHR(*expression*, ***extra*)

Availability: PostGIS

Accepts a single geographic field or expression and returns a modified version of the polygon/multipolygon in which all of the vertices follow the right-hand rule.

GeoHash

class GeoHash(*expression*, *precision=None*, ***extra*)

Availability: PostGIS, SpatiaLite (≥ 4.0, LWGEOM)

Accepts a single geographic field or expression and returns a GeoHash representation of the geometry.

The `precision` keyword argument controls the number of characters in the result.

SpatiaLite support was added.

Intersection

class Intersection(*expr1*, *expr2*, ***extra*)

Availability: MySQL (≥ 5.6.1), PostGIS, Oracle, SpatiaLite

Accepts two geographic fields or expressions and returns the geometric intersection between them.

MySQL support was added.

IsValid

class IsValid(*expr*)

Availability: PostGIS

Accepts a geographic field or expression and tests if the value is well formed. Returns `True` if its value is a valid geometry and `False` otherwise.

Length

class Length(*expression*, *spheroid=True*, ***extra*)

Availability: MySQL, Oracle, PostGIS, SpatiaLite

Accepts a single geographic linestring or multilinestring field or expression and returns its length as an *Distance* measure. On MySQL, a raw float value is returned, as it's not possible to automatically determine the unit of the field.

On PostGIS and SpatiaLite, when the coordinates are geodetic (angular), you can specify if the calculation should be based on a simple sphere (less accurate, less resource-intensive) or on a spheroid (more accurate, more resource-intensive) with the `spheroid` keyword argument.

MakeValid

class MakeValid(*expr*)

Availability: PostGIS

Accepts a geographic field or expression and attempts to convert the value into a valid geometry without losing any of the input vertices. Geometries that are already valid are returned without changes. Simple polygons might become a multipolygon and the result might be of lower dimension than the input.

MemSize

class MemSize(*expression*, ***extra*)

Availability: PostGIS

Accepts a single geographic field or expression and returns the memory size (number of bytes) that the geometry field takes.

NumGeometries

class NumGeometries(*expression*, ***extra*)

Availability: MySQL, PostGIS, Oracle, SpatiaLite

Accepts a single geographic field or expression and returns the number of geometries if the geometry field is a collection (e.g., a `GEOMETRYCOLLECTION` or `MULTI*` field); otherwise returns `None`.

NumPoints

class NumPoints(*expression*, ***extra*)

Availability: MySQL, PostGIS, Oracle, SpatiaLite

Accepts a single geographic field or expression and returns the number of points in the first linestring in the geometry field; otherwise returns `None`.

Perimeter

class Perimeter(*expression*, ***extra*)

Availability: PostGIS, Oracle, SpatiaLite ($\geq$ 4.0)

Accepts a single geographic field or expression and returns the perimeter of the geometry field as a *`Distance`* object. On MySQL, a raw float value is returned, as it's not possible to automatically determine the unit of the field.

PointOnSurface

class PointOnSurface(*expression*, ***extra*)

Availability: PostGIS, Oracle, SpatiaLite

Accepts a single geographic field or expression and returns a `Point` geometry guaranteed to lie on the surface of the field; otherwise returns `None`.

Reverse

class Reverse(*expression*, ***extra*)

Availability: PostGIS, Oracle, SpatiaLite (≥ 4.0)

Accepts a single geographic field or expression and returns a geometry with reversed coordinates.

Scale

class Scale(*expression*, *x*, *y*, *z=0.0*, ***extra*)

Availability: PostGIS, SpatiaLite

Accepts a single geographic field or expression and returns a geometry with scaled coordinates by multiplying them with the `x`, `y`, and optionally `z` parameters.

SnapToGrid

class SnapToGrid(*expression*, **args*, ***extra*)

Availability: PostGIS, SpatiaLite (≥ 3.1)

Accepts a single geographic field or expression and returns a geometry with all points snapped to the given grid. How the geometry is snapped to the grid depends on how many numeric (either float, integer, or long) arguments are given.

Number of Arguments	Description
1	A single size to snap both the X and Y grids to.
2	X and Y sizes to snap the grid to.
4	X, Y sizes and the corresponding X, Y origins.

SymDifference

class SymDifference(*expr1*, *expr2*, ***extra*)

Availability: MySQL (≥ 5.6.1), PostGIS, Oracle, SpatiaLite

Accepts two geographic fields or expressions and returns the geometric symmetric difference (union without the intersection) between the given parameters.

MySQL support was added.

Transform

class Transform(*expression*, *srid*, ***extra*)

Availability: PostGIS, Oracle, SpatiaLite

Accepts a geographic field or expression and a SRID integer code, and returns the transformed geometry to the spatial reference system specified by the `srid` parameter.

: What spatial reference system an integer SRID corresponds to may depend on the spatial database used. In other words, the SRID numbers used for Oracle are not necessarily the same as those used by PostGIS.

Translate

class Translate (*expression*, *x*, *y*, *z=0.0*, ***extra*)

Availability: PostGIS, SpatiaLite

Accepts a single geographic field or expression and returns a geometry with its coordinates offset by the `x`, `y`, and optionally `z` numeric parameters.

Union

class Union (*expr1*, *expr2*, ***extra*)

Availability: MySQL ($\geq$ 5.6.1), PostGIS, Oracle, SpatiaLite

Accepts two geographic fields or expressions and returns the union of both geometries.

Measurement Objects

The *django.contrib.gis.measure* module contains objects that allow for convenient representation of distance and area units of measure. [1] Specifically, it implements two objects, *Distance* and *Area* – both of which may be accessed via the *D* and *A* convenience aliases, respectively.

Example

Distance objects may be instantiated using a keyword argument indicating the context of the units. In the example below, two different distance objects are instantiated in units of kilometers (`km`) and miles (`mi`):

```
>>> from django.contrib.gis.measure import Distance, D
>>> d1 = Distance(km=5)
>>> print(d1)
5.0 km
>>> d2 = D(mi=5) # `D` is an alias for `Distance`
>>> print(d2)
5.0 mi
```

Conversions are easy, just access the preferred unit attribute to get a converted distance quantity:

```
>>> print(d1.mi) # Converting 5 kilometers to miles
3.10685596119
>>> print(d2.km) # Converting 5 miles to kilometers
8.04672
```

Moreover, arithmetic operations may be performed between the distance objects:

```
>>> print(d1 + d2) # Adding 5 miles to 5 kilometers
13.04672 km
>>> print(d2 - d1) # Subtracting 5 kilometers from 5 miles
1.89314403881 mi
```

Two *Distance* objects multiplied together will yield an *Area* object, which uses squared units of measure:

[1] Robert Coup is the initial author of the measure objects, and was inspired by Brian Beck's work in geopy and Geoff Biggs' PhD work on dimensioned units for robotics.

```
>>> a = d1 * d2 # Returns an Area object.
>>> print(a)
40.2336 sq_km
```

To determine what the attribute abbreviation of a unit is, the `unit_attname` class method may be used:

```
>>> print(Distance.unit_attname('US Survey Foot'))
survey_ft
>>> print(Distance.unit_attname('centimeter'))
cm
```

Supported units

Unit Attribute	Full name or alias(es)
km	Kilometre, Kilometer
mi	Mile
m	Meter, Metre
yd	Yard
ft	Foot, Foot (International)
survey_ft	U.S. Foot, US survey foot
inch	Inches
cm	Centimeter
mm	Millimetre, Millimeter
um	Micrometer, Micrometre
british_ft	British foot (Sears 1922)
british_yd	British yard (Sears 1922)
british_chain_sears	British chain (Sears 1922)
indian_yd	Indian yard, Yard (Indian)
sears_yd	Yard (Sears)
clarke_ft	Clarke's Foot
chain	Chain
chain_benoit	Chain (Benoit)
chain_sears	Chain (Sears)
british_chain_benoit	British chain (Benoit 1895 B)
british_chain_sears_truncated	British chain (Sears 1922 truncated)
gold_coast_ft	Gold Coast foot
link	Link
link_benoit	Link (Benoit)
link_sears	Link (Sears)
clarke_link	Clarke's link
fathom	Fathom
rod	Rod
nm	Nautical Mile
nm_uk	Nautical Mile (UK)
german_m	German legal metre

: *Area* attributes are the same as *Distance* attributes, except they are prefixed with `sq_` (area units are square in nature). For example, `Area(sq_m=2)` creates an *Area* object representing two square meters.

Measurement API

Distance

class Distance(***kwargs*)

To initialize a distance object, pass in a keyword corresponding to the desired *unit attribute name* set with desired value. For example, the following creates a distance object representing 5 miles:

```
>>> dist = Distance(mi=5)
```

__getattr__(*unit_att*)

Returns the distance value in units corresponding to the given unit attribute. For example:

```
>>> print(dist.km)
8.04672
```

classmethod unit_attname(*unit_name*)

Returns the distance unit attribute name for the given full unit name. For example:

```
>>> Distance.unit_attname('Mile')
'mi'
```

class D

Alias for *Distance* class.

Area

class Area(***kwargs*)

To initialize an area object, pass in a keyword corresponding to the desired *unit attribute name* set with desired value. For example, the following creates an area object representing 5 square miles:

```
>>> a = Area(sq_mi=5)
```

__getattr__(*unit_att*)

Returns the area value in units corresponding to the given unit attribute. For example:

```
>>> print(a.sq_km)
12.949940551680001
```

classmethod unit_attname(*unit_name*)

Returns the area unit attribute name for the given full unit name. For example:

```
>>> Area.unit_attname('Kilometer')
'sq_km'
```

class A

Alias for *Area* class.

GEOS API

Background

What is GEOS? GEOS stands for **Geometry Engine - Open Source**, and is a C++ library, ported from the Java Topology Suite. GEOS implements the OpenGIS Simple Features for SQL spatial predicate functions and spatial operators. GEOS, now an OSGeo project, was initially developed and maintained by Refractions Research of Victoria, Canada.

Features GeoDjango implements a high-level Python wrapper for the GEOS library, its features include:

- A BSD-licensed interface to the GEOS geometry routines, implemented purely in Python using `ctypes`.
- Loosely-coupled to GeoDjango. For example, *GEOSGeometry* objects may be used outside of a Django project/application. In other words, no need to have `DJANGO_SETTINGS_MODULE` set or use a database, etc.
- Mutability: *GEOSGeometry* objects may be modified.
- Cross-platform and tested; compatible with Windows, Linux, Solaris, and Mac OS X platforms.

Tutorial

This section contains a brief introduction and tutorial to using *GEOSGeometry* objects.

Creating a Geometry *GEOSGeometry* objects may be created in a few ways. The first is to simply instantiate the object on some spatial input – the following are examples of creating the same geometry from WKT, HEX, WKB, and GeoJSON:

```
>>> from django.contrib.gis.geos import GEOSGeometry
>>> pnt = GEOSGeometry('POINT(5 23)') # WKT
>>> pnt = GEOSGeometry('0101000000000000000000144000000000000003740') # HEX
>>> pnt = GEOSGeometry(buffer('\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x14@\x00\x00\x00\x00\x00
>>> pnt = GEOSGeometry('{ "type": "Point", "coordinates": [ 5.000000, 23.000000 ] }') # GeoJSON
```

Another option is to use the constructor for the specific geometry type that you wish to create. For example, a *Point* object may be created by passing in the X and Y coordinates into its constructor:

```
>>> from django.contrib.gis.geos import Point
>>> pnt = Point(5, 23)
```

All these constructors take the keyword argument `srid`. For example:

```
>>> from django.contrib.gis.geos import GEOSGeometry, LineString, Point
>>> print(GEOSGeometry('POINT (0 0)', srid=4326))
SRID=4326;POINT (0 0)
>>> print(LineString((0, 0), (1, 1), srid=4326))
SRID=4326;LINESTRING (0 0, 1 1)
>>> print(Point(0, 0, srid=32140))
SRID=32140;POINT (0 0)
```

Finally, there is the *fromfile()* factory method which returns a *GEOSGeometry* object from a file:

```
>>> from django.contrib.gis.geos import fromfile
>>> pnt = fromfile('/path/to/pnt.wkt')
>>> pnt = fromfile(open('/path/to/pnt.wkt'))
```

My logs are filled with GEOS-related errors

You find many `TypeError` or `AttributeError` exceptions filling your Web server's log files. This generally means that you are creating GEOS objects at the top level of some of your Python modules. Then, due to a race condition in the garbage collector, your module is garbage collected before the GEOS object. To prevent this, create *GEOSGeometry* objects inside the local scope of your functions/methods.

Geometries are Pythonic *GEOSGeometry* objects are 'Pythonic', in other words components may be accessed, modified, and iterated over using standard Python conventions. For example, you can iterate over the coordinates in a *Point*:

```
>>> pnt = Point(5, 23)
>>> [coord for coord in pnt]
[5.0, 23.0]
```

With any geometry object, the *GEOSGeometry.coords* property may be used to get the geometry coordinates as a Python tuple:

```
>>> pnt.coords
(5.0, 23.0)
```

You can get/set geometry components using standard Python indexing techniques. However, what is returned depends on the geometry type of the object. For example, indexing on a *LineString* returns a coordinate tuple:

```
>>> from django.contrib.gis.geos import LineString
>>> line = LineString((0, 0), (0, 50), (50, 50), (50, 0), (0, 0))
>>> line[0]
(0.0, 0.0)
>>> line[-2]
(50.0, 0.0)
```

Whereas indexing on a *Polygon* will return the ring (a *LinearRing* object) corresponding to the index:

```
>>> from django.contrib.gis.geos import Polygon
>>> poly = Polygon( ((0.0, 0.0), (0.0, 50.0), (50.0, 50.0), (50.0, 0.0), (0.0, 0.0)) )
>>> poly[0]
<LinearRing object at 0x1044395b0>
>>> poly[0][-2] # second-to-last coordinate of external ring
(50.0, 0.0)
```

In addition, coordinates/components of the geometry may added or modified, just like a Python list:

```
>>> line[0] = (1.0, 1.0)
>>> line.pop()
(0.0, 0.0)
>>> line.append((1.0, 1.0))
>>> line.coords
((1.0, 1.0), (0.0, 50.0), (50.0, 50.0), (50.0, 0.0), (1.0, 1.0))
```

Geometries support set-like operators:

```
>>> from django.contrib.gis.geos import LineString
>>> ls1 = LineString((0, 0), (2, 2))
>>> ls2 = LineString((1, 1), (3, 3))
>>> print(ls1 | ls2)  # equivalent to `ls1.union(ls2)`
MULTILINESTRING ((0 0, 1 1), (1 1, 2 2), (2 2, 3 3))
>>> print(ls1 & ls2)  # equivalent to `ls1.intersection(ls2)`
LINESTRING (1 1, 2 2)
>>> print(ls1 - ls2)  # equivalent to `ls1.difference(ls2)`
LINESTRING(0 0, 1 1)
>>> print(ls1 ^ ls2)  # equivalent to `ls1.sym_difference(ls2)`
MULTILINESTRING ((0 0, 1 1), (2 2, 3 3))
```

Equality operator doesn't check spatial equality

The *GEOSGeometry* equality operator uses *equals_exact()*, not *equals()*, i.e. it requires the compared geometries to have the same coordinates in the same positions:

```
>>> from django.contrib.gis.geos import LineString
>>> ls1 = LineString((0, 0), (1, 1))
>>> ls2 = LineString((1, 1), (0, 0))
>>> ls1.equals(ls2)
True
>>> ls1 == ls2
False
```

Geometry Objects

GEOSGeometry
class GEOSGeometry(*geo_input*, *srid=None*)

- **geo_input** – Geometry input value (string or buffer)
- **srid** (*int*) – spatial reference identifier

This is the base class for all GEOS geometry objects. It initializes on the given `geo_input` argument, and then assumes the proper geometry subclass (e.g., `GEOSGeometry('POINT(1 1)')` will create a *Point* object).

The following input formats, along with their corresponding Python types, are accepted:

Format	Input Type
WKT / EWKT	`str` or `unicode`
HEX / HEXEWKB	`str` or `unicode`
WKB / EWKB	`buffer`
GeoJSON (requires GDAL)	`str` or `unicode`

Properties

`GEOSGeometry.`**coords**
Returns the coordinates of the geometry as a tuple.

`GEOSGeometry.`**dims**
Returns the dimension of the geometry:

- `0` for *Point*s and *MultiPoint*s
- `1` for *LineString*s and *MultiLineString*s
- `2` for *Polygon*s and *MultiPolygon*s
- `-1` for empty *GeometryCollection*s
- the maximum dimension of its elements for non-empty *GeometryCollection*s

`GEOSGeometry.`**empty**
Returns whether or not the set of points in the geometry is empty.

`GEOSGeometry.`**geom_type**
Returns a string corresponding to the type of geometry. For example:

```
>>> pnt = GEOSGeometry('POINT(5 23)')
>>> pnt.geom_type
'Point'
```

`GEOSGeometry.`**geom_typeid**
Returns the GEOS geometry type identification number. The following table shows the value for each geometry type:

Geometry	ID
Point	0
LineString	1
LinearRing	2
Polygon	3
MultiPoint	4
MultiLineString	5
MultiPolygon	6
GeometryCollection	7

GEOSGeometry.**num_coords**
Returns the number of coordinates in the geometry.

GEOSGeometry.**num_geom**
Returns the number of geometries in this geometry. In other words, will return 1 on anything but geometry collections.

GEOSGeometry.**hasz**
Returns a boolean indicating whether the geometry is three-dimensional.

GEOSGeometry.**ring**
Returns a boolean indicating whether the geometry is a `LinearRing`.

GEOSGeometry.**simple**
Returns a boolean indicating whether the geometry is 'simple'. A geometry is simple if and only if it does not intersect itself (except at boundary points). For example, a *LineString* object is not simple if it intersects itself. Thus, *LinearRing* and *Polygon* objects are always simple because they do cannot intersect themselves, by definition.

GEOSGeometry.**valid**
Returns a boolean indicating whether the geometry is valid.

GEOSGeometry.**valid_reason**
Returns a string describing the reason why a geometry is invalid.

GEOSGeometry.**srid**
Property that may be used to retrieve or set the SRID associated with the geometry. For example:

```
>>> pnt = Point(5, 23)
>>> print(pnt.srid)
None
>>> pnt.srid = 4326
>>> pnt.srid
4326
```

Output Properties The properties in this section export the *GEOSGeometry* object into a different. This output may be in the form of a string, buffer, or even another object.

GEOSGeometry.**ewkt**
Returns the "extended" Well-Known Text of the geometry. This representation is specific to PostGIS and is a superset of the OGC WKT standard. [1] Essentially the SRID is prepended to the WKT representation, for example `SRID=4326;POINT(5 23)`.

: The output from this property does not include the 3dm, 3dz, and 4d information that PostGIS supports in its EWKT representations.

---

[1] *See* PostGIS EWKB, EWKT and Canonical Forms, PostGIS documentation at Ch. 4.1.2.

GEOSGeometry.**hex**
Returns the WKB of this Geometry in hexadecimal form. Please note that the SRID value is not included in this representation because it is not a part of the OGC specification (use the *GEOSGeometry.hexewkb* property instead).

GEOSGeometry.**hexewkb**
Returns the EWKB of this Geometry in hexadecimal form. This is an extension of the WKB specification that includes the SRID value that are a part of this geometry.

GEOSGeometry.**json**
Returns the GeoJSON representation of the geometry. Note that the result is not a complete GeoJSON structure but only the `geometry` key content of a GeoJSON structure. See also GeoJSON Serializer.

GEOSGeometry.**geojson**
Alias for *GEOSGeometry.json*.

GEOSGeometry.**kml**
Returns a KML (Keyhole Markup Language) representation of the geometry. This should only be used for geometries with an SRID of 4326 (WGS84), but this restriction is not enforced.

GEOSGeometry.**ogr**
Returns an *OGRGeometry* object corresponding to the GEOS geometry.

: Requires GDAL.

GEOSGeometry.**wkb**
Returns the WKB (Well-Known Binary) representation of this Geometry as a Python buffer. SRID value is not included, use the *GEOSGeometry.ewkb* property instead.

GEOSGeometry.**ewkb**
Return the EWKB representation of this Geometry as a Python buffer. This is an extension of the WKB specification that includes any SRID value that are a part of this geometry.

GEOSGeometry.**wkt**
Returns the Well-Known Text of the geometry (an OGC standard).

Non-significant zeros are stripped from the output.

Spatial Predicate Methods All of the following spatial predicate methods take another *GEOSGeometry* instance (`other`) as a parameter, and return a boolean.

GEOSGeometry.**contains**(*other*)
Returns `True` if *other.within(this)* returns `True`.

GEOSGeometry.**covers**(*other*)
Returns `True` if this geometry covers the specified geometry.

The `covers` predicate has the following equivalent definitions:

- Every point of the other geometry is a point of this geometry.
- The DE-9IM Intersection Matrix for the two geometries is `T*****FF*`, `*T****FF*`, `***T**FF*`, or `****T*FF*`.

If either geometry is empty, returns `False`.

This predicate is similar to *GEOSGeometry.contains()*, but is more inclusive (i.e. returns `True` for more cases). In particular, unlike *contains()* it does not distinguish between points in the boundary and in the interior of geometries. For most situations, `covers()` should be preferred to *contains()*. As an added benefit, `covers()` is more amenable to optimization and hence should outperform *contains()*.

GEOSGeometry.**crosses**(*other*)
Returns `True` if the DE-9IM intersection matrix for the two Geometries is `T*T******` (for a point and a curve,a point and an area or a line and an area) `0********` (for two curves).

GEOSGeometry.**disjoint**(*other*)
Returns `True` if the DE-9IM intersection matrix for the two geometries is `FF*FF****`.

GEOSGeometry.**equals**(*other*)
Returns `True` if the DE-9IM intersection matrix for the two geometries is `T*F**FFF*`.

GEOSGeometry.**equals_exact**(*other*, *tolerance=0*)
Returns true if the two geometries are exactly equal, up to a specified tolerance. The `tolerance` value should be a floating point number representing the error tolerance in the comparison, e.g., `poly1.equals_exact(poly2, 0.001)` will compare equality to within one thousandth of a unit.

GEOSGeometry.**intersects**(*other*)
Returns `True` if *GEOSGeometry.disjoint()* is `False`.

GEOSGeometry.**overlaps**(*other*)
Returns true if the DE-9IM intersection matrix for the two geometries is `T*T***T**` (for two points or two surfaces) `1*T***T**` (for two curves).

GEOSGeometry.**relate_pattern**(*other*, *pattern*)
Returns `True` if the elements in the DE-9IM intersection matrix for this geometry and the other matches the given `pattern` – a string of nine characters from the alphabet: {`T`, `F`, `*`, `0`}.

GEOSGeometry.**touches**(*other*)
Returns `True` if the DE-9IM intersection matrix for the two geometries is `FT*******`, `F**T*****` or `F***T****`.

GEOSGeometry.**within**(*other*)
Returns `True` if the DE-9IM intersection matrix for the two geometries is `T*F**F***`.

Topological Methods

GEOSGeometry.**buffer**(*width*, *quadsegs=8*)
Returns a *GEOSGeometry* that represents all points whose distance from this geometry is less than or equal to the given `width`. The optional `quadsegs` keyword sets the number of segments used to approximate a quarter circle (defaults is 8).

GEOSGeometry.**difference**(*other*)
Returns a *GEOSGeometry* representing the points making up this geometry that do not make up other.

GEOSGeometry.**interpolate**(*distance*)

GEOSGeometry.**interpolate_normalized**(*distance*)
Given a distance (float), returns the point (or closest point) within the geometry (*LineString* or *MultiLineString*) at that distance. The normalized version takes the distance as a float between 0 (origin) and 1 (endpoint).

Reverse of *GEOSGeometry.project()*.

GEOSGeometry.**intersection**(*other*)
Returns a *GEOSGeometry* representing the points shared by this geometry and other.

GEOSGeometry.**project**(*point*)

GEOSGeometry.**project_normalized**(*point*)
Returns the distance (float) from the origin of the geometry (*LineString* or *MultiLineString*) to the point projected on the geometry (that is to a point of the line the closest to the given point). The normalized version returns the distance as a float between 0 (origin) and 1 (endpoint).

Reverse of *GEOSGeometry.interpolate()*.

GEOSGeometry.**relate**(*other*)
Returns the DE-9IM intersection matrix (a string) representing the topological relationship between this geometry and the other.

GEOSGeometry.**simplify**(*tolerance=0.0*, *preserve_topology=False*)
Returns a new *GEOSGeometry*, simplified to the specified tolerance using the Douglas-Peucker algorithm. A higher tolerance value implies fewer points in the output. If no tolerance is provided, it defaults to 0.

By default, this function does not preserve topology. For example, *Polygon* objects can be split, be collapsed into lines, or disappear. *Polygon* holes can be created or disappear, and lines may cross. By specifying `preserve_topology=True`, the result will have the same dimension and number of components as the input; this is significantly slower, however.

GEOSGeometry.**sym_difference**(*other*)
Returns a *GEOSGeometry* combining the points in this geometry not in other, and the points in other not in this geometry.

GEOSGeometry.**union**(*other*)
Returns a *GEOSGeometry* representing all the points in this geometry and the other.

Topological Properties

GEOSGeometry.**boundary**
Returns the boundary as a newly allocated Geometry object.

GEOSGeometry.**centroid**
Returns a *Point* object representing the geometric center of the geometry. The point is not guaranteed to be on the interior of the geometry.

GEOSGeometry.**convex_hull**
Returns the smallest *Polygon* that contains all the points in the geometry.

GEOSGeometry.**envelope**
Returns a *Polygon* that represents the bounding envelope of this geometry. Note that it can also return a *Point* if the input geometry is a point.

GEOSGeometry.**point_on_surface**
Computes and returns a *Point* guaranteed to be on the interior of this geometry.

GEOSGeometry.**unary_union**
Computes the union of all the elements of this geometry.

The result obeys the following contract:

- Unioning a set of *LineString*s has the effect of fully noding and dissolving the linework.
- Unioning a set of *Polygon*s will always return a *Polygon* or *MultiPolygon* geometry (unlike *GEOSGeometry.union()*, which may return geometries of lower dimension if a topology collapse occurs).

Other Properties & Methods

GEOSGeometry.**area**
This property returns the area of the Geometry.

GEOSGeometry.**extent**
This property returns the extent of this geometry as a 4-tuple, consisting of `(xmin, ymin, xmax, ymax)`.

GEOSGeometry.**clone**()
This method returns a *GEOSGeometry* that is a clone of the original.

GEOSGeometry.**distance**(*geom*)
Returns the distance between the closest points on this geometry and the given `geom` (another *GEOSGeometry* object).

: GEOS distance calculations are linear – in other words, GEOS does not perform a spherical calculation even if the SRID specifies a geographic coordinate system.

GEOSGeometry.**length**
Returns the length of this geometry (e.g., 0 for a *Point*, the length of a *LineString*, or the circumference of a *Polygon*).

GEOSGeometry.**prepared**
Returns a GEOS `PreparedGeometry` for the contents of this geometry. `PreparedGeometry` objects are optimized for the contains, intersects, covers, crosses, disjoint, overlaps, touches and within operations. Refer to the *Prepared Geometries* documentation for more information.

GEOSGeometry.**srs**
Returns a *SpatialReference* object corresponding to the SRID of the geometry or `None`.

: Requires GDAL.

GEOSGeometry.**transform**(*ct*, *clone=False*)
Transforms the geometry according to the given coordinate transformation parameter (`ct`), which may be an integer SRID, spatial reference WKT string, a PROJ.4 string, a *SpatialReference* object, or a *CoordTransform* object. By default, the geometry is transformed in-place and nothing is returned. However if the `clone` keyword is set, then the geometry is not modified and a transformed clone of the geometry is returned instead.

: Requires GDAL. Raises *GEOSException* if GDAL is not available or if the geometry's SRID is `None` or less than 0. It doesn't impose any constraints on the geometry's SRID if called with a *CoordTransform* object.

In previous versions, it required the geometry's SRID to be a positive integer even if it was called with a *CoordTransform* object.

Point

class Point(*x=None*, *y=None*, *z=None*, *srid=None*)
`Point` objects are instantiated using arguments that represent the component coordinates of the point or with a single sequence coordinates. For example, the following are equivalent:

```
>>> pnt = Point(5, 23)
>>> pnt = Point([5, 23])
```

Empty `Point` objects may be instantiated by passing no arguments or an empty sequence. The following are equivalent:

```
>>> pnt = Point()
>>> pnt = Point([])
```

In previous versions, an empty `Point` couldn't be instantiated.

LineString

class LineString(**args*, ***kwargs*)

`LineString` objects are instantiated using arguments that are either a sequence of coordinates or *Point* objects. For example, the following are equivalent:

```
>>> ls = LineString((0, 0), (1, 1))
>>> ls = LineString(Point(0, 0), Point(1, 1))
```

In addition, `LineString` objects may also be created by passing in a single sequence of coordinate or *Point* objects:

```
>>> ls = LineString( ((0, 0), (1, 1)) )
>>> ls = LineString( [Point(0, 0), Point(1, 1)] )
```

Empty `LineString` objects may be instantiated by passing no arguments or an empty sequence. The following are equivalent:

```
>>> ls = LineString()
>>> ls = LineString([])
```

In previous versions, an empty `LineString` couldn't be instantiated.

closed

Returns whether or not this `LineString` is closed.

LinearRing

class LinearRing(**args*, ***kwargs*)

`LinearRing` objects are constructed in the exact same way as *LineString* objects, however the coordinates must be *closed*, in other words, the first coordinates must be the same as the last coordinates. For example:

```
>>> ls = LinearRing((0, 0), (0, 1), (1, 1), (0, 0))
```

Notice that `(0, 0)` is the first and last coordinate – if they were not equal, an error would be raised.

Polygon

class Polygon(**args*, ***kwargs*)

`Polygon` objects may be instantiated by passing in parameters that represent the rings of the polygon. The parameters must either be *LinearRing* instances, or a sequence that may be used to construct a *LinearRing*:

```
>>> ext_coords = ((0, 0), (0, 1), (1, 1), (1, 0), (0, 0))
>>> int_coords = ((0.4, 0.4), (0.4, 0.6), (0.6, 0.6), (0.6, 0.4), (0.4, 0.4))
>>> poly = Polygon(ext_coords, int_coords)
>>> poly = Polygon(LinearRing(ext_coords), LinearRing(int_coords))
```

In previous versions, an empty `Polygon` couldn't be instantiated.

classmethod from_bbox(*bbox*)

Returns a polygon object from the given bounding-box, a 4-tuple comprising `(xmin, ymin, xmax, ymax)`.

num_interior_rings

Returns the number of interior rings in this geometry.

Comparing Polygons

Note that it is possible to compare `Polygon` objects directly with < or >, but as the comparison is made through Polygon's *LineString*, it does not mean much (but is consistent and quick). You can always force the comparison with the *area* property:

```
>>> if poly_1.area > poly_2.area:
>>>     pass
```

Geometry Collections

MultiPoint

class MultiPoint (**args*, ***kwargs*)

`MultiPoint` objects may be instantiated by passing in *`Point`* objects as arguments, or a single sequence of *`Point`* objects:

```
>>> mp = MultiPoint(Point(0, 0), Point(1, 1))
>>> mp = MultiPoint( (Point(0, 0), Point(1, 1)) )
```

In previous versions, an empty `MultiPoint` couldn't be instantiated.

MultiLineString

class MultiLineString (**args*, ***kwargs*)

`MultiLineString` objects may be instantiated by passing in *`LineString`* objects as arguments, or a single sequence of *`LineString`* objects:

```
>>> ls1 = LineString((0, 0), (1, 1))
>>> ls2 = LineString((2, 2), (3, 3))
>>> mls = MultiLineString(ls1, ls2)
>>> mls = MultiLineString([ls1, ls2])
```

In previous versions, an empty `MultiLineString` couldn't be instantiated.

merged

Returns a *`LineString`* representing the line merge of all the components in this `MultiLineString`.

closed

Returns `True` if and only if all elements are closed. Requires GEOS 3.5.

MultiPolygon

class MultiPolygon (**args*, ***kwargs*)

`MultiPolygon` objects may be instantiated by passing *`Polygon`* objects as arguments, or a single sequence of *`Polygon`* objects:

```
>>> p1 = Polygon( ((0, 0), (0, 1), (1, 1), (0, 0)) )
>>> p2 = Polygon( ((1, 1), (1, 2), (2, 2), (1, 1)) )
>>> mp = MultiPolygon(p1, p2)
>>> mp = MultiPolygon([p1, p2])
```

In previous versions, an empty `MultiPolygon` couldn't be instantiated.

cascaded_union

1.10 : Use the *`GEOSGeometry.unary_union`* property instead.

Returns a *`Polygon`* that is the union of all of the component polygons in this collection. The algorithm employed is significantly more efficient (faster) than trying to union the geometries together individually. [2]

[2] For more information, read Paul Ramsey's blog post about (Much) Faster Unions in PostGIS 1.4 and Martin Davis' blog post on Fast polygon merging in JTS using Cascaded Union.

GeometryCollection

class GeometryCollection (**args*, ***kwargs*)

`GeometryCollection` objects may be instantiated by passing in other *`GEOSGeometry`* as arguments, or a single sequence of *`GEOSGeometry`* objects:

```
>>> poly = Polygon( ((0, 0), (0, 1), (1, 1), (0, 0)) )
>>> gc = GeometryCollection(Point(0, 0), MultiPoint(Point(0, 0), Point(1, 1)), poly)
>>> gc = GeometryCollection((Point(0, 0), MultiPoint(Point(0, 0), Point(1, 1)), poly))
```

In previous versions, an empty `GeometryCollection` couldn't be instantiated.

Prepared Geometries

In order to obtain a prepared geometry, just access the *`GEOSGeometry.prepared`* property. Once you have a `PreparedGeometry` instance its spatial predicate methods, listed below, may be used with other `GEOSGeometry` objects. An operation with a prepared geometry can be orders of magnitude faster – the more complex the geometry that is prepared, the larger the speedup in the operation. For more information, please consult the GEOS wiki page on prepared geometries.

For example:

```
>>> from django.contrib.gis.geos import Point, Polygon
>>> poly = Polygon.from_bbox((0, 0, 5, 5))
>>> prep_poly = poly.prepared
>>> prep_poly.contains(Point(2.5, 2.5))
True
```

PreparedGeometry

class PreparedGeometry

All methods on `PreparedGeometry` take an `other` argument, which must be a *`GEOSGeometry`* instance.

contains (*other*)

contains_properly (*other*)

covers (*other*)

crosses (*other*)

disjoint (*other*)

intersects (*other*)

overlaps (*other*)

touches (*other*)

within (*other*)

Geometry Factories

fromfile (*file_h*)

file_h (a Python `file` object or a string path to the file) – input file that contains spatial data

a *`GEOSGeometry`* corresponding to the spatial data in the file

Example:

```
>>> from django.contrib.gis.geos import fromfile
>>> g = fromfile('/home/bob/geom.wkt')
```

fromstr(*string*, *srid=None*)

- **string** (*string*) – string that contains spatial data
- **srid** (*int*) – spatial reference identifier

a *GEOSGeometry* corresponding to the spatial data in the string

`fromstr(string, srid)` is equivalent to *GEOSGeometry(string, srid)*.

Example:

```
>>> from django.contrib.gis.geos import fromstr
>>> pnt = fromstr('POINT(-90.5 29.5)', srid=4326)
```

I/O Objects

Reader Objects The reader I/O classes simply return a *GEOSGeometry* instance from the WKB and/or WKT input given to their `read(geom)` method.

class WKBReader

Example:

```
>>> from django.contrib.gis.geos import WKBReader
>>> wkb_r = WKBReader()
>>> wkb_r.read('0101000000000000000000F03F000000000000F03F')
<Point object at 0x103a88910>
```

class WKTReader

Example:

```
>>> from django.contrib.gis.geos import WKTReader
>>> wkt_r = WKTReader()
>>> wkt_r.read('POINT(1 1)')
<Point object at 0x103a88b50>
```

Writer Objects All writer objects have a `write(geom)` method that returns either the WKB or WKT of the given geometry. In addition, *WKBWriter* objects also have properties that may be used to change the byte order, and or include the SRID value (in other words, EWKB).

class WKBWriter(*dim=2*)

`WKBWriter` provides the most control over its output. By default it returns OGC-compliant WKB when its `write` method is called. However, it has properties that allow for the creation of EWKB, a superset of the WKB standard that includes additional information. See the *WKBWriter.outdim* documentation for more details about the `dim` argument.

The ability to pass the `dim` argument to the constructor was added.

write(*geom*)

Returns the WKB of the given geometry as a Python `buffer` object. Example:

```
>>> from django.contrib.gis.geos import Point, WKBWriter
>>> pnt = Point(1, 1)
>>> wkb_w = WKBWriter()
>>> wkb_w.write(pnt)
<read-only buffer for 0x103a898f0, size -1, offset 0 at 0x103a89930>
```

write_hex(*geom*)

Returns WKB of the geometry in hexadecimal. Example:

```
>>> from django.contrib.gis.geos import Point, WKBWriter
>>> pnt = Point(1, 1)
>>> wkb_w = WKBWriter()
>>> wkb_w.write_hex(pnt)
'0101000000000000000000F03F000000000000F03F'
```

byteorder

This property may be set to change the byte-order of the geometry representation.

Byteorder Value	Description
0	Big Endian (e.g., compatible with RISC systems)
1	Little Endian (e.g., compatible with x86 systems)

Example:

```
>>> from django.contrib.gis.geos import Point, WKBWriter
>>> wkb_w = WKBWriter()
>>> pnt = Point(1, 1)
>>> wkb_w.write_hex(pnt)
'0101000000000000000000F03F000000000000F03F'
>>> wkb_w.byteorder = 0
'00000000013FF00000000000003FF0000000000000'
```

outdim

This property may be set to change the output dimension of the geometry representation. In other words, if you have a 3D geometry then set to 3 so that the Z value is included in the WKB.

Outdim Value	Description
2	The default, output 2D WKB.
3	Output 3D WKB.

Example:

```
>>> from django.contrib.gis.geos import Point, WKBWriter
>>> wkb_w = WKBWriter()
>>> wkb_w.outdim
2
>>> pnt = Point(1, 1, 1)
>>> wkb_w.write_hex(pnt) # By default, no Z value included:
'0101000000000000000000F03F000000000000F03F'
>>> wkb_w.outdim = 3 # Tell writer to include Z values
>>> wkb_w.write_hex(pnt)
'01010000800000000000000F03F000000000000F03F000000000000F03F'
```

srid

Set this property with a boolean to indicate whether the SRID of the geometry should be included with the WKB representation. Example:

```
>>> from django.contrib.gis.geos import Point, WKBWriter
>>> wkb_w = WKBWriter()
>>> pnt = Point(1, 1, srid=4326)
>>> wkb_w.write_hex(pnt) # By default, no SRID included:
'0101000000000000000000F03F000000000000F03F'
>>> wkb_w.srid = True # Tell writer to include SRID
>>> wkb_w.write_hex(pnt)
'0101000020E6100000000000000000F03F000000000000F03F'
```

class **WKTWriter**(*dim=2*, *trim=False*, *precision=None*)

This class allows outputting the WKT representation of a geometry. See the *WKBWriter.outdim*, *trim*, and *precision* attributes for details about the constructor arguments.

The ability to pass the `dim`, `trim`, and `precision` arguments to the constructor was added.

write(*geom*)

Returns the WKT of the given geometry. Example:

```
>>> from django.contrib.gis.geos import Point, WKTWriter
>>> pnt = Point(1, 1)
>>> wkt_w = WKTWriter()
>>> wkt_w.write(pnt)
'POINT (1.0000000000000000 1.0000000000000000)'
```

outdim

See *WKBWriter.outdim*.

trim

This property is used to enable or disable trimming of unnecessary decimals.

```
>>> from django.contrib.gis.geos import Point, WKTWriter
>>> pnt = Point(1, 1)
>>> wkt_w = WKTWriter()
>>> wkt_w.trim
False
>>> wkt_w.write(pnt)
'POINT (1.0000000000000000 1.0000000000000000)'
>>> wkt_w.trim = True
>>> wkt_w.write(pnt)
'POINT (1 1)'
```

precision

This property controls the rounding precision of coordinates; if set to `None` rounding is disabled.

```
>>> from django.contrib.gis.geos import Point, WKTWriter
>>> pnt = Point(1.44, 1.66)
>>> wkt_w = WKTWriter()
>>> print(wkt_w.precision)
None
>>> wkt_w.write(pnt)
'POINT (1.4399999999999999 1.6599999999999999)'
>>> wkt_w.precision = 0
>>> wkt_w.write(pnt)
'POINT (1 2)'
>>> wkt_w.precision = 1
>>> wkt_w.write(pnt)
'POINT (1.4 1.7)'
```

Settings

GEOS_LIBRARY_PATH A string specifying the location of the GEOS C library. Typically, this setting is only used if the GEOS C library is in a non-standard location (e.g., `/home/bob/lib/libgeos_c.so`).

: The setting must be the *full* path to the **C** shared library; in other words you want to use `libgeos_c.so`, not `libgeos.so`.

Exceptions

exception GEOSException
The base GEOS exception, indicates a GEOS-related error.

GDAL API

GDAL stands for **Geospatial Data Abstraction Library**, and is a veritable "Swiss army knife" of GIS data functionality. A subset of GDAL is the OGR Simple Features Library, which specializes in reading and writing vector geographic data in a variety of standard formats.

GeoDjango provides a high-level Python interface for some of the capabilities of OGR, including the reading and coordinate transformation of vector spatial data and minimal support for GDAL's features with respect to raster (image) data.

: Although the module is named `gdal`, GeoDjango only supports some of the capabilities of OGR and GDAL's raster features at this time.

Overview

Sample Data The GDAL/OGR tools described here are designed to help you read in your geospatial data, in order for most of them to be useful you have to have some data to work with. If you're starting out and don't yet have any data of your own to use, GeoDjango tests contain a number of simple data sets that you can use for testing. You can download them here:

```
$ wget https://raw.githubusercontent.com/django/django/master/tests/gis_tests/data/cities/cities.{sh
$ wget https://raw.githubusercontent.com/django/django/master/tests/gis_tests/data/rasters/raster.ti
```

Vector Data Source Objects

DataSource *DataSource* is a wrapper for the OGR data source object that supports reading data from a variety of OGR-supported geospatial file formats and data sources using a simple, consistent interface. Each data source is represented by a *DataSource* object which contains one or more layers of data. Each layer, represented by a *Layer* object, contains some number of geographic features (*Feature*), information about the type of features contained in that layer (e.g. points, polygons, etc.), as well as the names and types of any additional fields (*Field*) of data that may be associated with each feature in that layer.

class DataSource (*ds_input*, *encoding='utf-8'*)
The constructor for `DataSource` only requires one parameter: the path of the file you want to read. However, OGR also supports a variety of more complex data sources, including databases, that may be accessed by passing

a special name string instead of a path. For more information, see the OGR Vector Formats documentation. The *name* property of a DataSource instance gives the OGR name of the underlying data source that it is using.

The optional encoding parameter allows you to specify a non-standard encoding of the strings in the source. This is typically useful when you obtain DjangoUnicodeDecodeError exceptions while reading field values.

Once you've created your DataSource, you can find out how many layers of data it contains by accessing the *layer_count* property, or (equivalently) by using the len() function. For information on accessing the layers of data themselves, see the next section:

```
>>> from django.contrib.gis.gdal import DataSource
>>> ds = DataSource('/path/to/your/cities.shp')
>>> ds.name
'/path/to/your/cities.shp'
>>> ds.layer_count                  # This file only contains one layer
1
```

layer_count

Returns the number of layers in the data source.

name

Returns the name of the data source.

Layer

class Layer

Layer is a wrapper for a layer of data in a DataSource object. You never create a Layer object directly. Instead, you retrieve them from a *DataSource* object, which is essentially a standard Python container of Layer objects. For example, you can access a specific layer by its index (e.g. ds[0] to access the first layer), or you can iterate over all the layers in the container in a for loop. The Layer itself acts as a container for geometric features.

Typically, all the features in a given layer have the same geometry type. The *geom_type* property of a layer is an *OGRGeomType* that identifies the feature type. We can use it to print out some basic information about each layer in a *DataSource*:

```
>>> for layer in ds:
...     print('Layer "%s": %i %ss' % (layer.name, len(layer), layer.geom_type.name))
...
Layer "cities": 3 Points
```

The example output is from the cities data source, loaded above, which evidently contains one layer, called "cities", which contains three point features. For simplicity, the examples below assume that you've stored that layer in the variable layer:

```
>>> layer = ds[0]
```

name

Returns the name of this layer in the data source.

```
>>> layer.name
'cities'
```

num_feat

Returns the number of features in the layer. Same as len(layer):

```
>>> layer.num_feat
3
```

geom_type

Returns the geometry type of the layer, as an *OGRGeomType* object:

```
>>> layer.geom_type.name
'Point'
```

num_fields

Returns the number of fields in the layer, i.e the number of fields of data associated with each feature in the layer:

```
>>> layer.num_fields
4
```

fields

Returns a list of the names of each of the fields in this layer:

```
>>> layer.fields
['Name', 'Population', 'Density', 'Created']
```

Returns a list of the data types of each of the fields in this layer. These are subclasses of `Field`, discussed below:

```
>>> [ft.__name__ for ft in layer.field_types]
['OFTString', 'OFTReal', 'OFTReal', 'OFTDate']
```

field_widths

Returns a list of the maximum field widths for each of the fields in this layer:

```
>>> layer.field_widths
[80, 11, 24, 10]
```

field_precisions

Returns a list of the numeric precisions for each of the fields in this layer. This is meaningless (and set to zero) for non-numeric fields:

```
>>> layer.field_precisions
[0, 0, 15, 0]
```

extent

Returns the spatial extent of this layer, as an *Envelope* object:

```
>>> layer.extent.tuple
(-104.609252, 29.763374, -95.23506, 38.971823)
```

srs

Property that returns the *SpatialReference* associated with this layer:

```
>>> print(layer.srs)
GEOGCS["GCS_WGS_1984",
    DATUM["WGS_1984",
        SPHEROID["WGS_1984",6378137,298.257223563]],
    PRIMEM["Greenwich",0],
    UNIT["Degree",0.01745329251994295]]
```

If the *Layer* has no spatial reference information associated with it, None is returned.

spatial_filter

Property that may be used to retrieve or set a spatial filter for this layer. A spatial filter can only be set with an *OGRGeometry* instance, a 4-tuple extent, or None. When set with something other than None, only features that intersect the filter will be returned when iterating over the layer:

```
>>> print(layer.spatial_filter)
None
>>> print(len(layer))
3
>>> [feat.get('Name') for feat in layer]
['Pueblo', 'Lawrence', 'Houston']
>>> ks_extent = (-102.051, 36.99, -94.59, 40.00) # Extent for state of Kansas
>>> layer.spatial_filter = ks_extent
>>> len(layer)
1
>>> [feat.get('Name') for feat in layer]
['Lawrence']
>>> layer.spatial_filter = None
>>> len(layer)
3
```

get_fields()

A method that returns a list of the values of a given field for each feature in the layer:

```
>>> layer.get_fields('Name')
['Pueblo', 'Lawrence', 'Houston']
```

get_geoms(*geos=False*)

A method that returns a list containing the geometry of each feature in the layer. If the optional argument geos is set to True then the geometries are converted to *GEOSGeometry* objects. Otherwise, they are returned as *OGRGeometry* objects:

```
>>> [pt.tuple for pt in layer.get_geoms()]
[(-104.609252, 38.255001), (-95.23506, 38.971823), (-95.363151, 29.763374)]
```

test_capability(*capability*)

Returns a boolean indicating whether this layer supports the given capability (a string). Examples of valid capability strings include: 'RandomRead', 'SequentialWrite', 'RandomWrite', 'FastSpatialFilter', 'FastFeatureCount', 'FastGetExtent', 'CreateField', 'Transactions','DeleteFeature', and 'FastSetNextByIndex'.

Feature

class Feature

Feature wraps an OGR feature. You never create a Feature object directly. Instead, you retrieve them from a *Layer* object. Each feature consists of a geometry and a set of fields containing additional properties. The geometry of a field is accessible via its geom property, which returns an *OGRGeometry* object. A Feature behaves like a standard Python container for its fields, which it returns as *Field* objects: you can access a field directly by its index or name, or you can iterate over a feature's fields, e.g. in a for loop.

geom

Returns the geometry for this feature, as an OGRGeometry object:

```
>>> city.geom.tuple
(-104.609252, 38.255001)
```

get

A method that returns the value of the given field (specified by name) for this feature, **not** a `Field` wrapper object:

```
>>> city.get('Population')
102121
```

geom_type

Returns the type of geometry for this feature, as an *OGRGeomType* object. This will be the same for all features in a given layer and is equivalent to the *Layer.geom_type* property of the *Layer* object the feature came from.

num_fields

Returns the number of fields of data associated with the feature. This will be the same for all features in a given layer and is equivalent to the *Layer.num_fields* property of the *Layer* object the feature came from.

fields

Returns a list of the names of the fields of data associated with the feature. This will be the same for all features in a given layer and is equivalent to the *Layer.fields* property of the *Layer* object the feature came from.

fid

Returns the feature identifier within the layer:

```
>>> city.fid
0
```

layer_name

Returns the name of the *Layer* that the feature came from. This will be the same for all features in a given layer:

```
>>> city.layer_name
'cities'
```

index

A method that returns the index of the given field name. This will be the same for all features in a given layer:

```
>>> city.index('Population')
1
```

Field

class Field

name

Returns the name of this field:

```
>>> city['Name'].name
'Name'
```

type

Returns the OGR type of this field, as an integer. The `FIELD_CLASSES` dictionary maps these values onto subclasses of `Field`:

```
>>> city['Density'].type
2
```

type_name

Returns a string with the name of the data type of this field:

```
>>> city['Name'].type_name
'String'
```

value

Returns the value of this field. The `Field` class itself returns the value as a string, but each subclass returns the value in the most appropriate form:

```
>>> city['Population'].value
102121
```

width

Returns the width of this field:

```
>>> city['Name'].width
80
```

precision

Returns the numeric precision of this field. This is meaningless (and set to zero) for non-numeric fields:

```
>>> city['Density'].precision
15
```

as_double()

Returns the value of the field as a double (float):

```
>>> city['Density'].as_double()
874.7
```

as_int()

Returns the value of the field as an integer:

```
>>> city['Population'].as_int()
102121
```

as_string()

Returns the value of the field as a string:

```
>>> city['Name'].as_string()
'Pueblo'
```

as_datetime()

Returns the value of the field as a tuple of date and time components:

```
>>> city['Created'].as_datetime()
(c_long(1999), c_long(5), c_long(23), c_long(0), c_long(0), c_long(0), c_long(0))
```

Driver

class Driver(*dr_input*)

The `Driver` class is used internally to wrap an OGR *DataSource* driver.

driver_count

Returns the number of OGR vector drivers currently registered.

OGR Geometries

OGRGeometry *OGRGeometry* objects share similar functionality with *GEOSGeometry* objects and are thin wrappers around OGR's internal geometry representation. Thus, they allow for more efficient access to data when using *DataSource*. Unlike its GEOS counterpart, *OGRGeometry* supports spatial reference systems and coordinate transformation:

```
>>> from django.contrib.gis.gdal import OGRGeometry
>>> polygon = OGRGeometry('POLYGON((0 0, 5 0, 5 5, 0 5))')
```

class OGRGeometry(*geom_input*, *srs=None*)

This object is a wrapper for the OGR Geometry class. These objects are instantiated directly from the given `geom_input` parameter, which may be a string containing WKT, HEX, GeoJSON, a `buffer` containing WKB data, or an *OGRGeomType* object. These objects are also returned from the *Feature.geom* attribute, when reading vector data from *Layer* (which is in turn a part of a *DataSource*).

classmethod from_bbox(*bbox*)

Constructs a *Polygon* from the given bounding-box (a 4-tuple).

__len__()

Returns the number of points in a *LineString*, the number of rings in a *Polygon*, or the number of geometries in a *GeometryCollection*. Not applicable to other geometry types.

__iter__()

Iterates over the points in a *LineString*, the rings in a *Polygon*, or the geometries in a *GeometryCollection*. Not applicable to other geometry types.

__getitem__()

Returns the point at the specified index for a *LineString*, the interior ring at the specified index for a *Polygon*, or the geometry at the specified index in a *GeometryCollection*. Not applicable to other geometry types.

dimension

Returns the number of coordinated dimensions of the geometry, i.e. 0 for points, 1 for lines, and so forth:

```
>> polygon.dimension
2
```

coord_dim

Returns or sets the coordinate dimension of this geometry. For example, the value would be 2 for two-dimensional geometries.

geom_count

Returns the number of elements in this geometry:

```
>>> polygon.geom_count
1
```

point_count

Returns the number of points used to describe this geometry:

```
>>> polygon.point_count
4
```

num_points

Alias for *point_count*.

num_coords

Alias for *point_count*.

geom_type

Returns the type of this geometry, as an *OGRGeomType* object.

geom_name

Returns the name of the type of this geometry:

```
>>> polygon.geom_name
'POLYGON'
```

area

Returns the area of this geometry, or 0 for geometries that do not contain an area:

```
>>> polygon.area
25.0
```

envelope

Returns the envelope of this geometry, as an *Envelope* object.

extent

Returns the envelope of this geometry as a 4-tuple, instead of as an *Envelope* object:

```
>>> point.extent
(0.0, 0.0, 5.0, 5.0)
```

srs

This property controls the spatial reference for this geometry, or `None` if no spatial reference system has been assigned to it. If assigned, accessing this property returns a *SpatialReference* object. It may be set with another *SpatialReference* object, or any input that *SpatialReference* accepts. Example:

```
>>> city.geom.srs.name
'GCS_WGS_1984'
```

srid

Returns or sets the spatial reference identifier corresponding to *SpatialReference* of this geometry. Returns `None` if there is no spatial reference information associated with this geometry, or if an SRID cannot be determined.

geos

Returns a *GEOSGeometry* object corresponding to this geometry.

gml

Returns a string representation of this geometry in GML format:

```
>>> OGRGeometry('POINT(1 2)').gml
'<gml:Point><gml:coordinates>1,2</gml:coordinates></gml:Point>'
```

hex

Returns a string representation of this geometry in HEX WKB format:

```
>>> OGRGeometry('POINT(1 2)').hex
'0101000000000000000000F03F0000000000000040'
```

json

Returns a string representation of this geometry in JSON format:

```
>>> OGRGeometry('POINT(1 2)').json
'{ "type": "Point", "coordinates": [ 1.000000, 2.000000 ] }'
```

kml

Returns a string representation of this geometry in KML format.

wkb_size

Returns the size of the WKB buffer needed to hold a WKB representation of this geometry:

```
>>> OGRGeometry('POINT(1 2)').wkb_size
21
```

wkb

Returns a `buffer` containing a WKB representation of this geometry.

wkt

Returns a string representation of this geometry in WKT format.

ewkt

Returns the EWKT representation of this geometry.

clone()

Returns a new *OGRGeometry* clone of this geometry object.

close_rings()

If there are any rings within this geometry that have not been closed, this routine will do so by adding the starting point to the end:

```
>>> triangle = OGRGeometry('LINEARRING (0 0,0 1,1 0)')
>>> triangle.close_rings()
>>> triangle.wkt
'LINEARRING (0 0,0 1,1 0,0 0)'
```

transform(*coord_trans*, *clone=False*)

Transforms this geometry to a different spatial reference system. May take a *CoordTransform* object, a *SpatialReference* object, or any other input accepted by *SpatialReference* (including spatial reference WKT and PROJ.4 strings, or an integer SRID).

By default nothing is returned and the geometry is transformed in-place. However, if the `clone` keyword is set to `True` then a transformed clone of this geometry is returned instead.

intersects(*other*)

Returns `True` if this geometry intersects the other, otherwise returns `False`.

equals(*other*)

Returns `True` if this geometry is equivalent to the other, otherwise returns `False`.

disjoint(*other*)

Returns `True` if this geometry is spatially disjoint to (i.e. does not intersect) the other, otherwise returns `False`.

touches(*other*)

Returns `True` if this geometry touches the other, otherwise returns `False`.

crosses(*other*)

Returns `True` if this geometry crosses the other, otherwise returns `False`.

within(*other*)

Returns `True` if this geometry is contained within the other, otherwise returns `False`.

contains(*other*)

Returns `True` if this geometry contains the other, otherwise returns `False`.

overlaps(*other*)

Returns `True` if this geometry overlaps the other, otherwise returns `False`.

boundary()

The boundary of this geometry, as a new *OGRGeometry* object.

convex_hull

The smallest convex polygon that contains this geometry, as a new *OGRGeometry* object.

difference()

Returns the region consisting of the difference of this geometry and the other, as a new *OGRGeometry* object.

intersection()

Returns the region consisting of the intersection of this geometry and the other, as a new *OGRGeometry* object.

sym_difference()

Returns the region consisting of the symmetric difference of this geometry and the other, as a new *OGRGeometry* object.

union()

Returns the region consisting of the union of this geometry and the other, as a new *OGRGeometry* object.

tuple

Returns the coordinates of a point geometry as a tuple, the coordinates of a line geometry as a tuple of tuples, and so forth:

```
>>> OGRGeometry('POINT (1 2)').tuple
(1.0, 2.0)
>>> OGRGeometry('LINESTRING (1 2,3 4)').tuple
((1.0, 2.0), (3.0, 4.0))
```

coords

An alias for *tuple*.

class Point

x

Returns the X coordinate of this point:

```
>>> OGRGeometry('POINT (1 2)').x
1.0
```

y

Returns the Y coordinate of this point:

```
>>> OGRGeometry('POINT (1 2)').y
2.0
```

z

Returns the Z coordinate of this point, or `None` if the point does not have a Z coordinate:

```
>>> OGRGeometry('POINT (1 2 3)').z
3.0
```

class LineString

x

Returns a list of X coordinates in this line:

```
>>> OGRGeometry('LINESTRING (1 2,3 4)').x
[1.0, 3.0]
```

y

Returns a list of Y coordinates in this line:

```
>>> OGRGeometry('LINESTRING (1 2,3 4)').y
[2.0, 4.0]
```

z

Returns a list of Z coordinates in this line, or `None` if the line does not have Z coordinates:

```
>>> OGRGeometry('LINESTRING (1 2 3,4 5 6)').z
[3.0, 6.0]
```

class Polygon

shell

Returns the shell or exterior ring of this polygon, as a `LinearRing` geometry.

exterior_ring

An alias for *shell*.

centroid

Returns a *Point* representing the centroid of this polygon.

class GeometryCollection

add(*geom*)

Adds a geometry to this geometry collection. Not applicable to other geometry types.

OGRGeomType

class OGRGeomType(*type_input*)

This class allows for the representation of an OGR geometry type in any of several ways:

```
>>> from django.contrib.gis.gdal import OGRGeomType
>>> gt1 = OGRGeomType(3)             # Using an integer for the type
>>> gt2 = OGRGeomType('Polygon')     # Using a string
>>> gt3 = OGRGeomType('POLYGON')     # It's case-insensitive
>>> print(gt1 == 3, gt1 == 'Polygon') # Equivalence works w/non-OGRGeomType objects
True True
```

name

Returns a short-hand string form of the OGR Geometry type:

```
>>> gt1.name
'Polygon'
```

num

Returns the number corresponding to the OGR geometry type:

```
>>> gt1.num
3
```

django

Returns the Django field type (a subclass of GeometryField) to use for storing this OGR type, or `None` if there is no appropriate Django type:

```
>>> gt1.django
'PolygonField'
```

Envelope

class Envelope(**args*)

Represents an OGR Envelope structure that contains the minimum and maximum X, Y coordinates for a rectangle bounding box. The naming of the variables is compatible with the OGR Envelope C structure.

min_x

The value of the minimum X coordinate.

min_y

The value of the maximum X coordinate.

max_x

The value of the minimum Y coordinate.

max_y

The value of the maximum Y coordinate.

ur

The upper-right coordinate, as a tuple.

ll

The lower-left coordinate, as a tuple.

tuple

A tuple representing the envelope.

wkt

A string representing this envelope as a polygon in WKT format.

expand_to_include (**args*)

Coordinate System Objects

SpatialReference

class SpatialReference (*srs_input*)

Spatial reference objects are initialized on the given `srs_input`, which may be one of the following:

- OGC Well Known Text (WKT) (a string)
- EPSG code (integer or string)
- PROJ.4 string
- A shorthand string for well-known standards (`'WGS84'`, `'WGS72'`, `'NAD27'`, `'NAD83'`)

Example:

```
>>> wgs84 = SpatialReference('WGS84') # shorthand string
>>> wgs84 = SpatialReference(4326) # EPSG code
>>> wgs84 = SpatialReference('EPSG:4326') # EPSG string
>>> proj4 = '+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs '
>>> wgs84 = SpatialReference(proj4) # PROJ.4 string
>>> wgs84 = SpatialReference("""GEOGCS["WGS 84",
DATUM["WGS_1984",
     SPHEROID["WGS 84",6378137,298.257223563,
         AUTHORITY["EPSG","7030"]],
     AUTHORITY["EPSG","6326"]],
 PRIMEM["Greenwich",0,
     AUTHORITY["EPSG","8901"]],
 UNIT["degree",0.01745329251994328,
     AUTHORITY["EPSG","9122"]],
 AUTHORITY["EPSG","4326"]]""") # OGC WKT
```

__getitem__ (*target*)

Returns the value of the given string attribute node, `None` if the node doesn't exist. Can also take a tuple as a parameter, (target, child), where child is the index of the attribute in the WKT. For example:

```
>>> wkt = 'GEOGCS["WGS 84", DATUM["WGS_1984, ... AUTHORITY["EPSG","4326"]]')
>>> srs = SpatialReference(wkt) # could also use 'WGS84', or 4326
>>> print(srs['GEOGCS'])
WGS 84
>>> print(srs['DATUM'])
WGS_1984
>>> print(srs['AUTHORITY'])
EPSG
>>> print(srs['AUTHORITY', 1]) # The authority value
4326
>>> print(srs['TOWGS84', 4]) # the fourth value in this wkt
0
>>> print(srs['UNIT|AUTHORITY']) # For the units authority, have to use the pipe symbol.
```

```
    EPSG
    >>> print(srs['UNIT|AUTHORITY', 1]) # The authority value for the units
    9122
```

attr_value(*target*, *index=0*)

The attribute value for the given target node (e.g. `'PROJCS'`). The index keyword specifies an index of the child node to return.

auth_name(*target*)

Returns the authority name for the given string target node.

auth_code(*target*)

Returns the authority code for the given string target node.

clone()

Returns a clone of this spatial reference object.

identify_epsg()

This method inspects the WKT of this `SpatialReference` and will add EPSG authority nodes where an EPSG identifier is applicable.

from_esri()

Morphs this SpatialReference from ESRI's format to EPSG

to_esri()

Morphs this SpatialReference to ESRI's format.

validate()

Checks to see if the given spatial reference is valid, if not an exception will be raised.

import_epsg(*epsg*)

Import spatial reference from EPSG code.

import_proj(*proj*)

Import spatial reference from PROJ.4 string.

import_user_input(*user_input*)

import_wkt(*wkt*)

Import spatial reference from WKT.

import_xml(*xml*)

Import spatial reference from XML.

name

Returns the name of this Spatial Reference.

srid

Returns the SRID of top-level authority, or `None` if undefined.

linear_name

Returns the name of the linear units.

linear_units

Returns the value of the linear units.

angular_name

Returns the name of the angular units."

angular_units

Returns the value of the angular units.

units

Returns a 2-tuple of the units value and the units name and will automatically determines whether to return the linear or angular units.

ellipsoid

Returns a tuple of the ellipsoid parameters for this spatial reference: (semimajor axis, semiminor axis, and inverse flattening).

semi_major

Returns the semi major axis of the ellipsoid for this spatial reference.

semi_minor

Returns the semi minor axis of the ellipsoid for this spatial reference.

inverse_flattening

Returns the inverse flattening of the ellipsoid for this spatial reference.

geographic

Returns `True` if this spatial reference is geographic (root node is `GEOGCS`).

local

Returns `True` if this spatial reference is local (root node is `LOCAL_CS`).

projected

Returns `True` if this spatial reference is a projected coordinate system (root node is `PROJCS`).

wkt

Returns the WKT representation of this spatial reference.

pretty_wkt

Returns the 'pretty' representation of the WKT.

proj

Returns the PROJ.4 representation for this spatial reference.

proj4

Alias for *SpatialReference.proj*.

xml

Returns the XML representation of this spatial reference.

CoordTransform

class CoordTransform(*source*, *target*)

Represents a coordinate system transform. It is initialized with two *SpatialReference*, representing the source and target coordinate systems, respectively. These objects should be used when performing the same coordinate transformation repeatedly on different geometries:

```
>>> ct = CoordTransform(SpatialReference('WGS84'), SpatialReference('NAD83'))
>>> for feat in layer:
...     geom = feat.geom # getting clone of feature geometry
...     geom.transform(ct) # transforming
```

Raster Data Objects

GDALRaster *GDALRaster* is a wrapper for the GDAL raster source object that supports reading data from a variety of GDAL-supported geospatial file formats and data sources using a simple, consistent interface. Each data source is represented by a *GDALRaster* object which contains one or more layers of data named bands. Each band, represented by a *GDALBand* object, contains georeferenced image data. For example, an RGB image is represented as three bands: one for red, one for green, and one for blue.

: For raster data there is no difference between a raster instance and its data source. Unlike for the Geometry objects, *GDALRaster* objects are always a data source. Temporary rasters can be instantiated in memory using the corresponding driver, but they will be of the same class as file-based raster sources.

---

class GDALRaster(*ds_input*, *write=False*)

The constructor for `GDALRaster` accepts two parameters. The first parameter defines the raster source, it is either a path to a file or spatial data with values defining the properties of a new raster (such as size and name). If the input is a file path, the second parameter specifies if the raster should be opened with write access. If the input is raw data, the parameters `width`, `height`, and `srid` are required. The following example shows how rasters can be created from different input sources (using the sample data from the GeoDjango tests, see also the *Sample Data* section):

```
>>> from django.contrib.gis.gdal import GDALRaster
>>> rst = GDALRaster('/path/to/your/raster.tif', write=False)
>>> rst.name
'/path/to/your/raster.tif'
>>> rst.width, rst.height           # This file has 163 x 174 pixels
(163, 174)
>>> rst = GDALRaster({'srid': 4326, 'width': 1, 'height': 2, 'datatype': 1
...                   'bands': [{'data': [0, 1]}]}) # Creates in-memory raster
>>> rst.srs.srid
4326
>>> rst.width, rst.height
(1, 2)
>>> rst.bands[0].data()
array([[0, 1]], dtype=int8)
```

`GDALRaster` objects can now be instantiated directly from raw data. Setters have been added for the following properties: `srs`, `geotransform`, `origin`, `scale`, and `skew`.

name

The name of the source which is equivalent to the input file path or the name provided upon instantiation.

```
>>> GDALRaster({'width': 10, 'height': 10, 'name': 'myraster', 'srid': 4326}).name
'myraster'
```

driver

The name of the GDAL driver used to handle the input file. For `GDALRaster`s created from a file, the driver type is detected automatically. The creation of rasters from scratch is a in-memory raster by default (`'MEM'`), but can be altered as needed. For instance, use `GTiff` for a `GeoTiff` file. For a list of file types, see also the GDAL Raster Formats list.

An in-memory raster is created through the following example:

```
>>> GDALRaster({'width': 10, 'height': 10, 'srid': 4326}).driver.name
'MEM'
```

A file based GeoTiff raster is created through the following example:

```
>>> import tempfile
>>> rstfile = tempfile.NamedTemporaryFile(suffix='.tif')
>>> rst = GDALRaster({'driver': 'GTiff', 'name': rstfile.name, 'srid': 4326,
...                   'width': 255, 'height': 255, 'nr_of_bands': 1})
>>> rst.name
'/tmp/tmp7x9H4J.tif'           # The exact filename will be different on your computer
>>> rst.driver.name
'GTiff'
```

width

The width of the source in pixels (X-axis).

```
>>> GDALRaster({'width': 10, 'height': 20, 'srid': 4326}).width
10
```

height

The height of the source in pixels (Y-axis).

```
>>> GDALRaster({'width': 10, 'height': 20, 'srid': 4326}).height
20
```

srs

The spatial reference system of the raster, as a *SpatialReference* instance. The SRS can be changed by setting it to an other *SpatialReference* or providing any input that is accepted by the *SpatialReference* constructor.

```
>>> rst = GDALRaster({'width': 10, 'height': 20, 'srid': 4326})
>>> rst.srs.srid
4326
>>> rst.srs = 3086
>>> rst.srs.srid
3086
```

srid

The Spatial Reference System Identifier (SRID) of the raster. This property is a shortcut to getting or setting the SRID through the *srs* attribute.

```
>>> rst = GDALRaster({'width': 10, 'height': 20, 'srid': 4326})
>>> rst.srid
4326
>>> rst.srid = 3086
>>> rst.srid
3086
>>> rst.srs.srid  # This is equivalent
3086
```

geotransform

The affine transformation matrix used to georeference the source, as a tuple of six coefficients which map

pixel/line coordinates into georeferenced space using the following relationship:

```
Xgeo = GT(0) + Xpixel*GT(1) + Yline*GT(2)
Ygeo = GT(3) + Xpixel*GT(4) + Yline*GT(5)
```

The same values can be retrieved by accessing the *origin* (indices 0 and 3), *scale* (indices 1 and 5) and *skew* (indices 2 and 4) properties.

The default is `[0.0, 1.0, 0.0, 0.0, 0.0, -1.0]`.

```
>>> rst = GDALRaster({'width': 10, 'height': 20, 'srid': 4326})
>>> rst.geotransform
[0.0, 1.0, 0.0, 0.0, 0.0, -1.0]
```

origin

Coordinates of the top left origin of the raster in the spatial reference system of the source, as a point object with `x` and `y` members.

```
>>> rst = GDALRaster({'width': 10, 'height': 20, 'srid': 4326})
>>> rst.origin
[0.0, 0.0]
>>> rst.origin.x = 1
>>> rst.origin
[1.0, 0.0]
```

scale

Pixel width and height used for georeferencing the raster, as a as a point object with `x` and `y` members. See *geotransform* for more information.

```
>>> rst = GDALRaster({'width': 10, 'height': 20, 'srid': 4326})
>>> rst.scale
[1.0, -1.0]
>>> rst.scale.x = 2
>>> rst.scale
[2.0, -1.0]
```

skew

Skew coefficients used to georeference the raster, as a point object with `x` and `y` members. In case of north up images, these coefficients are both `0`.

```
>>> rst = GDALRaster({'width': 10, 'height': 20, 'srid': 4326})
>>> rst.skew
[0.0, 0.0]
>>> rst.skew.x = 3
>>> rst.skew
[3.0, 0.0]
```

extent

Extent (boundary values) of the raster source, as a 4-tuple `(xmin, ymin, xmax, ymax)` in the spatial reference system of the source.

```
>>> rst = GDALRaster({'width': 10, 'height': 20, 'srid': 4326})
>>> rst.extent
(0.0, -20.0, 10.0, 0.0)
>>> rst.origin.x = 100
>>> rst.extent
(100.0, -20.0, 110.0, 0.0)
```

bands

List of all bands of the source, as *GDALBand* instances.

```
>>> rst = GDALRaster({"width": 1, "height": 2, 'srid': 4326,
...                   "bands": [{"data": [0, 1]}, {"data": [2, 3]}]})
>>> len(rst.bands)
2
>>> rst.bands[1].data()
array([[ 2.,  3.]], dtype=float32)
```

warp(*ds_input*, *resampling='NearestNeighbour'*, *max_error=0.0*)

Returns a warped version of this raster.

The warping parameters can be specified through the `ds_input` argument. The use of `ds_input` is analogous to the corresponding argument of the class constructor. It is a dictionary with the characteristics of the target raster. Allowed dictionary key values are width, height, SRID, origin, scale, skew, datatype, driver, and name (filename).

By default, the warp functions keeps most parameters equal to the values of the original source raster, so only parameters that should be changed need to be specified. Note that this includes the driver, so for file-based rasters the warp function will create a new raster on disk.

The only parameter that is set differently from the source raster is the name. The default value of the the raster name is the name of the source raster appended with `'_copy' + source_driver_name`. For file-based rasters it is recommended to provide the file path of the target raster.

The resampling algorithm used for warping can be specified with the `resampling` argument. The default is `NearestNeighbor`, and the other allowed values are `Bilinear`, `Cubic`, `CubicSpline`, `Lanczos`, `Average`, and `Mode`.

The `max_error` argument can be used to specify the maximum error measured in input pixels that is allowed in approximating the transformation. The default is 0.0 for exact calculations.

For users familiar with `GDAL`, this function has a similar functionality to the `gdalwarp` command-line utility.

For example, the warp function can be used for aggregating a raster to the double of its original pixel scale:

```
>>> rst = GDALRaster({
...     "width": 6, "height": 6, "srid": 3086,
...     "origin": [500000, 400000],
...     "scale": [100, -100],
...     "bands": [{"data": range(36), "nodata_value": 99}]
... })
>>> target = rst.warp({"scale": [200, -200], "width": 3, "height": 3})
>>> target.bands[0].data()
array([[  7.,   9.,  11.],
       [ 19.,  21.,  23.],
       [ 31.,  33.,  35.]], dtype=float32)
```

transform(*srid*, *driver=None*, *name=None*, *resampling='NearestNeighbour'*, *max_error=0.0*)

Returns a transformed version of this raster with the specified SRID.

This function transforms the current raster into a new spatial reference system that can be specified with an `srid`. It calculates the bounds and scale of the current raster in the new spatial reference system and warps the raster using the *warp* function.

By default, the driver of the source raster is used and the name of the raster is the original name appended with `'_copy' + source_driver_name`. A different driver or name can be specified with the `driver` and `name` arguments.

The default resampling algorithm is `NearestNeighbour` but can be changed using the `resampling` argument. The default maximum allowed error for resampling is 0.0 and can be changed using the `max_error` argument. Consult the *warp* documentation for detail on those arguments.

```
>>> rst = GDALRaster({
...     "width": 6, "height": 6, "srid": 3086,
...     "origin": [500000, 400000],
...     "scale": [100, -100],
...     "bands": [{"data": range(36), "nodata_value": 99}]
... })
>>> target = rst.transform(4326)
>>> target.origin
[-82.98492744885776, 27.601924753080144]
```

GDALBand

class GDALBand

`GDALBand` instances are not created explicitly, but rather obtained from a *`GDALRaster`* object, through its *`bands`* attribute. The GDALBands contain the actual pixel values of the raster.

`description`
The name or description of the band, if any.

`width`
The width of the band in pixels (X-axis).

`height`
The height of the band in pixels (Y-axis).

`pixel_count`
The total number of pixels in this band. Is equal to `width * height`.

`statistics`(*refresh=False*, *approximate=False*)
Compute statistics on the pixel values of this band. The return value is a tuple with the following structure: `(minimum, maximum, mean, standard deviation)`.

If the `approximate` argument is set to `True`, the statistics may be computed based on overviews or a subset of image tiles.

If the `refresh` argument is set to `True`, the statistics will be computed from the data directly, and the cache will be updated with the result.

If a persistent cache value is found, that value is returned. For raster formats using Persistent Auxiliary Metadata (PAM) services, the statistics might be cached in an auxiliary file. In some cases this metadata might be out of sync with the pixel values or cause values from a previous call to be returned which don't reflect the value of the `approximate` argument. In such cases, use the `refresh` argument to get updated values and store them in the cache.

For empty bands (where all pixel values are "no data"), all statistics are returned as `None`.

The statistics can also be retrieved directly by accessing the *`min`*, *`max`*, *`mean`*, and *`std`* properties.

`min`
The minimum pixel value of the band (excluding the "no data" value).

`max`
The maximum pixel value of the band (excluding the "no data" value).

`mean`
The mean of all pixel values of the band (excluding the "no data" value).

`std`
The standard deviation of all pixel values of the band (excluding the "no data" value).

nodata_value

The "no data" value for a band is generally a special marker value used to mark pixels that are not valid data. Such pixels should generally not be displayed, nor contribute to analysis operations.

To delete an existing "no data" value, set this property to `None` (requires GDAL $\geq$ 2.1).

This property can now be set as well.

The "no data" value can now be deleted by setting the `nodata_value` attribute to `None`.

datatype (*as_string=False*)

The data type contained in the band, as an integer constant between 0 (Unknown) and 11. If `as_string` is `True`, the data type is returned as a string with the following possible values: `GDT_Unknown`, `GDT_Byte`, `GDT_UInt16`, `GDT_Int16`, `GDT_UInt32`, `GDT_Int32`, `GDT_Float32`, `GDT_Float64`, `GDT_CInt16`, `GDT_CInt32`, `GDT_CFloat32`, and `GDT_CFloat64`.

data (*data=None*, *offset=None*, *size=None*, *shape=None*)

The accessor to the pixel values of the `GDALBand`. Returns the complete data array if no parameters are provided. A subset of the pixel array can be requested by specifying an offset and block size as tuples.

If NumPy is available, the data is returned as NumPy array. For performance reasons, it is highly recommended to use NumPy.

Data is written to the `GDALBand` if the `data` parameter is provided. The input can be of one of the following types - packed string, buffer, list, array, and NumPy array. The number of items in the input should normally correspond to the total number of pixels in the band, or to the number of pixels for a specific block of pixel values if the `offset` and `size` parameters are provided.

If the number of items in the input is different from the target pixel block, the `shape` parameter must be specified. The shape is a tuple that specifies the width and height of the input data in pixels. The data is then replicated to update the pixel values of the selected block. This is useful to fill an entire band with a single value, for instance.

For example:

```
>>> rst = GDALRaster({'width': 4, 'height': 4, 'srid': 4326, 'datatype': 1, 'nr_of_bands': 1
>>> bnd = rst.bands[0]
>>> bnd.data(range(16))
>>> bnd.data()
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]], dtype=int8)
>>> bnd.data(offset=(1, 1), size=(2, 2))
array([[ 5,  6],
       [ 9, 10]], dtype=int8)
>>> bnd.data(data=[-1, -2, -3, -4], offset=(1, 1), size=(2, 2))
>>> bnd.data()
array([[ 0,  1,  2,  3],
       [ 4, -1, -2,  7],
       [ 8, -3, -4, 11],
       [12, 13, 14, 15]], dtype=int8)
>>> bnd.data(data='\x9d\xa8\xb3\xbe', offset=(1, 1), size=(2, 2))
>>> bnd.data()
array([[  0,   1,   2,   3],
       [  4, -99, -88,   7],
       [  8, -77, -66,  11],
       [ 12,  13,  14,  15]], dtype=int8)
>>> bnd.data([1], shape=(1, 1))
>>> bnd.data()
```

```
        array([[1, 1, 1, 1],
               [1, 1, 1, 1],
               [1, 1, 1, 1],
               [1, 1, 1, 1]], dtype=uint8)
        >>> bnd.data(range(4), shape=(1, 4))
        array([[0, 0, 0, 0],
               [1, 1, 1, 1],
               [2, 2, 2, 2],
               [3, 3, 3, 3]], dtype=uint8)
```

The `shape` parameter and the ability to replicate data input when setting `GDALBand` data was added.

Settings

`GDAL_LIBRARY_PATH` A string specifying the location of the GDAL library. Typically, this setting is only used if the GDAL library is in a non-standard location (e.g., `/home/john/lib/libgdal.so`).

Geolocation with GeoIP

1.9 : This module is deprecated in favor of django.contrib.gis.geoip2, which supports IPv6 and the GeoLite2 database format.

The *`GeoIP`* object is a ctypes wrapper for the MaxMind GeoIP C API. [1]

In order to perform IP-based geolocation, the *`GeoIP`* object requires the GeoIP C library and either the GeoIP Country or City datasets in binary format (the CSV files will not work!). These datasets may be downloaded from MaxMind. Grab the `GeoLiteCountry/GeoIP.dat.gz` and `GeoLiteCity.dat.gz` files and unzip them in a directory corresponding to what you set *`GEOIP_PATH`* with in your settings. See the example and reference below for more details.

Example

Assuming you have the GeoIP C library installed, here is an example of its usage:

```
>>> from django.contrib.gis.geoip import GeoIP
>>> g = GeoIP()
>>> g.country('google.com')
{'country_code': 'US', 'country_name': 'United States'}
>>> g.city('72.14.207.99')
{'area_code': 650,
'city': 'Mountain View',
'country_code': 'US',
'country_code3': 'USA',
'country_name': 'United States',
'dma_code': 807,
'latitude': 37.419200897216797,
'longitude': -122.05740356445312,
'postal_code': '94043',
'region': 'CA'}
>>> g.lat_lon('salon.com')
(37.789798736572266, -122.39420318603516)
>>> g.lon_lat('uh.edu')
(-95.415199279785156, 29.77549934387207)
```

[1] GeoIP(R) is a registered trademark of MaxMind, LLC of Boston, Massachusetts.

```
>>> g.geos('24.124.1.80').wkt
'POINT (-95.2087020874023438 39.0392990112304688)'
```

GeoIP Settings

GEOIP_PATH A string specifying the directory where the GeoIP data files are located. This setting is *required* unless manually specified with `path` keyword when initializing the *GeoIP* object.

GEOIP_LIBRARY_PATH A string specifying the location of the GeoIP C library. Typically, this setting is only used if the GeoIP C library is in a non-standard location (e.g., `/home/sue/lib/libGeoIP.so`).

GEOIP_COUNTRY The basename to use for the GeoIP country data file. Defaults to `'GeoIP.dat'`.

GEOIP_CITY The basename to use for the GeoIP city data file. Defaults to `'GeoLiteCity.dat'`.

GeoIP API

class GeoIP(*path=None*, *cache=0*, *country=None*, *city=None*)

The `GeoIP` object does not require any parameters to use the default settings. However, at the very least the *GEOIP_PATH* setting should be set with the path of the location of your GeoIP data sets. The following initialization keywords may be used to customize any of the defaults.

Keyword Arguments	Description
`path`	Base directory to where GeoIP data is located or the full path to where the city or country data files (.dat) are located. Assumes that both the city and country data sets are located in this directory; overrides the *GEOIP_PATH* settings attribute.
`cache`	The cache settings when opening up the GeoIP datasets, and may be an integer in (0, 1, 2, 4) corresponding to the `GEOIP_STANDARD`, `GEOIP_MEMORY_CACHE`, `GEOIP_CHECK_CACHE`, and `GEOIP_INDEX_CACHE` `GeoIPOptions` C API settings, respectively. Defaults to 0 (`GEOIP_STANDARD`).
`country`	The name of the GeoIP country data file. Defaults to `GeoIP.dat`. Setting this keyword overrides the *GEOIP_COUNTRY* settings attribute.
`city`	The name of the GeoIP city data file. Defaults to `GeoLiteCity.dat`. Setting this keyword overrides the *GEOIP_CITY* settings attribute.

GeoIP Methods

Querying All the following querying routines may take either a string IP address or a fully qualified domain name (FQDN). For example, both `'205.186.163.125'` and `'djangoproject.com'` would be valid query parameters.

`GeoIP.`**city**(*query*)

Returns a dictionary of city information for the given query. Some of the values in the dictionary may be undefined (`None`).

`GeoIP.`**country**(*query*)

Returns a dictionary with the country code and country for the given query.

GeoIP.**country_code**(*query*)

Returns only the country code corresponding to the query.

GeoIP.**country_name**(*query*)

Returns only the country name corresponding to the query.

Coordinate Retrieval
GeoIP.**coords**(*query*)
Returns a coordinate tuple of (longitude, latitude).

GeoIP.**lon_lat**(*query*)

Returns a coordinate tuple of (longitude, latitude).

GeoIP.**lat_lon**(*query*)

Returns a coordinate tuple of (latitude, longitude),

GeoIP.**geos**(*query*)

Returns a *django.contrib.gis.geos.Point* object corresponding to the query.

Database Information
GeoIP.**country_info**
This property returns information about the GeoIP country database.

GeoIP.**city_info**

This property returns information about the GeoIP city database.

GeoIP.**info**

This property returns information about all GeoIP databases (both city and country), and the version of the GeoIP C library (if supported).

GeoIP-Python API compatibility methods These methods exist to ease compatibility with any code using MaxMind's existing Python API.

classmethod GeoIP.**open**(*path*, *cache*)

This classmethod instantiates the GeoIP object from the given database path and given cache setting.

GeoIP.**region_by_addr**(*query*)

GeoIP.**region_by_name**(*query*)

GeoIP.**record_by_addr**(*query*)

GeoIP.**record_by_name**(*query*)

GeoIP.**country_code_by_addr**(*query*)

GeoIP.**country_code_by_name**(*query*)

GeoIP.**country_name_by_addr**(*query*)

GeoIP.**country_name_by_name**(*query*)

Geolocation with GeoIP2

The *GeoIP2* object is a wrapper for the MaxMind geoip2 Python library. [1]

In order to perform IP-based geolocation, the *GeoIP2* object requires the geoip2 Python library and the GeoIP *Country* and/or *City* datasets in binary format (the CSV files will not work!). Grab the `GeoLite2-Country.mmdb.gz` and `GeoLite2-City.mmdb.gz` files and unzip them in a directory corresponding to the *GEOIP_PATH* setting.

Additionally, it is recommended to install the libmaxminddb C library, so that `geoip2` can leverage the C library's faster speed.

Example

Here is an example of its usage:

```
>>> from django.contrib.gis.geoip2 import GeoIP2
>>> g = GeoIP2()
>>> g.country('google.com')
{'country_code': 'US', 'country_name': 'United States'}
>>> g.city('72.14.207.99')
{'city': 'Mountain View',
'country_code': 'US',
'country_name': 'United States',
'dma_code': 807,
'latitude': 37.419200897216797,
'longitude': -122.05740356445312,
'postal_code': '94043',
'region': 'CA'}
>>> g.lat_lon('salon.com')
(39.0437, -77.4875)
>>> g.lon_lat('uh.edu')
(-95.4342, 29.834)
>>> g.geos('24.124.1.80').wkt
'POINT (-97 38)'
```

`GeoIP` Settings

`GEOIP_PATH` A string specifying the directory where the GeoIP data files are located. This setting is *required* unless manually specified with `path` keyword when initializing the *GeoIP2* object.

`GEOIP_COUNTRY` The basename to use for the GeoIP country data file. Defaults to `'GeoLite2-Country.mmdb'`.

`GEOIP_CITY` The basename to use for the GeoIP city data file. Defaults to `'GeoLite2-City.mmdb'`.

`GeoIP` API

class `GeoIP2` (*path=None*, *cache=0*, *country=None*, *city=None*)

[1] GeoIP(R) is a registered trademark of MaxMind, Inc.

The `GeoIP` object does not require any parameters to use the default settings. However, at the very least the *`GEOIP_PATH`* setting should be set with the path of the location of your GeoIP datasets. The following initialization keywords may be used to customize any of the defaults.

Keyword Arguments	Description
`path`	Base directory to where GeoIP data is located or the full path to where the city or country data files (`.mmdb`) are located. Assumes that both the city and country datasets are located in this directory; overrides the *`GEOIP_PATH`* setting.
`cache`	The cache settings when opening up the GeoIP datasets. May be an integer in (0, 1, 2, 4, 8) corresponding to the `MODE_AUTO`, `MODE_MMAP_EXT`, `MODE_MMAP`, and `GEOIP_INDEX_CACHE` `MODE_MEMORY` C API settings, respectively. Defaults to 0 (`MODE_AUTO`).
`country`	The name of the GeoIP country data file. Defaults to `GeoLite2-Country.mmdb`. Setting this keyword overrides the *`GEOIP_COUNTRY`* setting.
`city`	The name of the GeoIP city data file. Defaults to `GeoLite2-City.mmdb`. Setting this keyword overrides the *`GEOIP_CITY`* setting.

`GeoIP` Methods

Instantiating
classmethod `GeoIP2.`**`open`**(*path*, *cache*)
This classmethod instantiates the GeoIP object from the given database path and given cache setting.

Querying All the following querying routines may take either a string IP address or a fully qualified domain name (FQDN). For example, both `'205.186.163.125'` and `'djangoproject.com'` would be valid query parameters.

`GeoIP2.`**`city`**(*query*)

Returns a dictionary of city information for the given query. Some of the values in the dictionary may be undefined (`None`).

`GeoIP2.`**`country`**(*query*)

Returns a dictionary with the country code and country for the given query.

`GeoIP2.`**`country_code`**(*query*)

Returns the country code corresponding to the query.

`GeoIP2.`**`country_name`**(*query*)

Returns the country name corresponding to the query.

Coordinate Retrieval
`GeoIP2.`**`coords`**(*query*)
Returns a coordinate tuple of (longitude, latitude).

`GeoIP2.`**`lon_lat`**(*query*)

Returns a coordinate tuple of (longitude, latitude).

`GeoIP2.`**`lat_lon`**(*query*)

Returns a coordinate tuple of (latitude, longitude),

`GeoIP2.`**`geos`**(*query*)

Returns a *Point* object corresponding to the query.

GeoDjango Utilities

The *django.contrib.gis.utils* module contains various utilities that are useful in creating geospatial Web applications.

`LayerMapping` data import utility

The *LayerMapping* class provides a way to map the contents of vector spatial data files (e.g. shapefiles) into GeoDjango models.

This utility grew out of the author's personal needs to eliminate the code repetition that went into pulling geometries and fields out of a vector layer, converting to another coordinate system (e.g. WGS84), and then inserting into a GeoDjango model.

: Use of *LayerMapping* requires GDAL.

: GIS data sources, like shapefiles, may be very large. If you find that *LayerMapping* is using too much memory, set *DEBUG* to `False` in your settings. When *DEBUG* is set to `True`, Django *automatically logs every* SQL query – thus, when SQL statements contain geometries, it is easy to consume more memory than is typical.

Example

1. You need a GDAL-supported data source, like a shapefile (here we're using a simple polygon shapefile, `test_poly.shp`, with three features):

```
>>> from django.contrib.gis.gdal import DataSource
>>> ds = DataSource('test_poly.shp')
>>> layer = ds[0]
>>> print(layer.fields) # Exploring the fields in the layer, we only want the 'str' field.
['float', 'int', 'str']
>>> print(len(layer)) # getting the number of features in the layer (should be 3)
3
>>> print(layer.geom_type) # Should be 'Polygon'
Polygon
>>> print(layer.srs) # WGS84 in WKT
GEOGCS["GCS_WGS_1984",
    DATUM["WGS_1984",
        SPHEROID["WGS_1984",6378137,298.257223563]],
    PRIMEM["Greenwich",0],
    UNIT["Degree",0.01745329251994295]]
```

2. Now we define our corresponding Django model (make sure to use *migrate*):

```
from django.contrib.gis.db import models

class TestGeo(models.Model):
    name = models.CharField(max_length=25) # corresponds to the 'str' field
    poly = models.PolygonField(srid=4269) # we want our model in a different SRID

    def __str__(self):              # __unicode__ on Python 2
        return 'Name: %s' % self.name
```

3. Use *LayerMapping* to extract all the features and place them in the database:

```
>>> from django.contrib.gis.utils import LayerMapping
>>> from geoapp.models import TestGeo
>>> mapping = {'name' : 'str', # The 'name' model field maps to the 'str' layer field.
               'poly' : 'POLYGON', # For geometry fields use OGC name.
               } # The mapping is a dictionary
>>> lm = LayerMapping(TestGeo, 'test_poly.shp', mapping)
>>> lm.save(verbose=True) # Save the layermap, imports the data.
Saved: Name: 1
Saved: Name: 2
Saved: Name: 3
```

Here, *LayerMapping* just transformed the three geometries from the shapefile in their original spatial reference system (WGS84) to the spatial reference system of the GeoDjango model (NAD83). If no spatial reference system is defined for the layer, use the `source_srs` keyword with a *SpatialReference* object to specify one.

`LayerMapping` API

class `LayerMapping`(*model*, *data_source*, *mapping*, *layer=0*, *source_srs=None*, *encoding=None*, *transaction_mode='commit_on_success'*, *transform=True*, *unique=True*, *using='default'*)

The following are the arguments and keywords that may be used during instantiation of `LayerMapping` objects.

Argument	Description
`model`	The geographic model, *not* an instance.
`data_source`	The path to the OGR-supported data source file (e.g., a shapefile). Also accepts *django.contrib.gis.gdal.DataSource* instances.
`mapping`	A dictionary: keys are strings corresponding to the model field, and values correspond to string field names for the OGR feature, or if the model field is a geographic then it should correspond to the OGR geometry type, e.g., `'POINT'`, `'LINESTRING'`, `'POLYGON'`.

Keyword Arguments	
`layer`	The index of the layer to use from the Data Source (defaults to 0)
`source_srs`	Use this to specify the source SRS manually (for example, some shapefiles don't come with a `'.prj'` file). An integer SRID, WKT or PROJ.4 strings, and *django.contrib.gis.gdal.SpatialReference* objects are accepted.
`encoding`	Specifies the character set encoding of the strings in the OGR data source. For example, `'latin-1'`, `'utf-8'`, and `'cp437'` are all valid encoding parameters.
`transaction_mode`	May be `'commit_on_success'` (default) or `'autocommit'`.
`transform`	Setting this to False will disable coordinate transformations. In other words, geometries will be inserted into the database unmodified from their original state in the data source.
`unique`	Setting this to the name, or a tuple of names, from the given model will create models unique only to the given name(s). Geometries from each feature will be added into the collection associated with the unique model. Forces the transaction mode to be `'autocommit'`.
`using`	Sets the database to use when importing spatial data. Default is `'default'`.

`save()` Keyword Arguments

`LayerMapping.`**`save`**(*verbose=False*, *fid_range=False*, *step=False*, *progress=False*, *silent=False*, *stream=sys.stdout*, *strict=False*)

The `save()` method also accepts keywords. These keywords are used for controlling output logging, error handling, and for importing specific feature ranges.

Save Keyword Arguments	Description
`fid_range`	May be set with a slice or tuple of (begin, end) feature ID's to map from the data source. In other words, this keyword enables the user to selectively import a subset range of features in the geographic data source.
`progress`	When this keyword is set, status information will be printed giving the number of features processed and successfully saved. By default, progress information will be printed every 1000 features processed, however, this default may be overridden by setting this keyword with an integer for the desired interval.
`silent`	By default, non-fatal error notifications are printed to `sys.stdout`, but this keyword may be set to disable these notifications.
`step`	If set with an integer, transactions will occur at every step interval. For example, if `step=1000`, a commit would occur after the 1,000th feature, the 2,000th feature etc.
`stream`	Status information will be written to this file handle. Defaults to using `sys.stdout`, but any object with a `write` method is supported.
`strict`	Execution of the model mapping will cease upon the first error encountered. The default value (`False`) behavior is to attempt to continue.
`verbose`	If set, information will be printed subsequent to each model save executed on the database.

Troubleshooting

Running out of memory As noted in the warning at the top of this section, Django stores all SQL queries when `DEBUG=True`. Set `DEBUG=False` in your settings, and this should stop excessive memory use when running `LayerMapping` scripts.

MySQL: `max_allowed_packet` error If you encounter the following error when using `LayerMapping` and MySQL:

```
OperationalError: (1153, "Got a packet bigger than 'max_allowed_packet' bytes")
```

Then the solution is to increase the value of the `max_allowed_packet` setting in your MySQL configuration. For example, the default value may be something low like one megabyte – the setting may be modified in MySQL's configuration file (`my.cnf`) in the `[mysqld]` section:

```
max_allowed_packet = 10M
```

OGR Inspection

ogrinspect
ogrinspect(*data_source*, *model_name*, ***kwargs*)

mapping
mapping(*data_source*, *geom_name='geom'*, *layer_key=0*, *multi_geom=False*)

GeoJSON Serializer

GeoDjango provides a specific serializer for the GeoJSON format. See Serializing Django objects for more information on serialization.

The GDAL library is required if any of the serialized geometries need coordinate transformations (that is if the geometry's spatial reference system differs from the `srid` serializer option).

The GeoJSON serializer no longer needs GDAL if all geometries are in the same coordinate system as the `srid` serializer option.

The `geojson` serializer is not meant for round-tripping data, as it has no deserializer equivalent. For example, you cannot use *`loaddata`* to reload the output produced by this serializer. If you plan to reload the outputted data, use the plain *json serializer* instead.

In addition to the options of the `json` serializer, the `geojson` serializer accepts the following additional option when it is called by `serializers.serialize()`:

- `geometry_field`: A string containing the name of a geometry field to use for the `geometry` key of the GeoJSON feature. This is only needed when you have a model with more than one geometry field and you don't want to use the first defined geometry field (by default, the first geometry field is picked).
- `srid`: The SRID to use for the `geometry` content. Defaults to 4326 (WGS 84).

The *fields* option can be used to limit fields that will be present in the `properties` key, as it works with all other serializers.

Example:

```
from django.core.serializers import serialize
from my_app.models import City

serialize('geojson', City.objects.all(),
          geometry_field='point',
          fields=('name',))
```

Would output:

```
{
  'type': 'FeatureCollection',
  'crs': {
    'type': 'name',
    'properties': {'name': 'EPSG:4326'}
  },
  'features': [
    {
      'type': 'Feature',
      'geometry': {
        'type': 'Point',
        'coordinates': [-87.650175, 41.850385]
      },
      'properties': {
        'name': 'Chicago'
      }
    }
  ]
}
```

When the `fields` parameter is not specified, the `geojson` serializer adds a `pk` key to the `properties` dictionary with the primary key of the object as the value.

The `pk` key was added to the `properties` dictionary.

GeoDjango Management Commands

inspectdb

django-admin inspectdb

When *django.contrib.gis* is in your *INSTALLED_APPS*, the *inspectdb* management command is overridden with one from GeoDjango. The overridden command is spatially-aware, and places geometry fields in the auto-generated model definition, where appropriate.

ogrinspect

django-admin ogrinspect data_source model_name

The `ogrinspect` management command will inspect the given OGR-compatible *DataSource* (e.g., a shapefile) and will output a GeoDjango model with the given model name. There's a detailed example of using `ogrinspect` *in the tutorial*.

--blank BLANK
: Use a comma separated list of OGR field names to add the `blank=True` keyword option to the field definition. Set with `true` to apply to all applicable fields.

--decimal DECIMAL
: Use a comma separated list of OGR float fields to generate *DecimalField* instead of the default *FloatField*. Set to `true` to apply to all OGR float fields.

--geom-name GEOM_NAME
: Specifies the model attribute name to use for the geometry field. Defaults to `'geom'`.

--layer LAYER_KEY
: The key for specifying which layer in the OGR *DataSource* source to use. Defaults to 0 (the first layer). May be an integer or a string identifier for the *Layer*. When inspecting databases, `layer` is generally the table name you want to inspect.

--mapping
: Automatically generate a mapping dictionary for use with *LayerMapping*.

--multi-geom
: When generating the geometry field, treat it as a geometry collection. For example, if this setting is enabled then a *MultiPolygonField* will be placed in the generated model rather than *PolygonField*.

--name-field NAME_FIELD
: Generates a `__str__` routine (`__unicode__` on Python 2) on the model that will return the given field name.

--no-imports
: Suppresses the `from django.contrib.gis.db import models` import statement.

--null NULL
: Use a comma separated list of OGR field names to add the `null=True` keyword option to the field definition. Set with `true` to apply to all applicable fields.

--srid SRID
: The SRID to use for the geometry field. If not set, `ogrinspect` attempts to automatically determine of the SRID of the data source.

GeoDjango's admin site

GeoModelAdmin

class GeoModelAdmin

default_lon

The default center longitude.

default_lat

The default center latitude.

default_zoom

The default zoom level to use. Defaults to 18.

extra_js

Sequence of URLs to any extra JavaScript to include.

map_template

Override the template used to generate the JavaScript slippy map. Default is `'gis/admin/openlayers.html'`.

map_width

Width of the map, in pixels. Defaults to 600.

map_height

Height of the map, in pixels. Defaults to 400.

openlayers_url

Link to the URL of the OpenLayers JavaScript. Defaults to `'http://openlayers.org/api/2.13.1/OpenLayers.js`

modifiable

Defaults to `True`. When set to `False`, disables editing of existing geometry fields in the admin.

: This is different from adding the geometry field to *`readonly_fields`*, which will only display the WKT of the geometry. Setting `modifiable=False`, actually displays the geometry in a map, but disables the ability to edit its vertices.

OSMGeoAdmin

class OSMGeoAdmin

A subclass of *`GeoModelAdmin`* that uses a spherical mercator projection with OpenStreetMap street data tiles. See the *OSMGeoAdmin introduction* in the tutorial for a usage example.

Geographic Feeds

GeoDjango has its own *`Feed`* subclass that may embed location information in RSS/Atom feeds formatted according to either the Simple GeoRSS or W3C Geo standards. Because GeoDjango's syndication API is a superset of Django's, please consult Django's syndication documentation for details on general usage.

Example

API Reference

Feed Subclass

class Feed

In addition to methods provided by the *django.contrib.syndication.views.Feed* base class, GeoDjango's `Feed` class provides the following overrides. Note that these overrides may be done in multiple ways:

```
from django.contrib.gis.feeds import Feed

class MyFeed(Feed):

    # First, as a class attribute.
    geometry = ...
    item_geometry = ...

    # Also a function with no arguments
    def geometry(self):
        ...

    def item_geometry(self):
        ...

    # And as a function with a single argument
    def geometry(self, obj):
        ...

    def item_geometry(self, item):
        ...
```

geometry(*obj*)

Takes the object returned by `get_object()` and returns the *feed's* geometry. Typically this is a `GEOSGeometry` instance, or can be a tuple to represent a point or a box. For example:

```
class ZipcodeFeed(Feed):

    def geometry(self, obj):
        # Can also return: `obj.poly`, and `obj.poly.centroid`.
        return obj.poly.extent # tuple like: (X0, Y0, X1, Y1).
```

item_geometry(*item*)

Set this to return the geometry for each *item* in the feed. This can be a `GEOSGeometry` instance, or a tuple that represents a point coordinate or bounding box. For example:

```
class ZipcodeFeed(Feed):

    def item_geometry(self, obj):
        # Returns the polygon.
        return obj.poly
```

SyndicationFeed Subclasses The following *django.utils.feedgenerator.SyndicationFeed* subclasses are available:

class GeoRSSFeed

class GeoAtom1Feed

class W3CGeoFeed

: W3C Geo formatted feeds only support `PointField` geometries.

Geographic Sitemaps

KML is an XML language focused on geographic visualization [1]. `KMLSitemap` and its compressed counterpart `KMZSitemap` allow you to present geolocated data in a machine-readable format.

Example

Reference

`KMLSitemap`

`KMZSitemap`

Testing GeoDjango apps

Included in this documentation are some additional notes and settings for *PostGIS* users.

PostGIS

Settings

: The settings below have sensible defaults, and shouldn't require manual setting.

`POSTGIS_VERSION` When GeoDjango's spatial backend initializes on PostGIS, it has to perform an SQL query to determine the version in order to figure out what features are available. Advanced users wishing to prevent this additional query may set the version manually using a 3-tuple of integers specifying the major, minor, and micro version numbers for PostGIS. For example, to configure for PostGIS X.Y.Z you would use:

```
POSTGIS_VERSION = (X, Y, Z)
```

Obtaining sufficient privileges Depending on your configuration, this section describes several methods to configure a database user with sufficient privileges to run tests for GeoDjango applications on PostgreSQL. If your *spatial database template* was created like in the instructions, then your testing database user only needs to have the ability to create databases. In other configurations, you may be required to use a database superuser.

Create database user To make a database user with the ability to create databases, use the following command:

```
$ createuser --createdb -R -S <user_name>
```

The `-R -S` flags indicate that we do not want the user to have the ability to create additional users (roles) or to be a superuser, respectively.

Alternatively, you may alter an existing user's role from the SQL shell (assuming this is done from an existing superuser account):

[1] http://www.opengeospatial.org/standards/kml

```
postgres# ALTER ROLE <user_name> CREATEDB NOSUPERUSER NOCREATEROLE;
```

Create database superuser This may be done at the time the user is created, for example:

```
$ createuser --superuser <user_name>
```

Or you may alter the user's role from the SQL shell (assuming this is done from an existing superuser account):

```
postgres# ALTER ROLE <user_name> SUPERUSER;
```

Windows On Windows platforms the pgAdmin III utility may also be used as a simple way to add superuser privileges to your database user.

By default, the PostGIS installer on Windows includes a template spatial database entitled `template_postgis`.

GeoDjango tests

To have the GeoDjango tests executed when *running the Django test suite* with `runtests.py` all of the databases in the settings file must be using one of the *spatial database backends*.

Example The following is an example bare-bones settings file with spatial backends that can be used to run the entire Django test suite, including those in *`django.contrib.gis`*:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.contrib.gis.db.backends.postgis',
        'NAME': 'geodjango',
        'USER': 'geodjango',
    },
    'other': {
        'ENGINE': 'django.contrib.gis.db.backends.postgis',
        'NAME': 'other',
        'USER': 'geodjango',
    },
}

SECRET_KEY = 'django_tests_secret_key'
```

Assuming the settings above were in a `postgis.py` file in the same directory as `runtests.py`, then all Django and GeoDjango tests would be performed when executing the command:

```
$ ./runtests.py --settings=postgis
```

To run only the GeoDjango test suite, specify `gis_tests`:

```
$ ./runtests.py --settings=postgis gis_tests
```

Deploying GeoDjango

Basically, the deployment of a GeoDjango application is not different from the deployment of a normal Django application. Please consult Django's deployment documentation.

: GeoDjango uses the GDAL geospatial library which is not thread safe at this time. Thus, it is *highly* recommended to not use threading when deploying – in other words, use an appropriate configuration of Apache.
For example, when configuring your application with `mod_wsgi`, set the `WSGIDaemonProcess` attribute `threads` to `1`, unless Apache may crash when running your GeoDjango application. Increase the number of `processes` instead.

6.5.6 `django.contrib.humanize`

A set of Django template filters useful for adding a "human touch" to data.

To activate these filters, add `'django.contrib.humanize'` to your *INSTALLED_APPS* setting. Once you've done that, use `{% load humanize %}` in a template, and you'll have access to the following filters.

apnumber

For numbers 1-9, returns the number spelled out. Otherwise, returns the number. This follows Associated Press style.

Examples:

- `1` becomes `one`.
- `2` becomes `two`.
- `10` becomes `10`.

You can pass in either an integer or a string representation of an integer.

intcomma

Converts an integer to a string containing commas every three digits.

Examples:

- `4500` becomes `4,500`.
- `45000` becomes `45,000`.
- `450000` becomes `450,000`.
- `4500000` becomes `4,500,000`.

Format localization will be respected if enabled, e.g. with the `'de'` language:

- `45000` becomes `'45.000'`.
- `450000` becomes `'450.000'`.

You can pass in either an integer or a string representation of an integer.

intword

Converts a large integer to a friendly text representation. Works best for numbers over 1 million.

Examples:

- `1000000` becomes `1.0 million`.
- `1200000` becomes `1.2 million`.

- `1200000000` becomes `1.2 billion`.

Values up to 10^100 (Googol) are supported.

Format localization will be respected if enabled, e.g. with the `'de'` language:

- `1000000` becomes `'1,0 Million'`.
- `1200000` becomes `'1,2 Million'`.
- `1200000000` becomes `'1,2 Milliarden'`.

You can pass in either an integer or a string representation of an integer.

naturalday

For dates that are the current day or within one day, return "today", "tomorrow" or "yesterday", as appropriate. Otherwise, format the date using the passed in format string.

Argument: Date formatting string as described in the *date* tag.

Examples (when 'today' is 17 Feb 2007):

- `16 Feb 2007` becomes `yesterday`.
- `17 Feb 2007` becomes `today`.
- `18 Feb 2007` becomes `tomorrow`.
- Any other day is formatted according to given argument or the *DATE_FORMAT* setting if no argument is given.

naturaltime

For datetime values, returns a string representing how many seconds, minutes or hours ago it was – falling back to the *timesince* format if the value is more than a day old. In case the datetime value is in the future the return value will automatically use an appropriate phrase.

Examples (when 'now' is 17 Feb 2007 16:30:00):

- `17 Feb 2007 16:30:00` becomes `now`.
- `17 Feb 2007 16:29:31` becomes `29 seconds ago`.
- `17 Feb 2007 16:29:00` becomes `a minute ago`.
- `17 Feb 2007 16:25:35` becomes `4 minutes ago`.
- `17 Feb 2007 15:30:29` becomes `59 minutes ago`.
- `17 Feb 2007 15:30:01` becomes `59 minutes ago`.
- `17 Feb 2007 15:30:00` becomes `an hour ago`.
- `17 Feb 2007 13:31:29` becomes `2 hours ago`.
- `16 Feb 2007 13:31:29` becomes `1 day, 2 hours ago`.
- `16 Feb 2007 13:30:01` becomes `1 day, 2 hours ago`.
- `16 Feb 2007 13:30:00` becomes `1 day, 3 hours ago`.
- `17 Feb 2007 16:30:30` becomes `30 seconds from now`.
- `17 Feb 2007 16:30:29` becomes `29 seconds from now`.
- `17 Feb 2007 16:31:00` becomes `a minute from now`.

- `17 Feb 2007 16:34:35` becomes `4 minutes from now.`
- `17 Feb 2007 17:30:29` becomes `an hour from now.`
- `17 Feb 2007 18:31:29` becomes `2 hours from now.`
- `18 Feb 2007 16:31:29` becomes `1 day from now.`
- `26 Feb 2007 18:31:29` becomes `1 week, 2 days from now.`

`ordinal`

Converts an integer to its ordinal as a string.

Examples:

- `1` becomes `1st`.
- `2` becomes `2nd`.
- `3` becomes `3rd`.

You can pass in either an integer or a string representation of an integer.

6.5.7 The messages framework

Quite commonly in web applications, you need to display a one-time notification message (also known as "flash message") to the user after processing a form or some other types of user input.

For this, Django provides full support for cookie- and session-based messaging, for both anonymous and authenticated users. The messages framework allows you to temporarily store messages in one request and retrieve them for display in a subsequent request (usually the next one). Every message is tagged with a specific `level` that determines its priority (e.g., `info`, `warning`, or `error`).

Enabling messages

Messages are implemented through a middleware class and corresponding context processor.

The default `settings.py` created by `django-admin startproject` already contains all the settings required to enable message functionality:

- `'django.contrib.messages'` is in *INSTALLED_APPS*.
- *MIDDLEWARE* contains `'django.contrib.sessions.middleware.SessionMiddleware'` and `'django.contrib.messages.middleware.MessageMiddleware'`.

  The default *storage backend* relies on sessions. That's why `SessionMiddleware` must be enabled and appear before `MessageMiddleware` in *MIDDLEWARE*.
- The `'context_processors'` option of the `DjangoTemplates` backend defined in your *TEMPLATES* setting contains `'django.contrib.messages.context_processors.messages'`.

If you don't want to use messages, you can remove `'django.contrib.messages'` from your *INSTALLED_APPS*, the `MessageMiddleware` line from *MIDDLEWARE*, and the `messages` context processor from *TEMPLATES*.

Configuring the message engine

Storage backends

The messages framework can use different backends to store temporary messages.

Django provides three built-in storage classes in *django.contrib.messages*:

class storage.session.**SessionStorage**
This class stores all messages inside of the request's session. Therefore it requires Django's contrib.sessions application.

class storage.cookie.**CookieStorage**
This class stores the message data in a cookie (signed with a secret hash to prevent manipulation) to persist notifications across requests. Old messages are dropped if the cookie data size would exceed 2048 bytes.

class storage.fallback.**FallbackStorage**
This class first uses CookieStorage, and falls back to using SessionStorage for the messages that could not fit in a single cookie. It also requires Django's contrib.sessions application.

This behavior avoids writing to the session whenever possible. It should provide the best performance in the general case.

FallbackStorage is the default storage class. If it isn't suitable to your needs, you can select another storage class by setting *MESSAGE_STORAGE* to its full import path, for example:

```
MESSAGE_STORAGE = 'django.contrib.messages.storage.cookie.CookieStorage'
```

class storage.base.**BaseStorage**

To write your own storage class, subclass the BaseStorage class in django.contrib.messages.storage.base and implement the _get and _store methods.

Message levels

The messages framework is based on a configurable level architecture similar to that of the Python logging module. Message levels allow you to group messages by type so they can be filtered or displayed differently in views and templates.

The built-in levels, which can be imported from django.contrib.messages directly, are:

Constant	Purpose
DEBUG	Development-related messages that will be ignored (or removed) in a production deployment
INFO	Informational messages for the user
SUCCESS	An action was successful, e.g. "Your profile was updated successfully"
WARNING	A failure did not occur but may be imminent
ERROR	An action was **not** successful or some other failure occurred

The *MESSAGE_LEVEL* setting can be used to change the minimum recorded level (or it can be *changed per request*). Attempts to add messages of a level less than this will be ignored.

Message tags

Message tags are a string representation of the message level plus any extra tags that were added directly in the view (see *Adding extra message tags* below for more details). Tags are stored in a string and are separated by spaces. Typically, message tags are used as CSS classes to customize message style based on message type. By default, each level has a single tag that's a lowercase version of its own constant:

Level Constant	Tag
DEBUG	debug
INFO	info
SUCCESS	success
WARNING	warning
ERROR	error

To change the default tags for a message level (either built-in or custom), set the *MESSAGE_TAGS* setting to a dictionary containing the levels you wish to change. As this extends the default tags, you only need to provide tags for the levels you wish to override:

```
from django.contrib.messages import constants as messages
MESSAGE_TAGS = {
    messages.INFO: '',
    50: 'critical',
}
```

Using messages in views and templates

add_message(*request*, *level*, *message*, *extra_tags=''*, *fail_silently=False*)

Adding a message

To add a message, call:

```
from django.contrib import messages
messages.add_message(request, messages.INFO, 'Hello world.')
```

Some shortcut methods provide a standard way to add messages with commonly used tags (which are usually represented as HTML classes for the message):

```
messages.debug(request, '%s SQL statements were executed.' % count)
messages.info(request, 'Three credits remain in your account.')
messages.success(request, 'Profile details updated.')
messages.warning(request, 'Your account expires in three days.')
messages.error(request, 'Document deleted.')
```

Displaying messages

get_messages(*request*)

In your template, use something like:

```
{% if messages %}
<ul class="messages">
    {% for message in messages %}
    <li{% if message.tags %} class="{{ message.tags }}"{% endif %}>{{ message }}</li>
    {% endfor %}
</ul>
{% endif %}
```

If you're using the context processor, your template should be rendered with a `RequestContext`. Otherwise, ensure `messages` is available to the template context.

Even if you know there is only just one message, you should still iterate over the `messages` sequence, because otherwise the message storage will not be cleared for the next request.

The context processor also provides a `DEFAULT_MESSAGE_LEVELS` variable which is a mapping of the message level names to their numeric value:

```
{% if messages %}
<ul class="messages">
    {% for message in messages %}
    <li{% if message.tags %} class="{{ message.tags }}"{% endif %}>
        {% if message.level == DEFAULT_MESSAGE_LEVELS.ERROR %}Important: {% endif %}
        {{ message }}
    </li>
    {% endfor %}
</ul>
{% endif %}
```

Outside of templates, you can use *`get_messages()`*:

```
from django.contrib.messages import get_messages

storage = get_messages(request)
for message in storage:
    do_something_with_the_message(message)
```

For instance, you can fetch all the messages to return them in a *JSONResponseMixin* instead of a *`TemplateResponseMixin`*.

`get_messages()` will return an instance of the configured storage backend.

The `Message` class

class `storage.base.`**`Message`**

When you loop over the list of messages in a template, what you get are instances of the `Message` class. It's quite a simple object, with only a few attributes:

- `message`: The actual text of the message.
- `level`: An integer describing the type of the message (see the *message levels* section above).
- `tags`: A string combining all the message's tags (`extra_tags` and `level_tag`) separated by spaces.
- `extra_tags`: A string containing custom tags for this message, separated by spaces. It's empty by default.
- `level_tag`: The string representation of the level. By default, it's the lowercase version of the name of the associated constant, but this can be changed if you need by using the *`MESSAGE_TAGS`* setting.

Creating custom message levels

Messages levels are nothing more than integers, so you can define your own level constants and use them to create more customized user feedback, e.g.:

```
CRITICAL = 50

def my_view(request):
    messages.add_message(request, CRITICAL, 'A serious error occurred.')
```

When creating custom message levels you should be careful to avoid overloading existing levels. The values for the built-in levels are:

Level Constant	Value
DEBUG	10
INFO	20
SUCCESS	25
WARNING	30
ERROR	40

If you need to identify the custom levels in your HTML or CSS, you need to provide a mapping via the *MESSAGE_TAGS* setting.

: If you are creating a reusable application, it is recommended to use only the built-in *message levels* and not rely on any custom levels.

Changing the minimum recorded level per-request

The minimum recorded level can be set per request via the `set_level` method:

```
from django.contrib import messages

# Change the messages level to ensure the debug message is added.
messages.set_level(request, messages.DEBUG)
messages.debug(request, 'Test message...')

# In another request, record only messages with a level of WARNING and higher
messages.set_level(request, messages.WARNING)
messages.success(request, 'Your profile was updated.') # ignored
messages.warning(request, 'Your account is about to expire.') # recorded

# Set the messages level back to default.
messages.set_level(request, None)
```

Similarly, the current effective level can be retrieved with `get_level`:

```
from django.contrib import messages
current_level = messages.get_level(request)
```

For more information on how the minimum recorded level functions, see *Message levels* above.

Adding extra message tags

For more direct control over message tags, you can optionally provide a string containing extra tags to any of the add methods:

```
messages.add_message(request, messages.INFO, 'Over 9000!', extra_tags='dragonball')
messages.error(request, 'Email box full', extra_tags='email')
```

Extra tags are added before the default tag for that level and are space separated.

Failing silently when the message framework is disabled

If you're writing a reusable app (or other piece of code) and want to include messaging functionality, but don't want to require your users to enable it if they don't want to, you may pass an additional keyword argument

`fail_silently=True` to any of the `add_message` family of methods. For example:

```
messages.add_message(
    request, messages.SUCCESS, 'Profile details updated.',
    fail_silently=True,
)
messages.info(request, 'Hello world.', fail_silently=True)
```

: Setting `fail_silently=True` only hides the `MessageFailure` that would otherwise occur when the messages framework disabled and one attempts to use one of the `add_message` family of methods. It does not hide failures that may occur for other reasons.

Adding messages in class-based views

class `views.`**SuccessMessageMixin**

Adds a success message attribute to *`FormView`* based classes

get_success_message(*cleaned_data*)

`cleaned_data` is the cleaned data from the form which is used for string formatting

Example views.py:

```
from django.contrib.messages.views import SuccessMessageMixin
from django.views.generic.edit import CreateView
from myapp.models import Author

class AuthorCreate(SuccessMessageMixin, CreateView):
    model = Author
    success_url = '/success/'
    success_message = "%(name)s was created successfully"
```

The cleaned data from the `form` is available for string interpolation using the `%(field_name)s` syntax. For ModelForms, if you need access to fields from the saved `object` override the *`get_success_message()`* method.

Example views.py for ModelForms:

```
from django.contrib.messages.views import SuccessMessageMixin
from django.views.generic.edit import CreateView
from myapp.models import ComplicatedModel

class ComplicatedCreate(SuccessMessageMixin, CreateView):
    model = ComplicatedModel
    success_url = '/success/'
    success_message = "%(calculated_field)s was created successfully"

    def get_success_message(self, cleaned_data):
        return self.success_message % dict(
            cleaned_data,
            calculated_field=self.object.calculated_field,
        )
```

Expiration of messages

The messages are marked to be cleared when the storage instance is iterated (and cleared when the response is processed).

To avoid the messages being cleared, you can set the messages storage to `False` after iterating:

```
storage = messages.get_messages(request)
for message in storage:
    do_something_with(message)
storage.used = False
```

Behavior of parallel requests

Due to the way cookies (and hence sessions) work, **the behavior of any backends that make use of cookies or sessions is undefined when the same client makes multiple requests that set or get messages in parallel**. For example, if a client initiates a request that creates a message in one window (or tab) and then another that fetches any uniterated messages in another window, before the first window redirects, the message may appear in the second window instead of the first window where it may be expected.

In short, when multiple simultaneous requests from the same client are involved, messages are not guaranteed to be delivered to the same window that created them nor, in some cases, at all. Note that this is typically not a problem in most applications and will become a non-issue in HTML5, where each window/tab will have its own browsing context.

Settings

A few *settings* give you control over message behavior:

- *MESSAGE_LEVEL*
- *MESSAGE_STORAGE*
- *MESSAGE_TAGS*

For backends that use cookies, the settings for the cookie are taken from the session cookie settings:

- *SESSION_COOKIE_DOMAIN*
- *SESSION_COOKIE_SECURE*
- *SESSION_COOKIE_HTTPONLY*

6.5.8 `django.contrib.postgres`

PostgreSQL has a number of features which are not shared by the other databases Django supports. This optional module contains model fields and form fields for a number of PostgreSQL specific data types.

Psycopg2 2.5 or higher is required, though we highly recommend using the latest release. Some fields require higher versions.

: Django is, and will continue to be, a database-agnostic web framework. We would encourage those writing reusable applications for the Django community to write database-agnostic code where practical. However, we recognize that real world projects written using Django need not be database-agnostic. In fact, once a project reaches a given size changing the underlying data store is already a significant challenge and is likely to require changing the code base in some ways to handle differences between the data stores.

Django provides support for a number of data types which will only work with PostgreSQL. There is no fundamental reason why (for example) a `contrib.mysql` module does not exist, except that PostgreSQL has the richest feature set of the supported databases so its users have the most to gain.

PostgreSQL specific aggregation functions

These functions are described in more detail in the PostgreSQL docs.

: All functions come without default aliases, so you must explicitly provide one. For example:

```
>>> SomeModel.objects.aggregate(arr=ArrayAgg('somefield'))
{'arr': [0, 1, 2]}
```

General-purpose aggregation functions

ArrayAgg

class ArrayAgg(*expression*, ***extra*)

Returns a list of values, including nulls, concatenated into an array.

BitAnd

class BitAnd(*expression*, ***extra*)

Returns an `int` of the bitwise `AND` of all non-null input values, or `None` if all values are null.

BitOr

class BitOr(*expression*, ***extra*)

Returns an `int` of the bitwise `OR` of all non-null input values, or `None` if all values are null.

BoolAnd

class BoolAnd(*expression*, ***extra*)

Returns `True`, if all input values are true, `None` if all values are null or if there are no values, otherwise `False` .

BoolOr

class BoolOr(*expression*, ***extra*)

Returns `True` if at least one input value is true, `None` if all values are null or if there are no values, otherwise `False`.

StringAgg

class StringAgg(*expression*, *delimiter*)

Returns the input values concatenated into a string, separated by the `delimiter` string.

delimiter

Required argument. Needs to be a string.

Aggregate functions for statistics

y and x The arguments `y` and `x` for all these functions can be the name of a field or an expression returning a numeric data. Both are required.

Corr

class Corr(*y*, *x*)

Returns the correlation coefficient as a `float`, or `None` if there aren't any matching rows.

CovarPop

class CovarPop(*y*, *x*, *sample=False*)

Returns the population covariance as a `float`, or `None` if there aren't any matching rows.

Has one optional argument:

sample

By default `CovarPop` returns the general population covariance. However, if `sample=True`, the return value will be the sample population covariance.

RegrAvgX

class RegrAvgX(*y*, *x*)

Returns the average of the independent variable (`sum(x)/N`) as a `float`, or `None` if there aren't any matching rows.

RegrAvgY

class RegrAvgY(*y*, *x*)

Returns the average of the dependent variable (`sum(y)/N`) as a `float`, or `None` if there aren't any matching rows.

RegrCount

class RegrCount(*y*, *x*)

Returns an `int` of the number of input rows in which both expressions are not null.

RegrIntercept

class RegrIntercept(*y*, *x*)

Returns the y-intercept of the least-squares-fit linear equation determined by the `(x, y)` pairs as a `float`, or `None` if there aren't any matching rows.

RegrR2

class RegrR2(*y*, *x*)

Returns the square of the correlation coefficient as a `float`, or `None` if there aren't any matching rows.

RegrSlope

class RegrSlope(*y*, *x*)

Returns the slope of the least-squares-fit linear equation determined by the `(x, y)` pairs as a `float`, or `None` if there aren't any matching rows.

RegrSXX

class RegrSXX(*y*, *x*)

Returns `sum(x^2) - sum(x)^2/N` ("sum of squares" of the independent variable) as a `float`, or `None` if there aren't any matching rows.

RegrSXY

class RegrSXY(*y*, *x*)

Returns `sum(x*y) - sum(x) * sum(y)/N` ("sum of products" of independent times dependent variable) as a `float`, or `None` if there aren't any matching rows.

RegrSYY

class RegrSYY(*y*, *x*)

Returns `sum(y^2) - sum(y)^2/N` ("sum of squares" of the dependent variable) as a `float`, or `None` if there aren't any matching rows.

Usage examples

We will use this example table:

```
| FIELD1 | FIELD2 | FIELD3 |
|--------|--------|--------|
|    foo |      1 |     13 |
|    bar |      2 | (null) |
|   test |      3 |     13 |
```

Here's some examples of some of the general-purpose aggregation functions:

```
>>> TestModel.objects.aggregate(result=StringAgg('field1', delimiter=';'))
{'result': 'foo;bar;test'}
>>> TestModel.objects.aggregate(result=ArrayAgg('field2'))
{'result': [1, 2, 3]}
>>> TestModel.objects.aggregate(result=ArrayAgg('field1'))
{'result': ['foo', 'bar', 'test']}
```

The next example shows the usage of statistical aggregate functions. The underlying math will be not described (you can read about this, for example, at wikipedia):

```
>>> TestModel.objects.aggregate(count=RegrCount(y='field3', x='field2'))
{'count': 2}
>>> TestModel.objects.aggregate(avgx=RegrAvgX(y='field3', x='field2'),
...                             avgy=RegrAvgY(y='field3', x='field2'))
{'avgx': 2, 'avgy': 13}
```

PostgreSQL specific model fields

All of these fields are available from the `django.contrib.postgres.fields` module.

ArrayField

class ArrayField(*base_field*, *size=None*, ***options*)

A field for storing lists of data. Most field types can be used, you simply pass another field instance as the *base_field*. You may also specify a *size*. `ArrayField` can be nested to store multi-dimensional arrays.

If you give the field a *default*, ensure it's a callable such as `list` (for an empty default) or a callable that returns a list (such as a function). Incorrectly using `default=[]` creates a mutable default that is shared between all instances of `ArrayField`.

base_field

This is a required argument.

Specifies the underlying data type and behavior for the array. It should be an instance of a subclass of *Field*. For example, it could be an *IntegerField* or a *CharField*. Most field types are permitted, with the exception of those handling relational data (*ForeignKey*, *OneToOneField* and *ManyToManyField*).

It is possible to nest array fields - you can specify an instance of `ArrayField` as the `base_field`. For example:

```
from django.db import models
from django.contrib.postgres.fields import ArrayField

class ChessBoard(models.Model):
    board = ArrayField(
        ArrayField(
            models.CharField(max_length=10, blank=True),
            size=8,
        ),
        size=8,
    )
```

Transformation of values between the database and the model, validation of data and configuration, and serialization are all delegated to the underlying base field.

size

This is an optional argument.

If passed, the array will have a maximum size as specified. This will be passed to the database, although PostgreSQL at present does not enforce the restriction.

: When nesting `ArrayField`, whether you use the *size* parameter or not, PostgreSQL requires that the arrays are rectangular:

```
from django.contrib.postgres.fields import ArrayField
from django.db import models

class Board(models.Model):
    pieces = ArrayField(ArrayField(models.IntegerField()))

# Valid
Board(pieces=[
    [2, 3],
    [2, 1],
])

# Not valid
Board(pieces=[
    [2, 3],
    [2],
])
```

If irregular shapes are required, then the underlying field should be made nullable and the values padded with `None`.

Querying `ArrayField` There are a number of custom lookups and transforms for *`ArrayField`*. We will use the following example model:

```
from django.db import models
from django.contrib.postgres.fields import ArrayField

class Post(models.Model):
    name = models.CharField(max_length=200)
    tags = ArrayField(models.CharField(max_length=200), blank=True)
```

```
    def __str__(self):  # __unicode__ on Python 2
        return self.name
```

contains The *contains* lookup is overridden on *ArrayField*. The returned objects will be those where the values passed are a subset of the data. It uses the SQL operator `@>`. For example:

```
>>> Post.objects.create(name='First post', tags=['thoughts', 'django'])
>>> Post.objects.create(name='Second post', tags=['thoughts'])
>>> Post.objects.create(name='Third post', tags=['tutorial', 'django'])

>>> Post.objects.filter(tags__contains=['thoughts'])
<QuerySet [<Post: First post>, <Post: Second post>]>

>>> Post.objects.filter(tags__contains=['django'])
<QuerySet [<Post: First post>, <Post: Third post>]>

>>> Post.objects.filter(tags__contains=['django', 'thoughts'])
<QuerySet [<Post: First post>]>
```

contained_by This is the inverse of the *contains* lookup - the objects returned will be those where the data is a subset of the values passed. It uses the SQL operator `<@`. For example:

```
>>> Post.objects.create(name='First post', tags=['thoughts', 'django'])
>>> Post.objects.create(name='Second post', tags=['thoughts'])
>>> Post.objects.create(name='Third post', tags=['tutorial', 'django'])

>>> Post.objects.filter(tags__contained_by=['thoughts', 'django'])
<QuerySet [<Post: First post>, <Post: Second post>]>

>>> Post.objects.filter(tags__contained_by=['thoughts', 'django', 'tutorial'])
<QuerySet [<Post: First post>, <Post: Second post>, <Post: Third post>]>
```

overlap Returns objects where the data shares any results with the values passed. Uses the SQL operator `&&`. For example:

```
>>> Post.objects.create(name='First post', tags=['thoughts', 'django'])
>>> Post.objects.create(name='Second post', tags=['thoughts'])
>>> Post.objects.create(name='Third post', tags=['tutorial', 'django'])

>>> Post.objects.filter(tags__overlap=['thoughts'])
<QuerySet [<Post: First post>, <Post: Second post>]>

>>> Post.objects.filter(tags__overlap=['thoughts', 'tutorial'])
<QuerySet [<Post: First post>, <Post: Second post>, <Post: Third post>]>
```

len Returns the length of the array. The lookups available afterwards are those available for *IntegerField*. For example:

```
>>> Post.objects.create(name='First post', tags=['thoughts', 'django'])
>>> Post.objects.create(name='Second post', tags=['thoughts'])

>>> Post.objects.filter(tags__len=1)
<QuerySet [<Post: Second post>]>
```

Index transforms This class of transforms allows you to index into the array in queries. Any non-negative integer can be used. There are no errors if it exceeds the *size* of the array. The lookups available after the transform are those from the *base_field*. For example:

```
>>> Post.objects.create(name='First post', tags=['thoughts', 'django'])
>>> Post.objects.create(name='Second post', tags=['thoughts'])

>>> Post.objects.filter(tags__0='thoughts')
<QuerySet [<Post: First post>, <Post: Second post>]>

>>> Post.objects.filter(tags__1__iexact='Django')
<QuerySet [<Post: First post>]>

>>> Post.objects.filter(tags__276='javascript')
<QuerySet []>
```

: PostgreSQL uses 1-based indexing for array fields when writing raw SQL. However these indexes and those used in *slices* use 0-based indexing to be consistent with Python.

Slice transforms This class of transforms allow you to take a slice of the array. Any two non-negative integers can be used, separated by a single underscore. The lookups available after the transform do not change. For example:

```
>>> Post.objects.create(name='First post', tags=['thoughts', 'django'])
>>> Post.objects.create(name='Second post', tags=['thoughts'])
>>> Post.objects.create(name='Third post', tags=['django', 'python', 'thoughts'])

>>> Post.objects.filter(tags__0_1=['thoughts'])
<QuerySet [<Post: First post>, <Post: Second post>]>

>>> Post.objects.filter(tags__0_2__contains=['thoughts'])
<QuerySet [<Post: First post>, <Post: Second post>]>
```

: PostgreSQL uses 1-based indexing for array fields when writing raw SQL. However these slices and those used in *indexes* use 0-based indexing to be consistent with Python.

Multidimensional arrays with indexes and slices

PostgreSQL has some rather esoteric behavior when using indexes and slices on multidimensional arrays. It will always work to use indexes to reach down to the final underlying data, but most other slices behave strangely at the database level and cannot be supported in a logical, consistent fashion by Django.

Indexing `ArrayField` At present using *db_index* will create a `btree` index. This does not offer particularly significant help to querying. A more useful index is a `GIN` index, which you should create using a *RunSQL* operation.

HStoreField

class `HStoreField`(***options*)

A field for storing mappings of strings to strings. The Python data type used is a `dict`.

To use this field, you'll need to:

1. Add `'django.contrib.postgres'` in your *INSTALLED_APPS*.
2. Setup the hstore extension in PostgreSQL before the first `CreateModel` or `AddField` operation by adding a migration with the *HStoreExtension* operation. For example:

```
from django.contrib.postgres.operations import HStoreExtension

class Migration(migrations.Migration):
    ...

    operations = [
        HStoreExtension(),
        ...
    ]
```

Creating the extension requires a database user with superuser privileges. If the Django database user doesn't have superuser privileges, you'll have to create the extension outside of Django migrations with a user that has the appropriate privileges. In that case, connect to your Django database and run the query `CREATE EXTENSION IF NOT EXISTS hstore;`

You'll see an error like `can't adapt type 'dict'` if you skip the first step, or `type "hstore" does not exist` if you skip the second.

: On occasions it may be useful to require or restrict the keys which are valid for a given field. This can be done using the *KeysValidator*.

Querying `HStoreField` In addition to the ability to query by key, there are a number of custom lookups available for `HStoreField`.

We will use the following example model:

```
from django.contrib.postgres.fields import HStoreField
from django.db import models

class Dog(models.Model):
    name = models.CharField(max_length=200)
    data = HStoreField()

    def __str__(self):  # __unicode__ on Python 2
        return self.name
```

Key lookups To query based on a given key, you simply use that key as the lookup name:

```
>>> Dog.objects.create(name='Rufus', data={'breed': 'labrador'})
>>> Dog.objects.create(name='Meg', data={'breed': 'collie'})

>>> Dog.objects.filter(data__breed='collie')
<QuerySet [<Dog: Meg>]>
```

You can chain other lookups after key lookups:

```
>>> Dog.objects.filter(data__breed__contains='l')
<QuerySet [<Dog: Rufus>, <Dog: Meg>]>
```

If the key you wish to query by clashes with the name of another lookup, you need to use the *hstorefield.contains* lookup instead.

: Since any string could be a key in a hstore value, any lookup other than those listed below will be interpreted as a key lookup. No errors are raised. Be extra careful for typing mistakes, and always check your queries work as you intend.

contains The *contains* lookup is overridden on *HStoreField*. The returned objects are those where the given `dict` of key-value pairs are all contained in the field. It uses the SQL operator `@>`. For example:

```
>>> Dog.objects.create(name='Rufus', data={'breed': 'labrador', 'owner': 'Bob'})
>>> Dog.objects.create(name='Meg', data={'breed': 'collie', 'owner': 'Bob'})
>>> Dog.objects.create(name='Fred', data={})

>>> Dog.objects.filter(data__contains={'owner': 'Bob'})
<QuerySet [<Dog: Rufus>, <Dog: Meg>]>

>>> Dog.objects.filter(data__contains={'breed': 'collie'})
<QuerySet [<Dog: Meg>]>
```

contained_by This is the inverse of the *contains* lookup - the objects returned will be those where the key-value pairs on the object are a subset of those in the value passed. It uses the SQL operator `<@`. For example:

```
>>> Dog.objects.create(name='Rufus', data={'breed': 'labrador', 'owner': 'Bob'})
>>> Dog.objects.create(name='Meg', data={'breed': 'collie', 'owner': 'Bob'})
>>> Dog.objects.create(name='Fred', data={})

>>> Dog.objects.filter(data__contained_by={'breed': 'collie', 'owner': 'Bob'})
<QuerySet [<Dog: Meg>, <Dog: Fred>]>

>>> Dog.objects.filter(data__contained_by={'breed': 'collie'})
<QuerySet [<Dog: Fred>]>
```

has_key Returns objects where the given key is in the data. Uses the SQL operator `?`. For example:

```
>>> Dog.objects.create(name='Rufus', data={'breed': 'labrador'})
>>> Dog.objects.create(name='Meg', data={'breed': 'collie', 'owner': 'Bob'})

>>> Dog.objects.filter(data__has_key='owner')
<QuerySet [<Dog: Meg>]>
```

has_any_keys Returns objects where any of the given keys are in the data. Uses the SQL operator `?|`. For example:

```
>>> Dog.objects.create(name='Rufus', data={'breed': 'labrador'})
>>> Dog.objects.create(name='Meg', data={'owner': 'Bob'})
>>> Dog.objects.create(name='Fred', data={})

>>> Dog.objects.filter(data__has_any_keys=['owner', 'breed'])
<QuerySet [<Dog: Rufus>, <Dog: Meg>]>
```

has_keys Returns objects where all of the given keys are in the data. Uses the SQL operator `?&`. For example:

```
>>> Dog.objects.create(name='Rufus', data={})
>>> Dog.objects.create(name='Meg', data={'breed': 'collie', 'owner': 'Bob'})

>>> Dog.objects.filter(data__has_keys=['breed', 'owner'])
<QuerySet [<Dog: Meg>]>
```

keys Returns objects where the array of keys is the given value. Note that the order is not guaranteed to be reliable, so this transform is mainly useful for using in conjunction with lookups on *ArrayField*. Uses the SQL function `akeys()`. For example:

```
>>> Dog.objects.create(name='Rufus', data={'toy': 'bone'})
>>> Dog.objects.create(name='Meg', data={'breed': 'collie', 'owner': 'Bob'})

>>> Dog.objects.filter(data__keys__overlap=['breed', 'toy'])
<QuerySet [<Dog: Rufus>, <Dog: Meg>]>
```

values Returns objects where the array of values is the given value. Note that the order is not guaranteed to be reliable, so this transform is mainly useful for using in conjunction with lookups on *ArrayField*. Uses the SQL function `avalues()`. For example:

```
>>> Dog.objects.create(name='Rufus', data={'breed': 'labrador'})
>>> Dog.objects.create(name='Meg', data={'breed': 'collie', 'owner': 'Bob'})

>>> Dog.objects.filter(data__values__contains=['collie'])
<QuerySet [<Dog: Meg>]>
```

JSONField

class JSONField(***options*)

A field for storing JSON encoded data. In Python the data is represented in its Python native format: dictionaries, lists, strings, numbers, booleans and `None`.

If you want to store other data types, you'll need to serialize them first. For example, you might cast a `datetime` to a string. You might also want to convert the string back to a `datetime` when you retrieve the data from the database. There are some third-party `JSONField` implementations which do this sort of thing automatically.

If you give the field a *default*, ensure it's a callable such as `dict` (for an empty default) or a callable that returns a dict (such as a function). Incorrectly using `default={}` creates a mutable default that is shared between all instances of `JSONField`.

: PostgreSQL has two native JSON based data types: `json` and `jsonb`. The main difference between them is how they are stored and how they can be queried. PostgreSQL's `json` field is stored as the original string representation of the JSON and must be decoded on the fly when queried based on keys. The `jsonb` field is stored based on the actual structure of the JSON which allows indexing. The trade-off is a small additional cost on writing to the `jsonb` field. `JSONField` uses `jsonb`.

As a result, this field requires PostgreSQL $\geq$ 9.4 and Psycopg2 $\geq$ 2.5.4.

Querying JSONField We will use the following example model:

```
from django.contrib.postgres.fields import JSONField
from django.db import models

class Dog(models.Model):
    name = models.CharField(max_length=200)
    data = JSONField()

    def __str__(self):  # __unicode__ on Python 2
        return self.name
```

Key, index, and path lookups To query based on a given dictionary key, simply use that key as the lookup name:

```
>>> Dog.objects.create(name='Rufus', data={
...     'breed': 'labrador',
...     'owner': {
...         'name': 'Bob',
...         'other_pets': [{
...             'name': 'Fishy',
...         }],
...     },
... })
>>> Dog.objects.create(name='Meg', data={'breed': 'collie'})

>>> Dog.objects.filter(data__breed='collie')
<QuerySet [<Dog: Meg>]>
```

Multiple keys can be chained together to form a path lookup:

```
>>> Dog.objects.filter(data__owner__name='Bob')
<QuerySet [<QuerySet <Dog: Rufus>]>
```

If the key is an integer, it will be interpreted as an index lookup in an array:

```
>>> Dog.objects.filter(data__owner__other_pets__0__name='Fishy')
<QuerySet [<Dog: Rufus>]>
```

If the key you wish to query by clashes with the name of another lookup, use the *jsonfield.contains* lookup instead.

If only one key or index is used, the SQL operator -> is used. If multiple operators are used then the #> operator is used.

: Since any string could be a key in a JSON object, any lookup other than those listed below will be interpreted as a key lookup. No errors are raised. Be extra careful for typing mistakes, and always check your queries work as you intend.

Containment and key operations *JSONField* shares lookups relating to containment and keys with *HStoreField*.

- *contains* (accepts any JSON rather than just a dictionary of strings)
- *contained_by* (accepts any JSON rather than just a dictionary of strings)
- *has_key*
- *has_any_keys*
- *has_keys*

Range Fields

There are five range field types, corresponding to the built-in range types in PostgreSQL. These fields are used to store a range of values; for example the start and end timestamps of an event, or the range of ages an activity is suitable for.

All of the range fields translate to psycopg2 Range objects in python, but also accept tuples as input if no bounds information is necessary. The default is lower bound included, upper bound excluded; that is, `[)`.

IntegerRangeField

class IntegerRangeField(***options*)

Stores a range of integers. Based on an *IntegerField*. Represented by an `int4range` in the database and a `NumericRange` in Python.

Regardless of the bounds specified when saving the data, PostgreSQL always returns a range in a canonical form that includes the lower bound and excludes the upper bound; that is `[)`.

BigIntegerRangeField

class BigIntegerRangeField(***options*)

Stores a range of large integers. Based on a *BigIntegerField*. Represented by an `int8range` in the database and a `NumericRange` in Python.

Regardless of the bounds specified when saving the data, PostgreSQL always returns a range in a canonical form that includes the lower bound and excludes the upper bound; that is `[)`.

FloatRangeField

class FloatRangeField(***options*)

Stores a range of floating point values. Based on a *FloatField*. Represented by a `numrange` in the database and a `NumericRange` in Python.

DateTimeRangeField

class DateTimeRangeField(***options*)

Stores a range of timestamps. Based on a *DateTimeField*. Represented by a `tztsrange` in the database and a `DateTimeTZRange` in Python.

DateRangeField

class DateRangeField(***options*)

Stores a range of dates. Based on a *DateField*. Represented by a `daterange` in the database and a `DateRange` in Python.

Regardless of the bounds specified when saving the data, PostgreSQL always returns a range in a canonical form that includes the lower bound and excludes the upper bound; that is `[)`.

Querying Range Fields There are a number of custom lookups and transforms for range fields. They are available on all the above fields, but we will use the following example model:

```
from django.contrib.postgres.fields import IntegerRangeField
from django.db import models

class Event(models.Model):
    name = models.CharField(max_length=200)
    ages = IntegerRangeField()
    start = models.DateTimeField()
```

```
    def __str__(self):  # __unicode__ on Python 2
        return self.name
```

We will also use the following example objects:

```
>>> import datetime
>>> from django.utils import timezone
>>> now = timezone.now()
>>> Event.objects.create(name='Soft play', ages=(0, 10), start=now)
>>> Event.objects.create(name='Pub trip', ages=(21, None), start=now - datetime.timedelta(days=1))
```

and `NumericRange`:

```
>>> from psycopg2.extras import NumericRange
```

Containment functions As with other PostgreSQL fields, there are three standard containment operators: `contains`, `contained_by` and `overlap`, using the SQL operators `@>`, `<@`, and `&&` respectively.

contains

```
>>> Event.objects.filter(ages__contains=NumericRange(4, 5))
<QuerySet [<Event: Soft play>]>
```

contained_by

```
>>> Event.objects.filter(ages__contained_by=NumericRange(0, 15))
<QuerySet [<Event: Soft play>]>
```

overlap

```
>>> Event.objects.filter(ages__overlap=NumericRange(8, 12))
<QuerySet [<Event: Soft play>]>
```

Comparison functions Range fields support the standard lookups: *lt*, *gt*, *lte* and *gte*. These are not particularly helpful - they compare the lower bounds first and then the upper bounds only if necessary. This is also the strategy used to order by a range field. It is better to use the specific range comparison operators.

fully_lt The returned ranges are strictly less than the passed range. In other words, all the points in the returned range are less than all those in the passed range.

```
>>> Event.objects.filter(ages__fully_lt=NumericRange(11, 15))
<QuerySet [<Event: Soft play>]>
```

fully_gt The returned ranges are strictly greater than the passed range. In other words, the all the points in the returned range are greater than all those in the passed range.

```
>>> Event.objects.filter(ages__fully_gt=NumericRange(11, 15))
<QuerySet [<Event: Pub trip>]>
```

not_lt The returned ranges do not contain any points less than the passed range, that is the lower bound of the returned range is at least the lower bound of the passed range.

```
>>> Event.objects.filter(ages__not_lt=NumericRange(0, 15))
<QuerySet [<Event: Soft play>, <Event: Pub trip>]>
```

not_gt The returned ranges do not contain any points greater than the passed range, that is the upper bound of the returned range is at most the upper bound of the passed range.

```
>>> Event.objects.filter(ages__not_gt=NumericRange(3, 10))
<QuerySet [<Event: Soft play>]>
```

adjacent_to The returned ranges share a bound with the passed range.

```
>>> Event.objects.filter(ages__adjacent_to=NumericRange(10, 21))
<QuerySet [<Event: Soft play>, <Event: Pub trip>]>
```

Querying using the bounds There are three transforms available for use in queries. You can extract the lower or upper bound, or query based on emptiness.

startswith Returned objects have the given lower bound. Can be chained to valid lookups for the base field.

```
>>> Event.objects.filter(ages__startswith=21)
<QuerySet [<Event: Pub trip>]>
```

endswith Returned objects have the given upper bound. Can be chained to valid lookups for the base field.

```
>>> Event.objects.filter(ages__endswith=10)
<QuerySet [<Event: Soft play>]>
```

isempty Returned objects are empty ranges. Can be chained to valid lookups for a *BooleanField*.

```
>>> Event.objects.filter(ages__isempty=True)
<QuerySet []>
```

Defining your own range types PostgreSQL allows the definition of custom range types. Django's model and form field implementations use base classes below, and psycopg2 provides a `register_range()` to allow use of custom range types.

class RangeField(***options*)
Base class for model range fields.

base_field
The model field class to use.

range_type
The psycopg2 range type to use.

form_field
The form field class to use. Should be a subclass of *django.contrib.postgres.forms.BaseRangeField*.

class django.contrib.postgres.forms.**BaseRangeField**
Base class for form range fields.

base_field
The form field to use.

range_type
The psycopg2 range type to use.

PostgreSQL specific form fields and widgets

All of these fields and widgets are available from the `django.contrib.postgres.forms` module.

Fields

SimpleArrayField

class SimpleArrayField(*base_field*, *delimiter=','*, *max_length=None*, *min_length=None*)
A simple field which maps to an array. It is represented by an HTML `<input>`.

base_field
This is a required argument.

It specifies the underlying form field for the array. This is not used to render any HTML, but it is used to process the submitted data and validate it. For example:

```
>>> from django.contrib.postgres.forms import SimpleArrayField
>>> from django import forms

>>> class NumberListForm(forms.Form):
...     numbers = SimpleArrayField(forms.IntegerField())

>>> form = NumberListForm({'numbers': '1,2,3'})
>>> form.is_valid()
True
>>> form.cleaned_data
{'numbers': [1, 2, 3]}

>>> form = NumberListForm({'numbers': '1,2,a'})
>>> form.is_valid()
False
```

delimiter
This is an optional argument which defaults to a comma: `,`. This value is used to split the submitted data. It allows you to chain `SimpleArrayField` for multidimensional data:

```
>>> from django.contrib.postgres.forms import SimpleArrayField
>>> from django import forms

>>> class GridForm(forms.Form):
...     places = SimpleArrayField(SimpleArrayField(IntegerField()), delimiter='|')

>>> form = GridForm({'places': '1,2|2,1|4,3'})
>>> form.is_valid()
True
>>> form.cleaned_data
{'places': [[1, 2], [2, 1], [4, 3]]}
```

: The field does not support escaping of the delimiter, so be careful in cases where the delimiter is a valid

character in the underlying field. The delimiter does not need to be only one character.

max_length
This is an optional argument which validates that the array does not exceed the stated length.

min_length
This is an optional argument which validates that the array reaches at least the stated length.

User friendly forms

`SimpleArrayField` is not particularly user friendly in most cases, however it is a useful way to format data from a client-side widget for submission to the server.

SplitArrayField

class SplitArrayField(*base_field*, *size*, *remove_trailing_nulls=False*)
This field handles arrays by reproducing the underlying field a fixed number of times.

base_field
This is a required argument. It specifies the form field to be repeated.

size
This is the fixed number of times the underlying field will be used.

remove_trailing_nulls
By default, this is set to `False`. When `False`, each value from the repeated fields is stored. When set to `True`, any trailing values which are blank will be stripped from the result. If the underlying field has `required=True`, but `remove_trailing_nulls` is `True`, then null values are only allowed at the end, and will be stripped.

Some examples:

```
SplitArrayField(IntegerField(required=True), size=3, remove_trailing_nulls=False)

['1', '2', '3']  # -> [1, 2, 3]
['1', '2', '']  # -> ValidationError - third entry required.
['1', '', '3']  # -> ValidationError - second entry required.
['', '2', '']  # -> ValidationError - first and third entries required.

SplitArrayField(IntegerField(required=False), size=3, remove_trailing_nulls=False)

['1', '2', '3']  # -> [1, 2, 3]
['1', '2', '']  # -> [1, 2, None]
['1', '', '3']  # -> [1, None, 3]
['', '2', '']  # -> [None, 2, None]

SplitArrayField(IntegerField(required=True), size=3, remove_trailing_nulls=True)

['1', '2', '3']  # -> [1, 2, 3]
['1', '2', '']  # -> [1, 2]
['1', '', '3']  # -> ValidationError - second entry required.
['', '2', '']  # -> ValidationError - first entry required.

SplitArrayField(IntegerField(required=False), size=3, remove_trailing_nulls=True)

['1', '2', '3']  # -> [1, 2, 3]
['1', '2', '']  # -> [1, 2]
```

```
        ['1', '', '3']  # -> [1, None, 3]
        ['', '2', '']  # -> [None, 2]
```

HStoreField

class HStoreField

A field which accepts JSON encoded data for an *HStoreField*. It will cast all the values to strings. It is represented by an HTML `<textarea>`.

User friendly forms

`HStoreField` is not particularly user friendly in most cases, however it is a useful way to format data from a client-side widget for submission to the server.

: On occasions it may be useful to require or restrict the keys which are valid for a given field. This can be done using the *KeysValidator*.

JSONField

class JSONField

A field which accepts JSON encoded data for a *JSONField*. It is represented by an HTML `<textarea>`.

User friendly forms

`JSONField` is not particularly user friendly in most cases, however it is a useful way to format data from a client-side widget for submission to the server.

Range Fields This group of fields all share similar functionality for accepting range data. They are based on *MultiValueField*. They treat one omitted value as an unbounded range. They also validate that the lower bound is not greater than the upper bound. All of these fields use *RangeWidget*.

IntegerRangeField

class IntegerRangeField

Based on *IntegerField* and translates its input into `NumericRange`. Default for *IntegerRangeField* and *BigIntegerRangeField*.

FloatRangeField

class FloatRangeField

Based on *FloatField* and translates its input into `NumericRange`. Default for *FloatRangeField*.

DateTimeRangeField

class DateTimeRangeField

Based on *DateTimeField* and translates its input into `DateTimeTZRange`. Default for *DateTimeRangeField*.

DateRangeField

class DateRangeField

Based on *DateField* and translates its input into DateRange. Default for *DateRangeField*.

Widgets

RangeWidget

class RangeWidget(*base_widget*, *attrs=None*)

Widget used by all of the range fields. Based on *MultiWidget*.

RangeWidget has one required argument:

base_widget

A *RangeWidget* comprises a 2-tuple of base_widget.

decompress(*value*)

Takes a single "compressed" value of a field, for example a *DateRangeField*, and returns a tuple representing and lower and upper bound.

PostgreSQL specific database functions

All of these functions are available from the django.contrib.postgres.functions module.

TransactionNow

class TransactionNow

Returns the date and time on the database server that the current transaction started. If you are not in a transaction it will return the date and time of the current statement. This is a complement to *django.db.models.functions.Now*, which returns the date and time of the current statement.

Note that only the outermost call to *atomic()* sets up a transaction and thus sets the time that TransactionNow() will return; nested calls create savepoints which do not affect the transaction time.

Usage example:

```
>>> from django.contrib.postgres.functions import TransactionNow
>>> Article.objects.filter(published__lte=TransactionNow())
<QuerySet [<Article: How to Django>]>
```

PostgreSQL specific lookups

Trigram similarity

The trigram_similar lookup allows you to perform trigram lookups, measuring the number of trigrams (three consecutive characters) shared, using a dedicated PostgreSQL extension. A trigram lookup is given an expression and returns results that have a similarity measurement greater than the current similarity threshold.

To use it, add 'django.contrib.postgres' in your *INSTALLED_APPS* and activate the pg_trgm extension on PostgreSQL. You can install the extension using the *TrigramExtension* migration operation.

The trigram_similar lookup can be used on *CharField* and *TextField*:

```
>>> City.objects.filter(name__trigram_similar="Middlesborough")
['<City: Middlesbrough>']
```

Unaccent

The `unaccent` lookup allows you to perform accent-insensitive lookups using a dedicated PostgreSQL extension.

This lookup is implemented using *`Transform`*, so it can be chained with other lookup functions. To use it, you need to add `'django.contrib.postgres'` in your *`INSTALLED_APPS`* and activate the unaccent extension on PostgreSQL. The *`UnaccentExtension`* migration operation is available if you want to perform this activation using migrations).

The `unaccent` lookup can be used on *`CharField`* and *`TextField`*:

```
>>> City.objects.filter(name__unaccent="México")
['<City: Mexico>']

>>> User.objects.filter(first_name__unaccent__startswith="Jerem")
['<User: Jeremy>', '<User: Jérémy>', '<User: Jérémie>', '<User: Jeremie>']
```

: `unaccent` lookups should perform fine in most use cases. However, queries using this filter will generally perform full table scans, which can be slow on large tables. In those cases, using dedicated full text indexing tools might be appropriate.

Database migration operations

All of these operations are available from the `django.contrib.postgres.operations` module.

CreateExtension

class `CreateExtension`(*name*)

An `Operation` subclass which installs PostgreSQL extensions.

`name`

This is a required argument. The name of the extension to be installed.

HStoreExtension

class `HStoreExtension`

Installs the `hstore` extension and also sets up the connection to interpret hstore data for possible use in subsequent migrations.

TrigramExtension

class `TrigramExtension`

Installs the `pg_trgm` extension.

UnaccentExtension

class `UnaccentExtension`

Installs the `unaccent` extension.

Full text search

The database functions in the `django.contrib.postgres.search` module ease the use of PostgreSQL's full text search engine.

For the examples in this document, we'll use the models defined in Making queries.

:

For a high-level overview of searching, see the topic documentation.

The `search` lookup

The simplest way to use full text search is to search a single term against a single column in the database. For example:

```
>>> Entry.objects.filter(body_text__search='Cheese')
[<Entry: Cheese on Toast recipes>, <Entry: Pizza Recipes>]
```

This creates a `to_tsvector` in the database from the `body_text` field and a `plainto_tsquery` from the search term `'Cheese'`, both using the default database search configuration. The results are obtained by matching the query and the vector.

To use the `search` lookup, `'django.contrib.postgres'` must be in your *INSTALLED_APPS*.

SearchVector

class SearchVector(**expressions*, *config=None*, *weight=None*)

Searching against a single field is great but rather limiting. The `Entry` instances we're searching belong to a `Blog`, which has a `tagline` field. To query against both fields, use a `SearchVector`:

```
>>> from django.contrib.postgres.search import SearchVector
>>> Entry.objects.annotate(
...     search=SearchVector('body_text', 'blog__tagline'),
... ).filter(search='Cheese')
[<Entry: Cheese on Toast recipes>, <Entry: Pizza Recipes>]
```

The arguments to `SearchVector` can be any *Expression* or the name of a field. Multiple arguments will be concatenated together using a space so that the search document includes them all.

`SearchVector` objects can be combined together, allowing you to reuse them. For example:

```
>>> Entry.objects.annotate(
...     search=SearchVector('body_text') + SearchVector('blog__tagline'),
... ).filter(search='Cheese')
[<Entry: Cheese on Toast recipes>, <Entry: Pizza Recipes>]
```

See *Changing the search configuration* and *Weighting queries* for an explanation of the `config` and `weight` parameters.

SearchQuery

class SearchQuery(*value*, *config=None*)

`SearchQuery` translates the terms the user provides into a search query object that the database compares to a search vector. By default, all the words the user provides are passed through the stemming algorithms, and then it looks for matches for all of the resulting terms.

`SearchQuery` terms can be combined logically to provide more flexibility:

```
>>> from django.contrib.postgres.search import SearchQuery
>>> SearchQuery('potato') & SearchQuery('ireland')  # potato AND ireland
>>> SearchQuery('potato') | SearchQuery('penguin')  # potato OR penguin
>>> ~SearchQuery('sausage')  # NOT sausage
```

See *Changing the search configuration* for an explanation of the `config` parameter.

SearchRank

class SearchRank (*vector*, *query*, *weights=None*)

So far, we've just returned the results for which any match between the vector and the query are possible. It's likely you may wish to order the results by some sort of relevancy. PostgreSQL provides a ranking function which takes into account how often the query terms appear in the document, how close together the terms are in the document, and how important the part of the document is where they occur. The better the match, the higher the value of the rank. To order by relevancy:

```
>>> from django.contrib.postgres.search import SearchQuery, SearchRank, SearchVector
>>> vector = SearchVector('body_text')
>>> query = SearchQuery('cheese')
>>> Entry.objects.annotate(rank=SearchRank(vector, query)).order_by('-rank')
[<Entry: Cheese on Toast recipes>, <Entry: Pizza recipes>]
```

See *Weighting queries* for an explanation of the `weights` parameter.

Changing the search configuration

You can specify the `config` attribute to a *SearchVector* and *SearchQuery* to use a different search configuration. This allows using a different language parsers and dictionaries as defined by the database:

```
>>> from django.contrib.postgres.search import SearchQuery, SearchVector
>>> Entry.objects.annotate(
...     search=SearchVector('body_text', config='french'),
... ).filter(search=SearchQuery('œuf', config='french'))
[<Entry: Pain perdu>]
```

The value of `config` could also be stored in another column:

```
>>> from django.db.models import F
>>> Entry.objects.annotate(
...     search=SearchVector('body_text', config=F('blog__language')),
... ).filter(search=SearchQuery('œuf', config=F('blog__language')))
[<Entry: Pain perdu>]
```

Weighting queries

Every field may not have the same relevance in a query, so you can set weights of various vectors before you combine them:

```
>>> from django.contrib.postgres.search import SearchQuery, SearchRank, SearchVector
>>> vector = SearchVector('body_text', weight='A') + SearchVector('blog__tagline', weight='B')
>>> query = SearchQuery('cheese')
>>> Entry.objects.annotate(rank=SearchRank(vector, query)).filter(rank__gte=0.3).order_by('rank')
```

The weight should be one of the following letters: D, C, B, A. By default, these weights refer to the numbers `0.1`, `0.2`, `0.4`, and `1.0`, respectively. If you wish to weight them differently, pass a list of four floats to *SearchRank* as `weights` in the same order above:

```
>>> rank = SearchRank(vector, query, weights=[0.2, 0.4, 0.6, 0.8])
>>> Entry.objects.annotate(rank=rank).filter(rank__gte=0.3).order_by('-rank')
```

Performance

Special database configuration isn't necessary to use any of these functions, however, if you're searching more than a few hundred records, you're likely to run into performance problems. Full text search is a more intensive process than comparing the size of an integer, for example.

In the event that all the fields you're querying on are contained within one particular model, you can create a functional index which matches the search vector you wish to use. The PostgreSQL documentation has details on creating indexes for full text search.

SearchVectorField

class SearchVectorField

If this approach becomes too slow, you can add a `SearchVectorField` to your model. You'll need to keep it populated with triggers, for example, as described in the PostgreSQL documentation. You can then query the field as if it were an annotated `SearchVector`:

```
>>> Entry.objects.update(search_vector=SearchVector('body_text'))
>>> Entry.objects.filter(search_vector='cheese')
[<Entry: Cheese on Toast recipes>, <Entry: Pizza recipes>]
```

Trigram similarity

Another approach to searching is trigram similarity. A trigram is a group of three consecutive characters. In addition to the *trigram_similar* lookup, you can use a couple of other expressions.

To use them, you need to activate the pg_trgm extension on PostgreSQL. You can install it using the *TrigramExtension* migration operation.

TrigramSimilarity

class TrigramSimilarity(*expression*, *string*, ***extra*)

Accepts a field name or expression, and a string or expression. Returns the trigram similarity between the two arguments.

Usage example:

```
>>> from django.contrib.postgres.search import TrigramSimilarity
>>> Author.objects.create(name='Katy Stevens')
>>> Author.objects.create(name='Stephen Keats')
>>> test = 'Katie Stephens'
>>> Author.objects.annotate(
...     similarity=TrigramSimilarity('name', test),
... ).filter(similarity__gt=0.3).order_by('-similarity')
[<Author: Katy Stevens>, <Author: Stephen Keats>]
```

TrigramDistance

class TrigramDistance(*expression*, *string*, ***extra*)

Accepts a field name or expression, and a string or expression. Returns the trigram distance between the two arguments.

Usage example:

```
>>> from django.contrib.postgres.search import TrigramDistance
>>> Author.objects.create(name='Katy Stevens')
>>> Author.objects.create(name='Stephen Keats')
>>> test = 'Katie Stephens'
>>> Author.objects.annotate(
...     distance=TrigramDistance('name', test),
... ).filter(distance__lte=0.7).order_by('distance')
[<Author: Katy Stevens>, <Author: Stephen Keats>]
```

Validators

KeysValidator

class KeysValidator(*keys*, *strict=False*, *messages=None*)

Validates that the given keys are contained in the value. If `strict` is `True`, then it also checks that there are no other keys present.

The `messages` passed should be a dict containing the keys `missing_keys` and/or `extra_keys`.

: Note that this checks only for the existence of a given key, not that the value of a key is non-empty.

Range validators

RangeMaxValueValidator

class RangeMaxValueValidator(*limit_value*, *message=None*)

Validates that the upper bound of the range is not greater than `limit_value`.

RangeMinValueValidator

class RangeMinValueValidator(*limit_value*, *message=None*)

Validates that the lower bound of the range is not less than the `limit_value`.

6.5.9 The redirects app

Django comes with an optional redirects application. It lets you store simple redirects in a database and handles the redirecting for you. It uses the HTTP response status code `301 Moved Permanently` by default.

Installation

To install the redirects app, follow these steps:

1. Ensure that the `django.contrib.sites` framework *is installed*.
2. Add `'django.contrib.redirects'` to your *INSTALLED_APPS* setting.

3. Add `'django.contrib.redirects.middleware.RedirectFallbackMiddleware'` to your *MIDDLEWARE* setting.
4. Run the command *manage.py migrate*.

How it works

`manage.py migrate` creates a `django_redirect` table in your database. This is a simple lookup table with `site_id`, `old_path` and `new_path` fields.

The *RedirectFallbackMiddleware* does all of the work. Each time any Django application raises a 404 error, this middleware checks the redirects database for the requested URL as a last resort. Specifically, it checks for a redirect with the given `old_path` with a site ID that corresponds to the *SITE_ID* setting.

- If it finds a match, and `new_path` is not empty, it redirects to `new_path` using a 301 ("Moved Permanently") redirect. You can subclass *RedirectFallbackMiddleware* and set *response_redirect_class* to *django.http.HttpResponseRedirect* to use a `302 Moved Temporarily` redirect instead.
- If it finds a match, and `new_path` is empty, it sends a 410 ("Gone") HTTP header and empty (content-less) response.
- If it doesn't find a match, the request continues to be processed as usual.

The middleware only gets activated for 404s – not for 500s or responses of any other status code.

Note that the order of *MIDDLEWARE* matters. Generally, you can put *RedirectFallbackMiddleware* at the end of the list, because it's a last resort.

For more on middleware, read the middleware docs.

How to add, change and delete redirects

Via the admin interface

If you've activated the automatic Django admin interface, you should see a "Redirects" section on the admin index page. Edit redirects as you edit any other object in the system.

Via the Python API

class `models.`**Redirect**

Redirects are represented by a standard Django model, which lives in django/contrib/redirects/models.py. You can access redirect objects via the Django database API.

Middleware

class `middleware.`**RedirectFallbackMiddleware**

You can change the *HttpResponse* classes used by the middleware by creating a subclass of *RedirectFallbackMiddleware* and overriding `response_gone_class` and/or `response_redirect_class`.

response_gone_class

The *HttpResponse* class used when a *Redirect* is not found for the requested path or has a blank `new_path` value.

Defaults to *HttpResponseGone*.

response_redirect_class
The *HttpResponse* class that handles the redirect.

Defaults to *HttpResponsePermanentRedirect*.

6.5.10 The sitemap framework

Django comes with a high-level sitemap-generating framework that makes creating sitemap XML files easy.

Overview

A sitemap is an XML file on your website that tells search-engine indexers how frequently your pages change and how "important" certain pages are in relation to other pages on your site. This information helps search engines index your site.

The Django sitemap framework automates the creation of this XML file by letting you express this information in Python code.

It works much like Django's syndication framework. To create a sitemap, just write a *Sitemap* class and point to it in your URLconf.

Installation

To install the sitemap app, follow these steps:

1. Add `'django.contrib.sitemaps'` to your *INSTALLED_APPS* setting.
2. Make sure your *TEMPLATES* setting contains a `DjangoTemplates` backend whose `APP_DIRS` options is set to `True`. It's in there by default, so you'll only need to change this if you've changed that setting.
3. Make sure you've installed the *sites framework*.

(Note: The sitemap application doesn't install any database tables. The only reason it needs to go into *INSTALLED_APPS* is so that the *Loader()* template loader can find the default templates.)

Initialization

views.**sitemap**(*request*, *sitemaps*, *section=None*, *template_name='sitemap.xml'*, *content_type='application/xml'*)

To activate sitemap generation on your Django site, add this line to your URLconf:

```
from django.contrib.sitemaps.views import sitemap

url(r'^sitemap\.xml$', sitemap, {'sitemaps': sitemaps},
    name='django.contrib.sitemaps.views.sitemap')
```

This tells Django to build a sitemap when a client accesses `/sitemap.xml`.

The name of the sitemap file is not important, but the location is. Search engines will only index links in your sitemap for the current URL level and below. For instance, if `sitemap.xml` lives in your root directory, it may reference any URL in your site. However, if your sitemap lives at `/content/sitemap.xml`, it may only reference URLs that begin with `/content/`.

The sitemap view takes an extra, required argument: `{'sitemaps': sitemaps}`. `sitemaps` should be a dictionary that maps a short section label (e.g., `blog` or `news`) to its *Sitemap* class (e.g., `BlogSitemap` or `NewsSitemap`). It may also map to an *instance* of a *Sitemap* class (e.g., `BlogSitemap(some_var)`).

`Sitemap` classes

A *`Sitemap`* class is a simple Python class that represents a "section" of entries in your sitemap. For example, one *`Sitemap`* class could represent all the entries of your Weblog, while another could represent all of the events in your events calendar.

In the simplest case, all these sections get lumped together into one `sitemap.xml`, but it's also possible to use the framework to generate a sitemap index that references individual sitemap files, one per section. (See *Creating a sitemap index* below.)

`Sitemap` classes must subclass `django.contrib.sitemaps.Sitemap`. They can live anywhere in your codebase.

A simple example

Let's assume you have a blog system, with an `Entry` model, and you want your sitemap to include all the links to your individual blog entries. Here's how your sitemap class might look:

```
from django.contrib.sitemaps import Sitemap
from blog.models import Entry

class BlogSitemap(Sitemap):
    changefreq = "never"
    priority = 0.5

    def items(self):
        return Entry.objects.filter(is_draft=False)

    def lastmod(self, obj):
        return obj.pub_date
```

Note:

- *`changefreq`* and *`priority`* are class attributes corresponding to `<changefreq>` and `<priority>` elements, respectively. They can be made callable as functions, as *`lastmod`* was in the example.
- *`items()`* is simply a method that returns a list of objects. The objects returned will get passed to any callable methods corresponding to a sitemap property (*`location`*, *`lastmod`*, *`changefreq`*, and *`priority`*).
- *`lastmod`* should return a `datetime`.
- There is no *`location`* method in this example, but you can provide it in order to specify the URL for your object. By default, *`location()`* calls `get_absolute_url()` on each object and returns the result.

`Sitemap` class reference

class `Sitemap`

A `Sitemap` class can define the following methods/attributes:

`items`

Required. A method that returns a list of objects. The framework doesn't care what *type* of objects they are; all that matters is that these objects get passed to the *`location()`*, *`lastmod()`*, *`changefreq()`* and *`priority()`* methods.

`location`

Optional. Either a method or attribute.

If it's a method, it should return the absolute path for a given object as returned by *`items()`*.

If it's an attribute, its value should be a string representing an absolute path to use for *every* object returned by *items()*.

In both cases, "absolute path" means a URL that doesn't include the protocol or domain. Examples:

- Good: `'/foo/bar/'`
- Bad: `'example.com/foo/bar/'`
- Bad: `'https://example.com/foo/bar/'`

If *location* isn't provided, the framework will call the `get_absolute_url()` method on each object as returned by *items()*.

To specify a protocol other than `'http'`, use *protocol*.

lastmod

Optional. Either a method or attribute.

If it's a method, it should take one argument – an object as returned by *items()* – and return that object's last-modified date/time as a `datetime`.

If it's an attribute, its value should be a `datetime` representing the last-modified date/time for *every* object returned by *items()*.

If all items in a sitemap have a *lastmod*, the sitemap generated by *views.sitemap()* will have a `Last-Modified` header equal to the latest `lastmod`. You can activate the *ConditionalGetMiddleware* to make Django respond appropriately to requests with an `If-Modified-Since` header which will prevent sending the sitemap if it hasn't changed.

changefreq

Optional. Either a method or attribute.

If it's a method, it should take one argument – an object as returned by *items()* – and return that object's change frequency as a string.

If it's an attribute, its value should be a string representing the change frequency of *every* object returned by *items()*.

Possible values for *changefreq*, whether you use a method or attribute, are:

- `'always'`
- `'hourly'`
- `'daily'`
- `'weekly'`
- `'monthly'`
- `'yearly'`
- `'never'`

priority

Optional. Either a method or attribute.

If it's a method, it should take one argument – an object as returned by *items()* – and return that object's priority as either a string or float.

If it's an attribute, its value should be either a string or float representing the priority of *every* object returned by *items()*.

Example values for *priority*: `0.4`, `1.0`. The default priority of a page is `0.5`. See the sitemaps.org documentation for more.

protocol
Optional.

This attribute defines the protocol (`'http'` or `'https'`) of the URLs in the sitemap. If it isn't set, the protocol with which the sitemap was requested is used. If the sitemap is built outside the context of a request, the default is `'http'`.

limit
Optional.

This attribute defines the maximum number of URLs included on each page of the sitemap. Its value should not exceed the default value of `50000`, which is the upper limit allowed in the Sitemaps protocol.

i18n
Optional.

A boolean attribute that defines if the URLs of this sitemap should be generated using all of your *LANGUAGES*. The default is `False`.

Shortcuts

The sitemap framework provides a convenience class for a common case:

class GenericSitemap
The *django.contrib.sitemaps.GenericSitemap* class allows you to create a sitemap by passing it a dictionary which has to contain at least a `queryset` entry. This queryset will be used to generate the items of the sitemap. It may also have a `date_field` entry that specifies a date field for objects retrieved from the `queryset`. This will be used for the *lastmod* attribute in the generated sitemap. You may also pass *priority* and *changefreq* keyword arguments to the *GenericSitemap* constructor to specify these attributes for all URLs.

Example

Here's an example of a URLconf using *GenericSitemap*:

```
from django.conf.urls import url
from django.contrib.sitemaps import GenericSitemap
from django.contrib.sitemaps.views import sitemap
from blog.models import Entry

info_dict = {
    'queryset': Entry.objects.all(),
    'date_field': 'pub_date',
}

urlpatterns = [
    # some generic view using info_dict
    # ...

    # the sitemap
    url(r'^sitemap\.xml$', sitemap,
        {'sitemaps': {'blog': GenericSitemap(info_dict, priority=0.6)}},
        name='django.contrib.sitemaps.views.sitemap'),
]
```

Sitemap for static views

Often you want the search engine crawlers to index views which are neither object detail pages nor flatpages. The solution is to explicitly list URL names for these views in `items` and call *`reverse()`* in the `location` method of the sitemap. For example:

```
# sitemaps.py
from django.contrib import sitemaps
from django.urls import reverse

class StaticViewSitemap(sitemaps.Sitemap):
    priority = 0.5
    changefreq = 'daily'

    def items(self):
        return ['main', 'about', 'license']

    def location(self, item):
        return reverse(item)

# urls.py
from django.conf.urls import url
from django.contrib.sitemaps.views import sitemap

from .sitemaps import StaticViewSitemap
from . import views

sitemaps = {
    'static': StaticViewSitemap,
}

urlpatterns = [
    url(r'^$', views.main, name='main'),
    url(r'^about/$', views.about, name='about'),
    url(r'^license/$', views.license, name='license'),
    # ...
    url(r'^sitemap\.xml$', sitemap, {'sitemaps': sitemaps},
        name='django.contrib.sitemaps.views.sitemap')
]
```

Creating a sitemap index

`views.`**`index`**(*request*, *sitemaps*, *template_name='sitemap_index.xml'*, *content_type='application/xml'*, *sitemap_url_name='django.contrib.sitemaps.views.sitemap'*)

The sitemap framework also has the ability to create a sitemap index that references individual sitemap files, one per each section defined in your `sitemaps` dictionary. The only differences in usage are:

- You use two views in your URLconf: *`django.contrib.sitemaps.views.index()`* and *`django.contrib.sitemaps.views.sitemap()`*.
- The *`django.contrib.sitemaps.views.sitemap()`* view should take a `section` keyword argument.

Here's what the relevant URLconf lines would look like for the example above:

```
from django.contrib.sitemaps import views

urlpatterns = [
```

```
    url(r'^sitemap\.xml$', views.index, {'sitemaps': sitemaps}),
    url(r'^sitemap-(?P<section>.+)\.xml$', views.sitemap, {'sitemaps': sitemaps},
        name='django.contrib.sitemaps.views.sitemap'),
]
```

This will automatically generate a `sitemap.xml` file that references both `sitemap-flatpages.xml` and `sitemap-blog.xml`. The *`Sitemap`* classes and the `sitemaps` dict don't change at all.

You should create an index file if one of your sitemaps has more than 50,000 URLs. In this case, Django will automatically paginate the sitemap, and the index will reflect that.

If you're not using the vanilla sitemap view – for example, if it's wrapped with a caching decorator – you must name your sitemap view and pass `sitemap_url_name` to the index view:

```
from django.contrib.sitemaps import views as sitemaps_views
from django.views.decorators.cache import cache_page

urlpatterns = [
    url(r'^sitemap\.xml$',
        cache_page(86400)(sitemaps_views.index),
        {'sitemaps': sitemaps, 'sitemap_url_name': 'sitemaps'}),
    url(r'^sitemap-(?P<section>.+)\.xml$',
        cache_page(86400)(sitemaps_views.sitemap),
        {'sitemaps': sitemaps}, name='sitemaps'),
]
```

Template customization

If you wish to use a different template for each sitemap or sitemap index available on your site, you may specify it by passing a `template_name` parameter to the `sitemap` and `index` views via the URLconf:

```
from django.contrib.sitemaps import views

urlpatterns = [
    url(r'^custom-sitemap\.xml$', views.index, {
        'sitemaps': sitemaps,
        'template_name': 'custom_sitemap.html'
    }),
    url(r'^custom-sitemap-(?P<section>.+)\.xml$', views.sitemap, {
        'sitemaps': sitemaps,
        'template_name': 'custom_sitemap.html'
    }, name='django.contrib.sitemaps.views.sitemap'),
]
```

These views return *`TemplateResponse`* instances which allow you to easily customize the response data before rendering. For more details, see the TemplateResponse documentation.

Context variables

When customizing the templates for the *`index()`* and *`sitemap()`* views, you can rely on the following context variables.

Index

The variable `sitemaps` is a list of absolute URLs to each of the sitemaps.

Sitemap

The variable `urlset` is a list of URLs that should appear in the sitemap. Each URL exposes attributes as defined in the *`Sitemap`* class:

- `changefreq`
- `item`
- `lastmod`
- `location`
- `priority`

The `item` attribute has been added for each URL to allow more flexible customization of the templates, such as Google news sitemaps. Assuming Sitemap's *`items()`* would return a list of items with `publication_data` and a `tags` field something like this would generate a Google News compatible sitemap:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset
  xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"
  xmlns:news="https://www.google.com/schemas/sitemap-news/0.9">
{% spaceless %}
{% for url in urlset %}
  <url>
    <loc>{{ url.location }}</loc>
    {% if url.lastmod %}<lastmod>{{ url.lastmod|date:"Y-m-d" }}</lastmod>{% endif %}
    {% if url.changefreq %}<changefreq>{{ url.changefreq }}</changefreq>{% endif %}
    {% if url.priority %}<priority>{{ url.priority }}</priority>{% endif %}
    <news:news>
      {% if url.item.publication_date %}<news:publication_date>{{ url.item.publication_date|date:"Y-m
      {% if url.item.tags %}<news:keywords>{{ url.item.tags }}</news:keywords>{% endif %}
    </news:news>
  </url>
{% endfor %}
{% endspaceless %}
</urlset>
```

Pinging Google

You may want to "ping" Google when your sitemap changes, to let it know to reindex your site. The sitemaps framework provides a function to do just that: *`django.contrib.sitemaps.ping_google()`*.

`ping_google``()`

`ping_google()` takes an optional argument, `sitemap_url`, which should be the absolute path to your site's sitemap (e.g., `'/sitemap.xml'`). If this argument isn't provided, *`ping_google()`* will attempt to figure out your sitemap by performing a reverse looking in your URLconf.

`ping_google()` raises the exception `django.contrib.sitemaps.SitemapNotFound` if it cannot determine your sitemap URL.

Register with Google first!

The *`ping_google()`* command only works if you have registered your site with Google Webmaster Tools.

One useful way to call *`ping_google()`* is from a model's `save()` method:

```
from django.contrib.sitemaps import ping_google

class Entry(models.Model):
    # ...
    def save(self, force_insert=False, force_update=False):
        super(Entry, self).save(force_insert, force_update)
        try:
            ping_google()
        except Exception:
            # Bare 'except' because we could get a variety
            # of HTTP-related exceptions.
            pass
```

A more efficient solution, however, would be to call *ping_google()* from a cron script, or some other scheduled task. The function makes an HTTP request to Google's servers, so you may not want to introduce that network overhead each time you call `save()`.

Pinging Google via `manage.py`

`django-admin ping_google [sitemap_url]`

Once the sitemaps application is added to your project, you may also ping Google using the `ping_google` management command:

```
python manage.py ping_google [/sitemap.xml]
```

6.5.11 The "sites" framework

Django comes with an optional "sites" framework. It's a hook for associating objects and functionality to particular websites, and it's a holding place for the domain names and "verbose" names of your Django-powered sites.

Use it if your single Django installation powers more than one site and you need to differentiate between those sites in some way.

The sites framework is mainly based on a simple model:

class `models.`**`Site`**

A model for storing the `domain` and `name` attributes of a website.

`domain`

The fully qualified domain name associated with the website. For example, `www.example.com`.

The `domain` field was set to be *unique*.

`name`

A human-readable "verbose" name for the website.

The *SITE_ID* setting specifies the database ID of the *Site* object associated with that particular settings file. If the setting is omitted, the *get_current_site()* function will try to get the current site by comparing the *domain* with the host name from the *request.get_host()* method.

How you use this is up to you, but Django uses it in a couple of ways automatically via simple conventions.

Example usage

Why would you use sites? It's best explained through examples.

Associating content with multiple sites

The Django-powered sites LJWorld.com and Lawrence.com are operated by the same news organization – the Lawrence Journal-World newspaper in Lawrence, Kansas. LJWorld.com focuses on news, while Lawrence.com focuses on local entertainment. But sometimes editors want to publish an article on *both* sites.

The naive way of solving the problem would be to require site producers to publish the same story twice: once for LJWorld.com and again for Lawrence.com. But that's inefficient for site producers, and it's redundant to store multiple copies of the same story in the database.

The better solution is simple: Both sites use the same article database, and an article is associated with one or more sites. In Django model terminology, that's represented by a *ManyToManyField* in the `Article` model:

```
from django.db import models
from django.contrib.sites.models import Site

class Article(models.Model):
    headline = models.CharField(max_length=200)
    # ...
    sites = models.ManyToManyField(Site)
```

This accomplishes several things quite nicely:

- It lets the site producers edit all content – on both sites – in a single interface (the Django admin).
- It means the same story doesn't have to be published twice in the database; it only has a single record in the database.
- It lets the site developers use the same Django view code for both sites. The view code that displays a given story just checks to make sure the requested story is on the current site. It looks something like this:

```
    from django.contrib.sites.shortcuts import get_current_site

    def article_detail(request, article_id):
        try:
            a = Article.objects.get(id=article_id, sites__id=get_current_site(request).id)
        except Article.DoesNotExist:
            raise Http404("Article does not exist on this site")
        # ...
```

Associating content with a single site

Similarly, you can associate a model to the *Site* model in a many-to-one relationship, using *ForeignKey*.

For example, if an article is only allowed on a single site, you'd use a model like this:

```
from django.db import models
from django.contrib.sites.models import Site

class Article(models.Model):
    headline = models.CharField(max_length=200)
    # ...
    site = models.ForeignKey(Site, on_delete=models.CASCADE)
```

This has the same benefits as described in the last section.

Hooking into the current site from views

You can use the sites framework in your Django views to do particular things based on the site in which the view is being called. For example:

```
from django.conf import settings

def my_view(request):
    if settings.SITE_ID == 3:
        # Do something.
        pass
    else:
        # Do something else.
        pass
```

Of course, it's ugly to hard-code the site IDs like that. This sort of hard-coding is best for hackish fixes that you need done quickly. The cleaner way of accomplishing the same thing is to check the current site's domain:

```
from django.contrib.sites.shortcuts import get_current_site

def my_view(request):
    current_site = get_current_site(request)
    if current_site.domain == 'foo.com':
        # Do something
        pass
    else:
        # Do something else.
        pass
```

This has also the advantage of checking if the sites framework is installed, and return a *RequestSite* instance if it is not.

If you don't have access to the request object, you can use the `get_current()` method of the *Site* model's manager. You should then ensure that your settings file does contain the *SITE_ID* setting. This example is equivalent to the previous one:

```
from django.contrib.sites.models import Site

def my_function_without_request():
    current_site = Site.objects.get_current()
    if current_site.domain == 'foo.com':
        # Do something
        pass
    else:
        # Do something else.
        pass
```

Getting the current domain for display

LJWorld.com and Lawrence.com both have email alert functionality, which lets readers sign up to get notifications when news happens. It's pretty basic: A reader signs up on a Web form and immediately gets an email saying, "Thanks for your subscription."

It'd be inefficient and redundant to implement this sign up processing code twice, so the sites use the same code behind the scenes. But the "thank you for signing up" notice needs to be different for each site. By using *Site* objects, we can abstract the "thank you" notice to use the values of the current site's *name* and *domain*.

Here's an example of what the form-handling view looks like:

```
from django.contrib.sites.shortcuts import get_current_site
from django.core.mail import send_mail

def register_for_newsletter(request):
    # Check form values, etc., and subscribe the user.
    # ...

    current_site = get_current_site(request)
    send_mail(
        'Thanks for subscribing to %s alerts' % current_site.name,
        'Thanks for your subscription. We appreciate it.\n\n-The %s team.' % (
            current_site.name,
        ),
        'editor@%s' % current_site.domain,
        [user.email],
    )

    # ...
```

On Lawrence.com, this email has the subject line "Thanks for subscribing to lawrence.com alerts." On LJWorld.com, the email has the subject "Thanks for subscribing to LJWorld.com alerts." Same goes for the email's message body.

Note that an even more flexible (but more heavyweight) way of doing this would be to use Django's template system. Assuming Lawrence.com and LJWorld.com have different template directories (*DIRS*), you could simply farm out to the template system like so:

```
from django.core.mail import send_mail
from django.template import loader, Context

def register_for_newsletter(request):
    # Check form values, etc., and subscribe the user.
    # ...

    subject = loader.get_template('alerts/subject.txt').render(Context({}))
    message = loader.get_template('alerts/message.txt').render(Context({}))
    send_mail(subject, message, 'editor@ljworld.com', [user.email])

    # ...
```

In this case, you'd have to create `subject.txt` and `message.txt` template files for both the LJWorld.com and Lawrence.com template directories. That gives you more flexibility, but it's also more complex.

It's a good idea to exploit the *Site* objects as much as possible, to remove unneeded complexity and redundancy.

Getting the current domain for full URLs

Django's `get_absolute_url()` convention is nice for getting your objects' URL without the domain name, but in some cases you might want to display the full URL – with `http://` and the domain and everything – for an object. To do this, you can use the sites framework. A simple example:

```
>>> from django.contrib.sites.models import Site
>>> obj = MyModel.objects.get(id=3)
>>> obj.get_absolute_url()
'/mymodel/objects/3/'
>>> Site.objects.get_current().domain
'example.com'
>>> 'https://%s%s' % (Site.objects.get_current().domain, obj.get_absolute_url())
'https://example.com/mymodel/objects/3/'
```

Enabling the sites framework

To enable the sites framework, follow these steps:

1. Add `'django.contrib.sites'` to your *INSTALLED_APPS* setting.
2. Define a *SITE_ID* setting:

```
SITE_ID = 1
```

3. Run *migrate*.

`django.contrib.sites` registers a *post_migrate* signal handler which creates a default site named `example.com` with the domain `example.com`. This site will also be created after Django creates the test database. To set the correct name and domain for your project, you can use a *data migration*.

In order to serve different sites in production, you'd create a separate settings file with each `SITE_ID` (perhaps importing from a common settings file to avoid duplicating shared settings) and then specify the appropriate *DJANGO_SETTINGS_MODULE* for each site.

Caching the current `Site` object

As the current site is stored in the database, each call to `Site.objects.get_current()` could result in a database query. But Django is a little cleverer than that: on the first request, the current site is cached, and any subsequent call returns the cached data instead of hitting the database.

If for any reason you want to force a database query, you can tell Django to clear the cache using `Site.objects.clear_cache()`:

```
# First call; current site fetched from database.
current_site = Site.objects.get_current()
# ...

# Second call; current site fetched from cache.
current_site = Site.objects.get_current()
# ...

# Force a database query for the third call.
Site.objects.clear_cache()
current_site = Site.objects.get_current()
```

The `CurrentSiteManager`

class `managers.`**`CurrentSiteManager`**

If *Site* plays a key role in your application, consider using the helpful *CurrentSiteManager* in your model(s). It's a model manager that automatically filters its queries to include only objects associated with the current *Site*.

Mandatory `SITE_ID`

The `CurrentSiteManager` is only usable when the *SITE_ID* setting is defined in your settings.

Use *CurrentSiteManager* by adding it to your model explicitly. For example:

```
from django.db import models
from django.contrib.sites.models import Site
from django.contrib.sites.managers import CurrentSiteManager
```

```
class Photo(models.Model):
    photo = models.FileField(upload_to='/home/photos')
    photographer_name = models.CharField(max_length=100)
    pub_date = models.DateField()
    site = models.ForeignKey(Site, on_delete=models.CASCADE)
    objects = models.Manager()
    on_site = CurrentSiteManager()
```

With this model, `Photo.objects.all()` will return all `Photo` objects in the database, but `Photo.on_site.all()` will return only the `Photo` objects associated with the current site, according to the *`SITE_ID`* setting.

Put another way, these two statements are equivalent:

```
Photo.objects.filter(site=settings.SITE_ID)
Photo.on_site.all()
```

How did *`CurrentSiteManager`* know which field of `Photo` was the *`Site`*? By default, *`CurrentSiteManager`* looks for a either a *`ForeignKey`* called `site` or a *`ManyToManyField`* called `sites` to filter on. If you use a field named something other than `site` or `sites` to identify which *`Site`* objects your object is related to, then you need to explicitly pass the custom field name as a parameter to *`CurrentSiteManager`* on your model. The following model, which has a field called `publish_on`, demonstrates this:

```
from django.db import models
from django.contrib.sites.models import Site
from django.contrib.sites.managers import CurrentSiteManager

class Photo(models.Model):
    photo = models.FileField(upload_to='/home/photos')
    photographer_name = models.CharField(max_length=100)
    pub_date = models.DateField()
    publish_on = models.ForeignKey(Site, on_delete=models.CASCADE)
    objects = models.Manager()
    on_site = CurrentSiteManager('publish_on')
```

If you attempt to use *`CurrentSiteManager`* and pass a field name that doesn't exist, Django will raise a `ValueError`.

Finally, note that you'll probably want to keep a normal (non-site-specific) `Manager` on your model, even if you use *`CurrentSiteManager`*. As explained in the manager documentation, if you define a manager manually, then Django won't create the automatic `objects = models.Manager()` manager for you. Also note that certain parts of Django – namely, the Django admin site and generic views – use whichever manager is defined *first* in the model, so if you want your admin site to have access to all objects (not just site-specific ones), put `objects = models.Manager()` in your model, before you define *`CurrentSiteManager`*.

Site middleware

If you often use this pattern:

```
from django.contrib.sites.models import Site

def my_view(request):
    site = Site.objects.get_current()
    ...
```

there is simple way to avoid repetitions. Add *`django.contrib.sites.middleware.CurrentSiteMiddleware`* to *`MIDDLEWARE`*. The middleware sets the `site` attribute on every request object, so you can use `request.site`

to get the current site.

How Django uses the sites framework

Although it's not required that you use the sites framework, it's strongly encouraged, because Django takes advantage of it in a few places. Even if your Django installation is powering only a single site, you should take the two seconds to create the site object with your `domain` and `name`, and point to its ID in your *`SITE_ID`* setting.

Here's how Django uses the sites framework:

- In the *`redirects framework`*, each redirect object is associated with a particular site. When Django searches for a redirect, it takes into account the current site.
- In the *`flatpages framework`*, each flatpage is associated with a particular site. When a flatpage is created, you specify its *`Site`*, and the *`FlatpageFallbackMiddleware`* checks the current site in retrieving flatpages to display.
- In the *`syndication framework`*, the templates for `title` and `description` automatically have access to a variable `{{ site }}`, which is the *`Site`* object representing the current site. Also, the hook for providing item URLs will use the `domain` from the current *`Site`* object if you don't specify a fully-qualified domain.
- In the *`authentication framework`*, the *`django.contrib.auth.views.login()`* view passes the current *`Site`* name to the template as `{{ site_name }}`.
- The shortcut view (`django.contrib.contenttypes.views.shortcut`) uses the domain of the current *`Site`* object when calculating an object's URL.
- In the admin framework, the "view on site" link uses the current *`Site`* to work out the domain for the site that it will redirect to.

`RequestSite` objects

Some django.contrib applications take advantage of the sites framework but are architected in a way that doesn't *require* the sites framework to be installed in your database. (Some people don't want to, or just aren't *able* to install the extra database table that the sites framework requires.) For those cases, the framework provides a *`django.contrib.sites.requests.RequestSite`* class, which can be used as a fallback when the database-backed sites framework is not available.

class `requests.`**`RequestSite`**
A class that shares the primary interface of *`Site`* (i.e., it has `domain` and `name` attributes) but gets its data from a Django *`HttpRequest`* object rather than from a database.

`__init__`(*request*)
Sets the `name` and `domain` attributes to the value of *`get_host()`*.

A *`RequestSite`* object has a similar interface to a normal *`Site`* object, except its *`__init__()`* method takes an *`HttpRequest`* object. It's able to deduce the `domain` and `name` by looking at the request's domain. It has `save()` and `delete()` methods to match the interface of *`Site`*, but the methods raise `NotImplementedError`.

`get_current_site` shortcut

Finally, to avoid repetitive fallback code, the framework provides a *`django.contrib.sites.shortcuts.get_current_sit`* function.

`shortcuts.`**`get_current_site`**(*request*)
A function that checks if `django.contrib.sites` is installed and returns either the current *`Site`* object or

a *RequestSite* object based on the request. It looks up the current site based on *request.get_host()* if the *SITE_ID* setting is not defined.

Both a domain and a port may be returned by *request.get_host()* when the Host header has a port explicitly specified, e.g. `example.com:80`. In such cases, if the lookup fails because the host does not match a record in the database, the port is stripped and the lookup is retried with the domain part only. This does not apply to *RequestSite* which will always use the unmodified host.

Retrying the lookup with the port stripped was added.

6.5.12 The `staticfiles` app

`django.contrib.staticfiles` collects static files from each of your applications (and any other places you specify) into a single location that can easily be served in production.

:

For an introduction to the static files app and some usage examples, see Managing static files (e.g. images, JavaScript, CSS). For guidelines on deploying static files, see Deploying static files.

Settings

See *staticfiles settings* for details on the following settings:

- *STATIC_ROOT*
- *STATIC_URL*
- *STATICFILES_DIRS*
- *STATICFILES_STORAGE*
- *STATICFILES_FINDERS*

Management Commands

`django.contrib.staticfiles` exposes three management commands.

`collectstatic`

`django-admin collectstatic`

Collects the static files into *STATIC_ROOT*.

Duplicate file names are by default resolved in a similar way to how template resolution works: the file that is first found in one of the specified locations will be used. If you're confused, the *findstatic* command can help show you which files are found.

On subsequent `collectstatic` runs (if `STATIC_ROOT` isn't empty), files are copied only if they have a modified timestamp greater than the timestamp of the file in `STATIC_ROOT`. Therefore if you remove an application from *INSTALLED_APPS*, it's a good idea to use the *collectstatic --clear* option in order to remove stale static files.

Files are searched by using the *enabled finders*. The default is to look in all locations defined in *STATICFILES_DIRS* and in the `'static'` directory of apps specified by the *INSTALLED_APPS* setting.

The *collectstatic* management command calls the *post_process()* method of the *STATICFILES_STORAGE* after each run and passes a list of paths that have been found by the management command. It also receives all command line options of *collectstatic*. This is used by the *CachedStaticFilesStorage* by default.

By default, collected files receive permissions from *FILE_UPLOAD_PERMISSIONS* and collected directories receive permissions from *FILE_UPLOAD_DIRECTORY_PERMISSIONS*. If you would like different permissions for these files and/or directories, you can subclass either of the *static files storage classes* and specify the `file_permissions_mode` and/or `directory_permissions_mode` parameters, respectively. For example:

```
from django.contrib.staticfiles import storage

class MyStaticFilesStorage(storage.StaticFilesStorage):
    def __init__(self, *args, **kwargs):
        kwargs['file_permissions_mode'] = 0o640
        kwargs['directory_permissions_mode'] = 0o760
        super(MyStaticFilesStorage, self).__init__(*args, **kwargs)
```

Then set the *STATICFILES_STORAGE* setting to `'path.to.MyStaticFilesStorage'`.

Some commonly used options are:

`--noinput, --no-input`
Do NOT prompt the user for input of any kind.

The `--no-input` alias was added.

`--ignore` `PATTERN`**`, -i`** `PATTERN`
Ignore files or directories matching this glob-style pattern. Use multiple times to ignore more.

`--dry-run, -n`
Do everything except modify the filesystem.

`--clear, -c`
Clear the existing files before trying to copy or link the original file.

`--link, -l`
Create a symbolic link to each file instead of copying.

`--no-post-process`
Don't call the *post_process()* method of the configured *STATICFILES_STORAGE* storage backend.

`--no-default-ignore`
Don't ignore the common private glob-style patterns `'CVS'`, `'.*'` and `'*~'`.

For a full list of options, refer to the commands own help by running:

```
$ python manage.py collectstatic --help
```

Customizing the ignored pattern list The default ignored pattern list, `['CVS', '.*', '*~']`, can be customized in a more persistent way than providing the `--ignore` command option at each `collectstatic` invocation. Provide a custom *AppConfig* class, override the `ignore_patterns` attribute of this class and replace `'django.contrib.staticfiles'` with that class path in your *INSTALLED_APPS* setting:

```
from django.contrib.staticfiles.apps import StaticFilesConfig

class MyStaticFilesConfig(StaticFilesConfig):
    ignore_patterns = [...]  # your custom ignore list
```

findstatic

django-admin findstatic staticfile [staticfile ...]

Searches for one or more relative paths with the enabled finders.

For example:

```
$ python manage.py findstatic css/base.css admin/js/core.js
Found 'css/base.css' here:
  /home/special.polls.com/core/static/css/base.css
  /home/polls.com/core/static/css/base.css
Found 'admin/js/core.js' here:
  /home/polls.com/src/django/contrib/admin/media/js/core.js
```

findstatic `-first`

By default, all matching locations are found. To only return the first match for each relative path, use the `--first` option:

```
$ python manage.py findstatic css/base.css --first
Found 'css/base.css' here:
  /home/special.polls.com/core/static/css/base.css
```

This is a debugging aid; it'll show you exactly which static file will be collected for a given path.

By setting the `--verbosity` flag to 0, you can suppress the extra output and just get the path names:

```
$ python manage.py findstatic css/base.css --verbosity 0
/home/special.polls.com/core/static/css/base.css
/home/polls.com/core/static/css/base.css
```

On the other hand, by setting the `--verbosity` flag to 2, you can get all the directories which were searched:

```
$ python manage.py findstatic css/base.css --verbosity 2
Found 'css/base.css' here:
  /home/special.polls.com/core/static/css/base.css
  /home/polls.com/core/static/css/base.css
Looking in the following locations:
  /home/special.polls.com/core/static
  /home/polls.com/core/static
  /some/other/path/static
```

runserver

django-admin runserver [addrport]

Overrides the core *runserver* command if the `staticfiles` app is *installed* and adds automatic serving of static files and the following new options.

--nostatic

Use the `--nostatic` option to disable serving of static files with the staticfiles app entirely. This option is only available if the staticfiles app is in your project's *INSTALLED_APPS* setting.

Example usage:

```
django-admin runserver --nostatic
```

--insecure

Use the `--insecure` option to force serving of static files with the staticfiles app even if the *DEBUG* setting is `False`. By using this you acknowledge the fact that it's **grossly inefficient** and probably **insecure**. This is only intended for local development, should **never be used in production** and is only available if the staticfiles app is in your project's *INSTALLED_APPS* setting. *runserver* `--insecure` doesn't work with *CachedStaticFilesStorage*.

Example usage:

```
django-admin runserver --insecure
```

Storages

StaticFilesStorage

class `storage.`**`StaticFilesStorage`**

A subclass of the *FileSystemStorage* storage backend that uses the *STATIC_ROOT* setting as the base file system location and the *STATIC_URL* setting respectively as the base URL.

`storage.StaticFilesStorage.`**`post_process`**(*paths*, ***options*)

This method is called by the *collectstatic* management command after each run and gets passed the local storages and paths of found files as a dictionary, as well as the command line options.

The *CachedStaticFilesStorage* uses this behind the scenes to replace the paths with their hashed counterparts and update the cache appropriately.

ManifestStaticFilesStorage

class `storage.`**`ManifestStaticFilesStorage`**

A subclass of the *StaticFilesStorage* storage backend which stores the file names it handles by appending the MD5 hash of the file's content to the filename. For example, the file `css/styles.css` would also be saved as `css/styles.55e7cbb9ba48.css`.

The purpose of this storage is to keep serving the old files in case some pages still refer to those files, e.g. because they are cached by you or a 3rd party proxy server. Additionally, it's very helpful if you want to apply far future Expires headers to the deployed files to speed up the load time for subsequent page visits.

The storage backend automatically replaces the paths found in the saved files matching other saved files with the path of the cached copy (using the *post_process()* method). The regular expressions used to find those paths (`django.contrib.staticfiles.storage.HashedFilesMixin.patterns`) by default covers the @import rule and url() statement of Cascading Style Sheets. For example, the `'css/styles.css'` file with the content

```
@import url("../admin/css/base.css");
```

would be replaced by calling the *url()* method of the `ManifestStaticFilesStorage` storage backend, ultimately saving a `'css/styles.55e7cbb9ba48.css'` file with the following content:

```
@import url("../admin/css/base.27e20196a850.css");
```

To enable the `ManifestStaticFilesStorage` you have to make sure the following requirements are met:

- the *STATICFILES_STORAGE* setting is set to `'django.contrib.staticfiles.storage.ManifestStaticFiles`
- the *DEBUG* setting is set to `False`
- you've collected all your static files by using the *collectstatic* management command

In older versions, you also had to use `{% load static from staticfiles %}` in your template. The *static* template tag (`{% load static %}`) now uses *django.contrib.staticfiles* if it's installed.

Since creating the MD5 hash can be a performance burden to your website during runtime, `staticfiles` will automatically store the mapping with hashed names for all processed files in a file called `staticfiles.json`. This happens once when you run the *collectstatic* management command.

Due to the requirement of running *collectstatic*, this storage typically shouldn't be used when running tests as `collectstatic` isn't run as part of the normal test setup. During testing, ensure that the *STATICFILES_STORAGE* setting is set to something else like `'django.contrib.staticfiles.storage.StaticFilesStorage'` (the default).

`storage.ManifestStaticFilesStorage.`**`file_hash`**(*name*, *content=None*)

The method that is used when creating the hashed name of a file. Needs to return a hash for the given file name and content. By default it calculates a MD5 hash from the content's chunks as mentioned above. Feel free to override this method to use your own hashing algorithm.

CachedStaticFilesStorage

class `storage.`**`CachedStaticFilesStorage`**

`CachedStaticFilesStorage` is a similar class like the *ManifestStaticFilesStorage* class but uses Django's caching framework for storing the hashed names of processed files instead of a static manifest file called `staticfiles.json`. This is mostly useful for situations in which you don't have access to the file system.

If you want to override certain options of the cache backend the storage uses, simply specify a custom entry in the *CACHES* setting named `'staticfiles'`. It falls back to using the `'default'` cache backend.

Finders Module

`staticfiles` finders has a `searched_locations` attribute which is a list of directory paths in which the finders searched. Example usage:

```
from django.contrib.staticfiles import finders

result = finders.find('css/base.css')
searched_locations = finders.searched_locations
```

Other Helpers

There are a few other helpers outside of the *staticfiles* app to work with static files:

- The *django.template.context_processors.static()* context processor which adds *STATIC_URL* to every template context rendered with *RequestContext* contexts.
- The builtin template tag *static* which takes a path and urljoins it with the static prefix *STATIC_URL*. If `django.contrib.staticfiles` is installed, the tag uses the `url()` method of the *STATICFILES_STORAGE* instead.
- The builtin template tag *get_static_prefix* which populates a template variable with the static prefix *STATIC_URL* to be used as a variable or directly.
- The similar template tag *get_media_prefix* which works like *get_static_prefix* but uses *MEDIA_URL*.

Static file development view

The static files tools are mostly designed to help with getting static files successfully deployed into production. This usually means a separate, dedicated static file server, which is a lot of overhead to mess with when developing locally. Thus, the `staticfiles` app ships with a **quick and dirty helper view** that you can use to serve files locally in development.

`views.`**`serve`**(*request*, *path*)

This view function serves static files in development.

: This view will only work if *DEBUG* is `True`.
That's because this view is **grossly inefficient** and probably **insecure**. This is only intended for local development, and should **never be used in production**.

: To guess the served files' content types, this view relies on the `mimetypes` module from the Python standard library, which itself relies on the underlying platform's map files. If you find that this view doesn't return proper content types for certain files, it is most likely that the platform's map files need to be updated. This can be achieved, for example, by installing or updating the `mailcap` package on a Red Hat distribution, or `mime-support` on a Debian distribution.

This view is automatically enabled by *runserver* (with a *DEBUG* setting set to `True`). To use the view with a different local development server, add the following snippet to the end of your primary URL configuration:

```
from django.conf import settings
from django.contrib.staticfiles import views

if settings.DEBUG:
    urlpatterns += [
        url(r'^static/(?P<path>.*)$', views.serve),
    ]
```

Note, the beginning of the pattern (`r'^static/'`) should be your *STATIC_URL* setting.

Since this is a bit finicky, there's also a helper function that'll do this for you:

`urls.`**`staticfiles_urlpatterns`**`()`

This will return the proper URL pattern for serving static files to your already defined pattern list. Use it like this:

```
from django.contrib.staticfiles.urls import staticfiles_urlpatterns

# ... the rest of your URLconf here ...

urlpatterns += staticfiles_urlpatterns()
```

This will inspect your *STATIC_URL* setting and wire up the view to serve static files accordingly. Don't forget to set the *STATICFILES_DIRS* setting appropriately to let `django.contrib.staticfiles` know where to look for files in addition to files in app directories.

: This helper function will only work if *DEBUG* is `True` and your *STATIC_URL* setting is neither empty nor a full URL such as `http://static.example.com/`.
That's because this view is **grossly inefficient** and probably **insecure**. This is only intended for local development, and should **never be used in production**.

Specialized test case to support 'live testing'

class `testing.`**`StaticLiveServerTestCase`**

This unittest TestCase subclass extends *`django.test.LiveServerTestCase`*.

Just like its parent, you can use it to write tests that involve running the code under test and consuming it with testing tools through HTTP (e.g. Selenium, PhantomJS, etc.), because of which it's needed that the static assets are also published.

But given the fact that it makes use of the *`django.contrib.staticfiles.views.serve()`* view described above, it can transparently overlay at test execution-time the assets provided by the `staticfiles` finders. This means you don't need to run *`collectstatic`* before or as a part of your tests setup.

6.5.13 The syndication feed framework

Django comes with a high-level syndication-feed-generating framework that makes creating RSS and Atom feeds easy.

To create any syndication feed, all you have to do is write a short Python class. You can create as many feeds as you want.

Django also comes with a lower-level feed-generating API. Use this if you want to generate feeds outside of a Web context, or in some other lower-level way.

The high-level framework

Overview

The high-level feed-generating framework is supplied by the *`Feed`* class. To create a feed, write a *`Feed`* class and point to an instance of it in your URLconf.

`Feed` classes

A *`Feed`* class is a Python class that represents a syndication feed. A feed can be simple (e.g., a "site news" feed, or a basic feed displaying the latest entries of a blog) or more complex (e.g., a feed displaying all the blog entries in a particular category, where the category is variable).

Feed classes subclass *`django.contrib.syndication.views.Feed`*. They can live anywhere in your codebase.

Instances of *`Feed`* classes are views which can be used in your URLconf.

A simple example

This simple example, taken from a hypothetical police beat news site describes a feed of the latest five news items:

```
from django.contrib.syndication.views import Feed
from django.urls import reverse
from policebeat.models import NewsItem

class LatestEntriesFeed(Feed):
    title = "Police beat site news"
    link = "/sitenews/"
```

```
    description = "Updates on changes and additions to police beat central."

    def items(self):
        return NewsItem.objects.order_by('-pub_date')[:5]

    def item_title(self, item):
        return item.title

    def item_description(self, item):
        return item.description

    # item_link is only needed if NewsItem has no get_absolute_url method.
    def item_link(self, item):
        return reverse('news-item', args=[item.pk])
```

To connect a URL to this feed, put an instance of the Feed object in your URLconf. For example:

```
from django.conf.urls import url
from myproject.feeds import LatestEntriesFeed

urlpatterns = [
    # ...
    url(r'^latest/feed/$', LatestEntriesFeed()),
    # ...
]
```

Note:

- The Feed class subclasses *django.contrib.syndication.views.Feed*.
- `title`, `link` and `description` correspond to the standard RSS `<title>`, `<link>` and `<description>` elements, respectively.
- `items()` is, simply, a method that returns a list of objects that should be included in the feed as `<item>` elements. Although this example returns `NewsItem` objects using Django's object-relational mapper, `items()` doesn't have to return model instances. Although you get a few bits of functionality "for free" by using Django models, `items()` can return any type of object you want.
- If you're creating an Atom feed, rather than an RSS feed, set the `subtitle` attribute instead of the `description` attribute. See *Publishing Atom and RSS feeds in tandem*, later, for an example.

One thing is left to do. In an RSS feed, each `<item>` has a `<title>`, `<link>` and `<description>`. We need to tell the framework what data to put into those elements.

- For the contents of `<title>` and `<description>`, Django tries calling the methods `item_title()` and `item_description()` on the *Feed* class. They are passed a single parameter, `item`, which is the object itself. These are optional; by default, the unicode representation of the object is used for both.

  If you want to do any special formatting for either the title or description, Django templates can be used instead. Their paths can be specified with the `title_template` and `description_template` attributes on the *Feed* class. The templates are rendered for each item and are passed two template context variables:

  - `{{ obj }}` – The current object (one of whichever objects you returned in `items()`).
  - `{{ site }}` – A *django.contrib.sites.models.Site* object representing the current site. This is useful for `{{ site.domain }}` or `{{ site.name }}`. If you do *not* have the Django sites framework installed, this will be set to a *RequestSite* object. See the *RequestSite section of the sites framework documentation* for more.

  See *a complex example* below that uses a description template.

Feed.**get_context_data**(***kwargs*)

There is also a way to pass additional information to title and description templates, if you need to supply more than the two variables mentioned before. You can provide your implementation of `get_context_data` method in your `Feed` subclass. For example:

```
from mysite.models import Article
from django.contrib.syndication.views import Feed

class ArticlesFeed(Feed):
    title = "My articles"
    description_template = "feeds/articles.html"

    def items(self):
        return Article.objects.order_by('-pub_date')[:5]

    def get_context_data(self, **kwargs):
        context = super(ArticlesFeed, self).get_context_data(**kwargs)
        context['foo'] = 'bar'
        return context
```

And the template:

```
Something about {{ foo }}: {{ obj.description }}
```

This method will be called once per each item in the list returned by `items()` with the following keyword arguments:

- `item`: the current item. For backward compatibility reasons, the name of this context variable is `{{ obj }}`.
- `obj`: the object returned by `get_object()`. By default this is not exposed to the templates to avoid confusion with `{{ obj }}` (see above), but you can use it in your implementation of `get_context_data()`.
- `site`: current site as described above.
- `request`: current request.

The behavior of `get_context_data()` mimics that of *generic views* - you're supposed to call `super()` to retrieve context data from parent class, add your data and return the modified dictionary.

- To specify the contents of `<link>`, you have two options. For each item in `items()`, Django first tries calling the `item_link()` method on the *`Feed`* class. In a similar way to the title and description, it is passed it a single parameter, `item`. If that method doesn't exist, Django tries executing a `get_absolute_url()` method on that object. Both `get_absolute_url()` and `item_link()` should return the item's URL as a normal Python string. As with `get_absolute_url()`, the result of `item_link()` will be included directly in the URL, so you are responsible for doing all necessary URL quoting and conversion to ASCII inside the method itself.

A complex example

The framework also supports more complex feeds, via arguments.

For example, a website could offer an RSS feed of recent crimes for every police beat in a city. It'd be silly to create a separate *`Feed`* class for each police beat; that would violate the *DRY principle* and would couple data to programming logic. Instead, the syndication framework lets you access the arguments passed from your URLconf so feeds can output items based on information in the feed's URL.

The police beat feeds could be accessible via URLs like this:

- `/beats/613/rss/` – Returns recent crimes for beat 613.
- `/beats/1424/rss/` – Returns recent crimes for beat 1424.

These can be matched with a URLconf line such as:

```
url(r'^beats/(?P<beat_id>[0-9]+)/rss/$', BeatFeed()),
```

Like a view, the arguments in the URL are passed to the `get_object()` method along with the request object.

Here's the code for these beat-specific feeds:

```
from django.contrib.syndication.views import Feed

class BeatFeed(Feed):
    description_template = 'feeds/beat_description.html'

    def get_object(self, request, beat_id):
        return Beat.objects.get(pk=beat_id)

    def title(self, obj):
        return "Police beat central: Crimes for beat %s" % obj.beat

    def link(self, obj):
        return obj.get_absolute_url()

    def description(self, obj):
        return "Crimes recently reported in police beat %s" % obj.beat

    def items(self, obj):
        return Crime.objects.filter(beat=obj).order_by('-crime_date')[:30]
```

To generate the feed's `<title>`, `<link>` and `<description>`, Django uses the `title()`, `link()` and `description()` methods. In the previous example, they were simple string class attributes, but this example illustrates that they can be either strings *or* methods. For each of `title`, `link` and `description`, Django follows this algorithm:

- First, it tries to call a method, passing the `obj` argument, where `obj` is the object returned by `get_object()`.
- Failing that, it tries to call a method with no arguments.
- Failing that, it uses the class attribute.

Also note that `items()` also follows the same algorithm – first, it tries `items(obj)`, then `items()`, then finally an `items` class attribute (which should be a list).

We are using a template for the item descriptions. It can be very simple:

```
{{ obj.description }}
```

However, you are free to add formatting as desired.

The `ExampleFeed` class below gives full documentation on methods and attributes of *`Feed`* classes.

Specifying the type of feed

By default, feeds produced in this framework use RSS 2.0.

To change that, add a `feed_type` attribute to your *`Feed`* class, like so:

```
from django.utils.feedgenerator import Atom1Feed

class MyFeed(Feed):
    feed_type = Atom1Feed
```

Note that you set `feed_type` to a class object, not an instance.

Currently available feed types are:

- *django.utils.feedgenerator.Rss201rev2Feed* (RSS 2.01. Default.)
- *django.utils.feedgenerator.RssUserland091Feed* (RSS 0.91.)
- *django.utils.feedgenerator.Atom1Feed* (Atom 1.0.)

Enclosures

To specify enclosures, such as those used in creating podcast feeds, use the `item_enclosures` hook or, alternatively and if you only have a single enclosure per item, the `item_enclosure_url`, `item_enclosure_length`, and `item_enclosure_mime_type` hooks. See the `ExampleFeed` class below for usage examples.

Support for multiple enclosures per feed item was added through the `item_enclosures` hook.

Language

Feeds created by the syndication framework automatically include the appropriate `<language>` tag (RSS 2.0) or `xml:lang` attribute (Atom). This comes directly from your *LANGUAGE_CODE* setting.

URLs

The `link` method/attribute can return either an absolute path (e.g. `"/blog/"`) or a URL with the fully-qualified domain and protocol (e.g. `"https://www.example.com/blog/"`). If `link` doesn't return the domain, the syndication framework will insert the domain of the current site, according to your *SITE_ID setting*.

Atom feeds require a `<link rel="self">` that defines the feed's current location. The syndication framework populates this automatically, using the domain of the current site according to the *SITE_ID* setting.

Publishing Atom and RSS feeds in tandem

Some developers like to make available both Atom *and* RSS versions of their feeds. That's easy to do with Django: Just create a subclass of your *Feed* class and set the `feed_type` to something different. Then update your URLconf to add the extra versions.

Here's a full example:

```
from django.contrib.syndication.views import Feed
from policebeat.models import NewsItem
from django.utils.feedgenerator import Atom1Feed

class RssSiteNewsFeed(Feed):
    title = "Police beat site news"
    link = "/sitenews/"
    description = "Updates on changes and additions to police beat central."
```

```
    def items(self):
        return NewsItem.objects.order_by('-pub_date')[:5]

class AtomSiteNewsFeed(RssSiteNewsFeed):
    feed_type = Atom1Feed
    subtitle = RssSiteNewsFeed.description
```

: In this example, the RSS feed uses a `description` while the Atom feed uses a `subtitle`. That's because Atom feeds don't provide for a feed-level "description," but they *do* provide for a "subtitle."

If you provide a `description` in your *`Feed`* class, Django will *not* automatically put that into the `subtitle` element, because a subtitle and description are not necessarily the same thing. Instead, you should define a `subtitle` attribute.

In the above example, we simply set the Atom feed's `subtitle` to the RSS feed's `description`, because it's quite short already.

And the accompanying URLconf:

```
from django.conf.urls import url
from myproject.feeds import RssSiteNewsFeed, AtomSiteNewsFeed

urlpatterns = [
    # ...
    url(r'^sitenews/rss/$', RssSiteNewsFeed()),
    url(r'^sitenews/atom/$', AtomSiteNewsFeed()),
    # ...
]
```

`Feed` class reference

class `views.`**`Feed`**

This example illustrates all possible attributes and methods for a *`Feed`* class:

```
from django.contrib.syndication.views import Feed
from django.utils import feedgenerator

class ExampleFeed(Feed):

    # FEED TYPE -- Optional. This should be a class that subclasses
    # django.utils.feedgenerator.SyndicationFeed. This designates
    # which type of feed this should be: RSS 2.0, Atom 1.0, etc. If
    # you don't specify feed_type, your feed will be RSS 2.0. This
    # should be a class, not an instance of the class.

    feed_type = feedgenerator.Rss201rev2Feed

    # TEMPLATE NAMES -- Optional. These should be strings
    # representing names of Django templates that the system should
    # use in rendering the title and description of your feed items.
    # Both are optional. If a template is not specified, the
    # item_title() or item_description() methods are used instead.

    title_template = None
    description_template = None
```

```
    # TITLE -- One of the following three is required. The framework
    # looks for them in this order.

    def title(self, obj):
        """
        Takes the object returned by get_object() and returns the
        feed's title as a normal Python string.
        """

    def title(self):
        """
        Returns the feed's title as a normal Python string.
        """

    title = 'foo' # Hard-coded title.

    # LINK -- One of the following three is required. The framework
    # looks for them in this order.

    def link(self, obj):
        """
        # Takes the object returned by get_object() and returns the URL
        # of the HTML version of the feed as a normal Python string.
        """

    def link(self):
        """
        Returns the URL of the HTML version of the feed as a normal Python
        string.
        """

    link = '/blog/' # Hard-coded URL.

    # FEED_URL -- One of the following three is optional. The framework
    # looks for them in this order.

    def feed_url(self, obj):
        """
        # Takes the object returned by get_object() and returns the feed's
        # own URL as a normal Python string.
        """

    def feed_url(self):
        """
        Returns the feed's own URL as a normal Python string.
        """

    feed_url = '/blog/rss/' # Hard-coded URL.

    # GUID -- One of the following three is optional. The framework looks
    # for them in this order. This property is only used for Atom feeds
    # (where it is the feed-level ID element). If not provided, the feed
    # link is used as the ID.

    def feed_guid(self, obj):
        """
        Takes the object returned by get_object() and returns the globally
        unique ID for the feed as a normal Python string.
```

```
        """

    def feed_guid(self):
        """
        Returns the feed's globally unique ID as a normal Python string.
        """

    feed_guid = '/foo/bar/1234' # Hard-coded guid.

    # DESCRIPTION -- One of the following three is required. The framework
    # looks for them in this order.

    def description(self, obj):
        """
        Takes the object returned by get_object() and returns the feed's
        description as a normal Python string.
        """

    def description(self):
        """
        Returns the feed's description as a normal Python string.
        """

    description = 'Foo bar baz.' # Hard-coded description.

    # AUTHOR NAME --One of the following three is optional. The framework
    # looks for them in this order.

    def author_name(self, obj):
        """
        Takes the object returned by get_object() and returns the feed's
        author's name as a normal Python string.
        """

    def author_name(self):
        """
        Returns the feed's author's name as a normal Python string.
        """

    author_name = 'Sally Smith' # Hard-coded author name.

    # AUTHOR EMAIL --One of the following three is optional. The framework
    # looks for them in this order.

    def author_email(self, obj):
        """
        Takes the object returned by get_object() and returns the feed's
        author's email as a normal Python string.
        """

    def author_email(self):
        """
        Returns the feed's author's email as a normal Python string.
        """

    author_email = 'test@example.com' # Hard-coded author email.

    # AUTHOR LINK --One of the following three is optional. The framework
```

```
    # looks for them in this order. In each case, the URL should include
    # the "http://" and domain name.

    def author_link(self, obj):
        """
        Takes the object returned by get_object() and returns the feed's
        author's URL as a normal Python string.
        """

    def author_link(self):
        """
        Returns the feed's author's URL as a normal Python string.
        """

    author_link = 'https://www.example.com/' # Hard-coded author URL.

    # CATEGORIES -- One of the following three is optional. The framework
    # looks for them in this order. In each case, the method/attribute
    # should return an iterable object that returns strings.

    def categories(self, obj):
        """
        Takes the object returned by get_object() and returns the feed's
        categories as iterable over strings.
        """

    def categories(self):
        """
        Returns the feed's categories as iterable over strings.
        """

    categories = ("python", "django") # Hard-coded list of categories.

    # COPYRIGHT NOTICE -- One of the following three is optional. The
    # framework looks for them in this order.

    def feed_copyright(self, obj):
        """
        Takes the object returned by get_object() and returns the feed's
        copyright notice as a normal Python string.
        """

    def feed_copyright(self):
        """
        Returns the feed's copyright notice as a normal Python string.
        """

    feed_copyright = 'Copyright (c) 2007, Sally Smith' # Hard-coded copyright notice.

    # TTL -- One of the following three is optional. The framework looks
    # for them in this order. Ignored for Atom feeds.

    def ttl(self, obj):
        """
        Takes the object returned by get_object() and returns the feed's
        TTL (Time To Live) as a normal Python string.
        """
```

```
    def ttl(self):
        """
        Returns the feed's TTL as a normal Python string.
        """

    ttl = 600 # Hard-coded Time To Live.

    # ITEMS -- One of the following three is required. The framework looks
    # for them in this order.

    def items(self, obj):
        """
        Takes the object returned by get_object() and returns a list of
        items to publish in this feed.
        """

    def items(self):
        """
        Returns a list of items to publish in this feed.
        """

    items = ('Item 1', 'Item 2') # Hard-coded items.

    # GET_OBJECT -- This is required for feeds that publish different data
    # for different URL parameters. (See "A complex example" above.)

    def get_object(self, request, *args, **kwargs):
        """
        Takes the current request and the arguments from the URL, and
        returns an object represented by this feed. Raises
        django.core.exceptions.ObjectDoesNotExist on error.
        """

    # ITEM TITLE AND DESCRIPTION -- If title_template or
    # description_template are not defined, these are used instead. Both are
    # optional, by default they will use the unicode representation of the
    # item.

    def item_title(self, item):
        """
        Takes an item, as returned by items(), and returns the item's
        title as a normal Python string.
        """

    def item_title(self):
        """
        Returns the title for every item in the feed.
        """

    item_title = 'Breaking News: Nothing Happening' # Hard-coded title.

    def item_description(self, item):
        """
        Takes an item, as returned by items(), and returns the item's
        description as a normal Python string.
        """

    def item_description(self):
```

```
        """
        Returns the description for every item in the feed.
        """

    item_description = 'A description of the item.' # Hard-coded description.

    def get_context_data(self, **kwargs):
        """
        Returns a dictionary to use as extra context if either
        description_template or item_template are used.

        Default implementation preserves the old behavior
        of using {'obj': item, 'site': current_site} as the context.
        """

    # ITEM LINK -- One of these three is required. The framework looks for
    # them in this order.

    # First, the framework tries the two methods below, in
    # order. Failing that, it falls back to the get_absolute_url()
    # method on each item returned by items().

    def item_link(self, item):
        """
        Takes an item, as returned by items(), and returns the item's URL.
        """

    def item_link(self):
        """
        Returns the URL for every item in the feed.
        """

    # ITEM_GUID -- The following method is optional. If not provided, the
    # item's link is used by default.

    def item_guid(self, obj):
        """
        Takes an item, as return by items(), and returns the item's ID.
        """

    # ITEM_GUID_IS_PERMALINK -- The following method is optional. If
    # provided, it sets the 'isPermaLink' attribute of an item's
    # GUID element. This method is used only when 'item_guid' is
    # specified.

    def item_guid_is_permalink(self, obj):
        """
        Takes an item, as returned by items(), and returns a boolean.
        """

    item_guid_is_permalink = False  # Hard coded value

    # ITEM AUTHOR NAME -- One of the following three is optional. The
    # framework looks for them in this order.

    def item_author_name(self, item):
        """
        Takes an item, as returned by items(), and returns the item's
```

```
        author's name as a normal Python string.
        """

    def item_author_name(self):
        """
        Returns the author name for every item in the feed.
        """

    item_author_name = 'Sally Smith' # Hard-coded author name.

    # ITEM AUTHOR EMAIL --One of the following three is optional. The
    # framework looks for them in this order.
    #
    # If you specify this, you must specify item_author_name.

    def item_author_email(self, obj):
        """
        Takes an item, as returned by items(), and returns the item's
        author's email as a normal Python string.
        """

    def item_author_email(self):
        """
        Returns the author email for every item in the feed.
        """

    item_author_email = 'test@example.com' # Hard-coded author email.

    # ITEM AUTHOR LINK -- One of the following three is optional. The
    # framework looks for them in this order. In each case, the URL should
    # include the "http://" and domain name.
    #
    # If you specify this, you must specify item_author_name.

    def item_author_link(self, obj):
        """
        Takes an item, as returned by items(), and returns the item's
        author's URL as a normal Python string.
        """

    def item_author_link(self):
        """
        Returns the author URL for every item in the feed.
        """

    item_author_link = 'https://www.example.com/' # Hard-coded author URL.

    # ITEM ENCLOSURES -- One of the following three is optional. The
    # framework looks for them in this order. If one of them is defined,
    # ``item_enclosure_url``, ``item_enclosure_length``, and
    # ``item_enclosure_mime_type`` will have no effect.

    def item_enclosures(self, item):
        """
        Takes an item, as returned by items(), and returns a list of
        ``django.utils.feedgenerator.Enclosure`` objects.
        """
```

```
    def item_enclosures(self):
        """
        Returns the ``django.utils.feedgenerator.Enclosure`` list for every
        item in the feed.
        """

    item_enclosures = []  # Hard-coded enclosure list

    # ITEM ENCLOSURE URL -- One of these three is required if you're
    # publishing enclosures and you're not using ``item_enclosures``. The
    # framework looks for them in this order.

    def item_enclosure_url(self, item):
        """
        Takes an item, as returned by items(), and returns the item's
        enclosure URL.
        """

    def item_enclosure_url(self):
        """
        Returns the enclosure URL for every item in the feed.
        """

    item_enclosure_url = "/foo/bar.mp3" # Hard-coded enclosure link.

    # ITEM ENCLOSURE LENGTH -- One of these three is required if you're
    # publishing enclosures and you're not using ``item_enclosures``. The
    # framework looks for them in this order. In each case, the returned
    # value should be either an integer, or a string representation of the
    # integer, in bytes.

    def item_enclosure_length(self, item):
        """
        Takes an item, as returned by items(), and returns the item's
        enclosure length.
        """

    def item_enclosure_length(self):
        """
        Returns the enclosure length for every item in the feed.
        """

    item_enclosure_length = 32000 # Hard-coded enclosure length.

    # ITEM ENCLOSURE MIME TYPE -- One of these three is required if you're
    # publishing enclosures and you're not using ``item_enclosures``. The
    # framework looks for them in this order.

    def item_enclosure_mime_type(self, item):
        """
        Takes an item, as returned by items(), and returns the item's
        enclosure MIME type.
        """

    def item_enclosure_mime_type(self):
        """
        Returns the enclosure MIME type for every item in the feed.
        """
```

```
    item_enclosure_mime_type = "audio/mpeg" # Hard-coded enclosure MIME type.

    # ITEM PUBDATE -- It's optional to use one of these three. This is a
    # hook that specifies how to get the pubdate for a given item.
    # In each case, the method/attribute should return a Python
    # datetime.datetime object.

    def item_pubdate(self, item):
        """
        Takes an item, as returned by items(), and returns the item's
        pubdate.
        """

    def item_pubdate(self):
        """
        Returns the pubdate for every item in the feed.
        """

    item_pubdate = datetime.datetime(2005, 5, 3) # Hard-coded pubdate.

    # ITEM UPDATED -- It's optional to use one of these three. This is a
    # hook that specifies how to get the updateddate for a given item.
    # In each case, the method/attribute should return a Python
    # datetime.datetime object.

    def item_updateddate(self, item):
        """
        Takes an item, as returned by items(), and returns the item's
        updateddate.
        """

    def item_updateddate(self):
        """
        Returns the updateddate for every item in the feed.
        """

    item_updateddate = datetime.datetime(2005, 5, 3) # Hard-coded updateddate.

    # ITEM CATEGORIES -- It's optional to use one of these three. This is
    # a hook that specifies how to get the list of categories for a given
    # item. In each case, the method/attribute should return an iterable
    # object that returns strings.

    def item_categories(self, item):
        """
        Takes an item, as returned by items(), and returns the item's
        categories.
        """

    def item_categories(self):
        """
        Returns the categories for every item in the feed.
        """

    item_categories = ("python", "django") # Hard-coded categories.

    # ITEM COPYRIGHT NOTICE (only applicable to Atom feeds) -- One of the
    # following three is optional. The framework looks for them in this
```

```
    # order.

    def item_copyright(self, obj):
        """
        Takes an item, as returned by items(), and returns the item's
        copyright notice as a normal Python string.
        """

    def item_copyright(self):
        """
        Returns the copyright notice for every item in the feed.
        """

    item_copyright = 'Copyright (c) 2007, Sally Smith' # Hard-coded copyright notice.
```

The low-level framework

Behind the scenes, the high-level RSS framework uses a lower-level framework for generating feeds' XML. This framework lives in a single module: django/utils/feedgenerator.py.

You use this framework on your own, for lower-level feed generation. You can also create custom feed generator subclasses for use with the `feed_type` `Feed` option.

`SyndicationFeed` classes

The *`feedgenerator`* module contains a base class:

- *`django.utils.feedgenerator.SyndicationFeed`*

and several subclasses:

- *`django.utils.feedgenerator.RssUserland091Feed`*
- *`django.utils.feedgenerator.Rss201rev2Feed`*
- *`django.utils.feedgenerator.Atom1Feed`*

Each of these three classes knows how to render a certain type of feed as XML. They share this interface:

`SyndicationFeed.__init__()` Initialize the feed with the given dictionary of metadata, which applies to the entire feed. Required keyword arguments are:

- `title`
- `link`
- `description`

There's also a bunch of other optional keywords:

- `language`
- `author_email`
- `author_name`
- `author_link`
- `subtitle`
- `categories`

- feed_url
- feed_copyright
- feed_guid
- ttl

Any extra keyword arguments you pass to __init__ will be stored in self.feed for use with *custom feed generators*.

All parameters should be Unicode objects, except categories, which should be a sequence of Unicode objects. Beware that some control characters are not allowed in XML documents. If your content has some of them, you might encounter a ValueError when producing the feed.

SyndicationFeed.add_item() Add an item to the feed with the given parameters.

Required keyword arguments are:

- title
- link
- description

Optional keyword arguments are:

- author_email
- author_name
- author_link
- pubdate
- comments
- unique_id
- enclosure
- enclosures
- categories
- item_copyright
- ttl
- updateddate

Extra keyword arguments will be stored for *custom feed generators*.

All parameters, if given, should be Unicode objects, except:

- pubdate should be a Python datetime object.
- updateddate should be a Python datetime object.
- enclosure should be an instance of *django.utils.feedgenerator.Enclosure*.
- enclosures should be a list of *django.utils.feedgenerator.Enclosure* instances.
- categories should be a sequence of Unicode objects.

1.9 : The enclosure keyword argument is deprecated in favor of the enclosures keyword argument.

SyndicationFeed.write() Outputs the feed in the given encoding to outfile, which is a file-like object.

SyndicationFeed.writeString() Returns the feed as a string in the given encoding.

For example, to create an Atom 1.0 feed and print it to standard output:

```
>>> from django.utils import feedgenerator
>>> from datetime import datetime
>>> f = feedgenerator.Atom1Feed(
...     title="My Weblog",
...     link="https://www.example.com/",
...     description="In which I write about what I ate today.",
...     language="en",
...     author_name="Myself",
...     feed_url="https://example.com/atom.xml")
>>> f.add_item(title="Hot dog today",
...     link="https://www.example.com/entries/1/",
...     pubdate=datetime.now(),
...     description="<p>Today I had a Vienna Beef hot dog. It was pink, plump and perfect.</p>")
>>> print(f.writeString('UTF-8'))
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom" xml:lang="en">
...
</feed>
```

Custom feed generators

If you need to produce a custom feed format, you've got a couple of options.

If the feed format is totally custom, you'll want to subclass `SyndicationFeed` and completely replace the `write()` and `writeString()` methods.

However, if the feed format is a spin-off of RSS or Atom (i.e. GeoRSS, Apple's iTunes podcast format, etc.), you've got a better choice. These types of feeds typically add extra elements and/or attributes to the underlying format, and there are a set of methods that `SyndicationFeed` calls to get these extra attributes. Thus, you can subclass the appropriate feed generator class (`Atom1Feed` or `Rss201rev2Feed`) and extend these callbacks. They are:

`SyndicationFeed.root_attributes(self, )` Return a `dict` of attributes to add to the root feed element (`feed`/`channel`).

`SyndicationFeed.add_root_elements(self, handler)` Callback to add elements inside the root feed element (`feed`/`channel`). `handler` is an `XMLGenerator` from Python's built-in SAX library; you'll call methods on it to add to the XML document in process.

`SyndicationFeed.item_attributes(self, item)` Return a `dict` of attributes to add to each item (`item`/`entry`) element. The argument, `item`, is a dictionary of all the data passed to `SyndicationFeed.add_item()`.

`SyndicationFeed.add_item_elements(self, handler, item)` Callback to add elements to each item (`item`/`entry`) element. `handler` and `item` are as above.

: If you override any of these methods, be sure to call the superclass methods since they add the required elements for each feed format.

For example, you might start implementing an iTunes RSS feed generator like so:

```
class iTunesFeed(Rss201rev2Feed):
    def root_attributes(self):
        attrs = super(iTunesFeed, self).root_attributes()
        attrs['xmlns:itunes'] = 'http://www.itunes.com/dtds/podcast-1.0.dtd'
        return attrs
```

```
    def add_root_elements(self, handler):
        super(iTunesFeed, self).add_root_elements(handler)
        handler.addQuickElement('itunes:explicit', 'clean')
```

Obviously there's a lot more work to be done for a complete custom feed class, but the above example should demonstrate the basic idea.

6.5.14 admin

The automatic Django administrative interface. For more information, see Tutorial 2 and the admin documentation.

Requires the *auth* and *contenttypes* contrib packages to be installed.

6.5.15 auth

Django's authentication framework.

See User authentication in Django.

6.5.16 contenttypes

A light framework for hooking into "types" of content, where each installed Django model is a separate content type.

See the contenttypes documentation.

6.5.17 flatpages

A framework for managing simple "flat" HTML content in a database.

See the flatpages documentation.

Requires the *sites* contrib package to be installed as well.

6.5.18 gis

A world-class geospatial framework built on top of Django, that enables storage, manipulation and display of spatial data.

See the GeoDjango documentation for more.

6.5.19 humanize

A set of Django template filters useful for adding a "human touch" to data.

See the humanize documentation.

6.5.20 messages

A framework for storing and retrieving temporary cookie- or session-based messages

See the messages documentation.

6.5.21 `postgres`

A collection of PostgreSQL specific features.

See the contrib.postgres documentation.

6.5.22 `redirects`

A framework for managing redirects.

See the redirects documentation.

6.5.23 `sessions`

A framework for storing data in anonymous sessions.

See the sessions documentation.

6.5.24 `sites`

A light framework that lets you operate multiple websites off of the same database and Django installation. It gives you hooks for associating objects to one or more sites.

See the sites documentation.

6.5.25 `sitemaps`

A framework for generating Google sitemap XML files.

See the sitemaps documentation.

6.5.26 `syndication`

A framework for generating syndication feeds, in RSS and Atom, quite easily.

See the syndication documentation.

6.5.27 Other add-ons

If you have an idea for functionality to include in `contrib`, let us know! Code it up, and post it to the *django-users* mailing list.

6.6 Cross Site Request Forgery protection

The CSRF middleware and template tag provides easy-to-use protection against Cross Site Request Forgeries. This type of attack occurs when a malicious website contains a link, a form button or some JavaScript that is intended to perform some action on your website, using the credentials of a logged-in user who visits the malicious site in their browser. A related type of attack, 'login CSRF', where an attacking site tricks a user's browser into logging into a site with someone else's credentials, is also covered.

The first defense against CSRF attacks is to ensure that GET requests (and other 'safe' methods, as defined by RFC 7231#section-4.2.1) are side effect free. Requests via 'unsafe' methods, such as POST, PUT, and DELETE, can then be protected by following the steps below.

6.6.1 How to use it

To take advantage of CSRF protection in your views, follow these steps:

1. The CSRF middleware is activated by default in the *MIDDLEWARE* setting. If you override that setting, remember that `'django.middleware.csrf.CsrfViewMiddleware'` should come before any view middleware that assume that CSRF attacks have been dealt with.

   If you disabled it, which is not recommended, you can use *csrf_protect()* on particular views you want to protect (see below).

2. In any template that uses a POST form, use the *csrf_token* tag inside the `<form>` element if the form is for an internal URL, e.g.:

```
<form action="" method="post">{% csrf_token %}
```

   This should not be done for POST forms that target external URLs, since that would cause the CSRF token to be leaked, leading to a vulnerability.

3. In the corresponding view functions, ensure that *RequestContext* is used to render the response so that `{% csrf_token %}` will work properly. If you're using the *render()* function, generic views, or contrib apps, you are covered already since these all use `RequestContext`.

AJAX

While the above method can be used for AJAX POST requests, it has some inconveniences: you have to remember to pass the CSRF token in as POST data with every POST request. For this reason, there is an alternative method: on each XMLHttpRequest, set a custom `X-CSRFToken` header to the value of the CSRF token. This is often easier, because many JavaScript frameworks provide hooks that allow headers to be set on every request.

As a first step, you must get the CSRF token itself. The recommended source for the token is the `csrftoken` cookie, which will be set if you've enabled CSRF protection for your views as outlined above.

: The CSRF token cookie is named `csrftoken` by default, but you can control the cookie name via the *CSRF_COOKIE_NAME* setting.

The CSRF header name is `HTTP_X_CSRFTOKEN` by default, but you can customize it using the *CSRF_HEADER_NAME* setting.

Acquiring the token is straightforward:

```
// using jQuery
function getCookie(name) {
    var cookieValue = null;
    if (document.cookie && document.cookie !== '') {
        var cookies = document.cookie.split(';');
        for (var i = 0; i < cookies.length; i++) {
            var cookie = jQuery.trim(cookies[i]);
            // Does this cookie string begin with the name we want?
            if (cookie.substring(0, name.length + 1) === (name + '=')) {
                cookieValue = decodeURIComponent(cookie.substring(name.length + 1));
                break;
```

```
            }
        }
    }
    return cookieValue;
}
var csrftoken = getCookie('csrftoken');
```

The above code could be simplified by using the JavaScript Cookie library to replace `getCookie`:

```
var csrftoken = Cookies.get('csrftoken');
```

: The CSRF token is also present in the DOM, but only if explicitly included using *csrf_token* in a template. The cookie contains the canonical token; the `CsrfViewMiddleware` will prefer the cookie to the token in the DOM. Regardless, you're guaranteed to have the cookie if the token is present in the DOM, so you should use the cookie!

: If your view is not rendering a template containing the *csrf_token* template tag, Django might not set the CSRF token cookie. This is common in cases where forms are dynamically added to the page. To address this case, Django provides a view decorator which forces setting of the cookie: *ensure_csrf_cookie()*.

Finally, you'll have to actually set the header on your AJAX request, while protecting the CSRF token from being sent to other domains using settings.crossDomain in jQuery 1.5.1 and newer:

```
function csrfSafeMethod(method) {
    // these HTTP methods do not require CSRF protection
    return (/^(GET|HEAD|OPTIONS|TRACE)$/.test(method));
}
$.ajaxSetup({
    beforeSend: function(xhr, settings) {
        if (!csrfSafeMethod(settings.type) && !this.crossDomain) {
            xhr.setRequestHeader("X-CSRFToken", csrftoken);
        }
    }
});
```

If you're using AngularJS 1.1.3 and newer, it's sufficient to configure the `$http` provider with the cookie and header names:

```
$httpProvider.defaults.xsrfCookieName = 'csrftoken';
$httpProvider.defaults.xsrfHeaderName = 'X-CSRFToken';
```

Using CSRF in Jinja2 templates

Django's *Jinja2* template backend adds `{{ csrf_input }}` to the context of all templates which is equivalent to `{% csrf_token %}` in the Django template language. For example:

```
<form action="" method="post">{{ csrf_input }}
```

The decorator method

Rather than adding `CsrfViewMiddleware` as a blanket protection, you can use the `csrf_protect` decorator, which has exactly the same functionality, on particular views that need the protection. It must be used **both** on views that insert the CSRF token in the output, and on those that accept the POST form data. (These are often the same view function, but not always).

Use of the decorator by itself is **not recommended**, since if you forget to use it, you will have a security hole. The 'belt and braces' strategy of using both is fine, and will incur minimal overhead.

csrf_protect(*view*)
Decorator that provides the protection of `CsrfViewMiddleware` to a view.

Usage:

```
from django.views.decorators.csrf import csrf_protect
from django.shortcuts import render

@csrf_protect
def my_view(request):
    c = {}
    # ...
    return render(request, "a_template.html", c)
```

If you are using class-based views, you can refer to *Decorating class-based views*.

6.6.2 Rejected requests

By default, a '403 Forbidden' response is sent to the user if an incoming request fails the checks performed by `CsrfViewMiddleware`. This should usually only be seen when there is a genuine Cross Site Request Forgery, or when, due to a programming error, the CSRF token has not been included with a POST form.

The error page, however, is not very friendly, so you may want to provide your own view for handling this condition. To do this, simply set the *CSRF_FAILURE_VIEW* setting.

CSRF failures are logged as warnings to the *django.request* logger.

6.6.3 How it works

The CSRF protection is based on the following things:

1. A CSRF cookie that is based on a random secret value, which other sites will not have access to.

   This cookie is set by `CsrfViewMiddleware`. It is sent with every response that has called `django.middleware.csrf.get_token()` (the function used internally to retrieve the CSRF token), if it wasn't already set on the request.

   In order to protect against BREACH attacks, the token is not simply the secret; a random salt is prepended to the secret and used to scramble it.

   For security reasons, the value of the secret is changed each time a user logs in.

2. A hidden form field with the name 'csrfmiddlewaretoken' present in all outgoing POST forms. The value of this field is, again, the value of the secret, with a salt which is both added to it and used to scramble it. The salt is regenerated on every call to `get_token()` so that the form field value is changed in every such response.

   This part is done by the template tag.

3. For all incoming requests that are not using HTTP GET, HEAD, OPTIONS or TRACE, a CSRF cookie must be present, and the 'csrfmiddlewaretoken' field must be present and correct. If it isn't, the user will get a 403 error.

   When validating the 'csrfmiddlewaretoken' field value, only the secret, not the full token, is compared with the secret in the cookie value. This allows the use of ever-changing tokens. While each request may use its own token, the secret remains common to all.

   This check is done by `CsrfViewMiddleware`.

4. In addition, for HTTPS requests, strict referer checking is done by `CsrfViewMiddleware`. This means that even if a subdomain can set or modify cookies on your domain, it can't force a user to post to your application since that request won't come from your own exact domain.

   This also addresses a man-in-the-middle attack that's possible under HTTPS when using a session independent secret, due to the fact that HTTP `Set-Cookie` headers are (unfortunately) accepted by clients even when they are talking to a site under HTTPS. (Referer checking is not done for HTTP requests because the presence of the `Referer` header isn't reliable enough under HTTP.)

   If the *CSRF_COOKIE_DOMAIN* setting is set, the referer is compared against it. This setting supports subdomains. For example, `CSRF_COOKIE_DOMAIN = '.example.com'` will allow POST requests from `www.example.com` and `api.example.com`. If the setting is not set, then the referer must match the HTTP `Host` header.

   Expanding the accepted referers beyond the current host or cookie domain can be done with the *CSRF_TRUSTED_ORIGINS* setting.

This ensures that only forms that have originated from trusted domains can be used to POST data back.

It deliberately ignores GET requests (and other requests that are defined as 'safe' by **RFC 7231**). These requests ought never to have any potentially dangerous side effects , and so a CSRF attack with a GET request ought to be harmless. **RFC 7231** defines POST, PUT, and DELETE as 'unsafe', and all other methods are also assumed to be unsafe, for maximum protection.

The CSRF protection cannot protect against man-in-the-middle attacks, so use *HTTPS* with *HTTP Strict Transport Security*. It also assumes *validation of the HOST header* and that there aren't any *cross-site scripting vulnerabilities* on your site (because XSS vulnerabilities already let an attacker do anything a CSRF vulnerability allows and much worse).

Checking against the *CSRF_COOKIE_DOMAIN* setting was added.

Added salting to the token and started changing it with each request to protect against BREACH attacks.

6.6.4 Caching

If the *csrf_token* template tag is used by a template (or the `get_token` function is called some other way), `CsrfViewMiddleware` will add a cookie and a `Vary: Cookie` header to the response. This means that the middleware will play well with the cache middleware if it is used as instructed (`UpdateCacheMiddleware` goes before all other middleware).

However, if you use cache decorators on individual views, the CSRF middleware will not yet have been able to set the Vary header or the CSRF cookie, and the response will be cached without either one. In this case, on any views that will require a CSRF token to be inserted you should use the *django.views.decorators.csrf.csrf_protect()* decorator first:

```
from django.views.decorators.cache import cache_page
from django.views.decorators.csrf import csrf_protect

@cache_page(60 * 15)
@csrf_protect
def my_view(request):
    ...
```

If you are using class-based views, you can refer to *Decorating class-based views*.

6.6.5 Testing

The `CsrfViewMiddleware` will usually be a big hindrance to testing view functions, due to the need for the CSRF token which must be sent with every POST request. For this reason, Django's HTTP client for tests has been modified to set a flag on requests which relaxes the middleware and the `csrf_protect` decorator so that they no longer rejects requests. In every other respect (e.g. sending cookies etc.), they behave the same.

If, for some reason, you *want* the test client to perform CSRF checks, you can create an instance of the test client that enforces CSRF checks:

```
>>> from django.test import Client
>>> csrf_client = Client(enforce_csrf_checks=True)
```

6.6.6 Limitations

Subdomains within a site will be able to set cookies on the client for the whole domain. By setting the cookie and using a corresponding token, subdomains will be able to circumvent the CSRF protection. The only way to avoid this is to ensure that subdomains are controlled by trusted users (or, are at least unable to set cookies). Note that even without CSRF, there are other vulnerabilities, such as session fixation, that make giving subdomains to untrusted parties a bad idea, and these vulnerabilities cannot easily be fixed with current browsers.

6.6.7 Edge cases

Certain views can have unusual requirements that mean they don't fit the normal pattern envisaged here. A number of utilities can be useful in these situations. The scenarios they might be needed in are described in the following section.

Utilities

The examples below assume you are using function-based views. If you are working with class-based views, you can refer to *Decorating class-based views*.

csrf_exempt(*view*)

This decorator marks a view as being exempt from the protection ensured by the middleware. Example:

```
from django.views.decorators.csrf import csrf_exempt
from django.http import HttpResponse

@csrf_exempt
def my_view(request):
    return HttpResponse('Hello world')
```

requires_csrf_token(*view*)

Normally the *csrf_token* template tag will not work if `CsrfViewMiddleware.process_view` or an equivalent like `csrf_protect` has not run. The view decorator `requires_csrf_token` can be used to ensure the template tag does work. This decorator works similarly to `csrf_protect`, but never rejects an incoming request.

Example:

```
from django.views.decorators.csrf import requires_csrf_token
from django.shortcuts import render

@requires_csrf_token
def my_view(request):
    c = {}
```

```
        # ...
        return render(request, "a_template.html", c)
```

ensure_csrf_cookie(*view*)
This decorator forces a view to send the CSRF cookie.

Scenarios

CSRF protection should be disabled for just a few views

Most views requires CSRF protection, but a few do not.

Solution: rather than disabling the middleware and applying `csrf_protect` to all the views that need it, enable the middleware and use *csrf_exempt()*.

CsrfViewMiddleware.process_view not used

There are cases when `CsrfViewMiddleware.process_view` may not have run before your view is run - 404 and 500 handlers, for example - but you still need the CSRF token in a form.

Solution: use *requires_csrf_token()*

Unprotected view needs the CSRF token

There may be some views that are unprotected and have been exempted by `csrf_exempt`, but still need to include the CSRF token.

Solution: use *csrf_exempt()* followed by *requires_csrf_token()*. (i.e. `requires_csrf_token` should be the innermost decorator).

View needs protection for one path

A view needs CSRF protection under one set of conditions only, and mustn't have it for the rest of the time.

Solution: use *csrf_exempt()* for the whole view function, and *csrf_protect()* for the path within it that needs protection. Example:

```
from django.views.decorators.csrf import csrf_exempt, csrf_protect

@csrf_exempt
def my_view(request):

    @csrf_protect
    def protected_path(request):
        do_something()

    if some_condition():
       return protected_path(request)
    else:
       do_something_else()
```

Page uses AJAX without any HTML form

A page makes a POST request via AJAX, and the page does not have an HTML form with a *csrf_token* that would cause the required CSRF cookie to be sent.

Solution: use *ensure_csrf_cookie()* on the view that sends the page.

6.6.8 Contrib and reusable apps

Because it is possible for the developer to turn off the `CsrfViewMiddleware`, all relevant views in contrib apps use the `csrf_protect` decorator to ensure the security of these applications against CSRF. It is recommended that the developers of other reusable apps that want the same guarantees also use the `csrf_protect` decorator on their views.

6.6.9 Settings

A number of settings can be used to control Django's CSRF behavior:

- *CSRF_COOKIE_AGE*
- *CSRF_COOKIE_DOMAIN*
- *CSRF_COOKIE_HTTPONLY*
- *CSRF_COOKIE_NAME*
- *CSRF_COOKIE_PATH*
- *CSRF_COOKIE_SECURE*
- *CSRF_FAILURE_VIEW*
- *CSRF_HEADER_NAME*
- *CSRF_TRUSTED_ORIGINS*

6.6.10 Frequently Asked Questions

Is posting an arbitrary CSRF token pair (cookie and POST data) a vulnerability?

No, this is by design. Without a man-in-the-middle attack, there is no way for an attacker to send a CSRF token cookie to a victim's browser, so a successful attack would need to obtain the victim's browser's cookie via XSS or similar, in which case an attacker usually doesn't need CSRF attacks.

Some security audit tools flag this as a problem but as mentioned before, an attacker cannot steal a user's browser's CSRF cookie. "Stealing" or modifying *your own* token using Firebug, Chrome dev tools, etc. isn't a vulnerability.

Is the fact that Django's CSRF protection isn't linked to a session a problem?

No, this is by design. Not linking CSRF protection to a session allows using the protection on sites such as a *pastebin* that allow submissions from anonymous users which don't have a session.

Why might a user encounter a CSRF validation failure after logging in?

For security reasons, CSRF tokens are rotated each time a user logs in. Any page with a form generated before a login will have an old, invalid CSRF token and need to be reloaded. This might happen if a user uses the back button after a login or if they log in in a different browser tab.

6.7 Databases

Django attempts to support as many features as possible on all database backends. However, not all database backends are alike, and we've had to make design decisions on which features to support and which assumptions we can make safely.

This file describes some of the features that might be relevant to Django usage. Of course, it is not intended as a replacement for server-specific documentation or reference manuals.

6.7.1 General notes

Persistent connections

Persistent connections avoid the overhead of re-establishing a connection to the database in each request. They're controlled by the *CONN_MAX_AGE* parameter which defines the maximum lifetime of a connection. It can be set independently for each database.

The default value is `0`, preserving the historical behavior of closing the database connection at the end of each request. To enable persistent connections, set *CONN_MAX_AGE* to a positive number of seconds. For unlimited persistent connections, set it to `None`.

Connection management

Django opens a connection to the database when it first makes a database query. It keeps this connection open and reuses it in subsequent requests. Django closes the connection once it exceeds the maximum age defined by *CONN_MAX_AGE* or when it isn't usable any longer.

In detail, Django automatically opens a connection to the database whenever it needs one and doesn't have one already — either because this is the first connection, or because the previous connection was closed.

At the beginning of each request, Django closes the connection if it has reached its maximum age. If your database terminates idle connections after some time, you should set *CONN_MAX_AGE* to a lower value, so that Django doesn't attempt to use a connection that has been terminated by the database server. (This problem may only affect very low traffic sites.)

At the end of each request, Django closes the connection if it has reached its maximum age or if it is in an unrecoverable error state. If any database errors have occurred while processing the requests, Django checks whether the connection still works, and closes it if it doesn't. Thus, database errors affect at most one request; if the connection becomes unusable, the next request gets a fresh connection.

Caveats

Since each thread maintains its own connection, your database must support at least as many simultaneous connections as you have worker threads.

Sometimes a database won't be accessed by the majority of your views, for example because it's the database of an external system, or thanks to caching. In such cases, you should set *CONN_MAX_AGE* to a low value or even `0`, because it doesn't make sense to maintain a connection that's unlikely to be reused. This will help keep the number of simultaneous connections to this database small.

The development server creates a new thread for each request it handles, negating the effect of persistent connections. Don't enable them during development.

When Django establishes a connection to the database, it sets up appropriate parameters, depending on the backend being used. If you enable persistent connections, this setup is no longer repeated every request. If you modify parameters such as the connection's isolation level or time zone, you should either restore Django's defaults at the end of each request, force an appropriate value at the beginning of each request, or disable persistent connections.

Encoding

Django assumes that all databases use UTF-8 encoding. Using other encodings may result in unexpected behavior such as "value too long" errors from your database for data that is valid in Django. See the database specific notes below for information on how to set up your database correctly.

6.7.2 PostgreSQL notes

Django supports PostgreSQL 9.2 and higher. It requires the use of psycopg2 2.4.5 or higher (or 2.5+ if you want to use *django.contrib.postgres*).

PostgreSQL connection settings

See *HOST* for details.

Optimizing PostgreSQL's configuration

Django needs the following parameters for its database connections:

- `client_encoding`: `'UTF8'`,
- `default_transaction_isolation`: `'read committed'` by default, or the value set in the connection options (see below),
- `timezone`: `'UTC'` when *USE_TZ* is `True`, value of *TIME_ZONE* otherwise.

If these parameters already have the correct values, Django won't set them for every new connection, which improves performance slightly. You can configure them directly in `postgresql.conf` or more conveniently per database user with ALTER ROLE.

Django will work just fine without this optimization, but each new connection will do some additional queries to set these parameters.

Isolation level

Like PostgreSQL itself, Django defaults to the `READ COMMITTED` isolation level. If you need a higher isolation level such as `REPEATABLE READ` or `SERIALIZABLE`, set it in the *OPTIONS* part of your database configuration in *DATABASES*:

```
import psycopg2.extensions

DATABASES = {
    # ...
    'OPTIONS': {
        'isolation_level': psycopg2.extensions.ISOLATION_LEVEL_SERIALIZABLE,
    },
}
```

: Under higher isolation levels, your application should be prepared to handle exceptions raised on serialization failures. This option is designed for advanced uses.

Indexes for `varchar` and `text` columns

When specifying `db_index=True` on your model fields, Django typically outputs a single `CREATE INDEX` statement. However, if the database type for the field is either `varchar` or `text` (e.g., used by `CharField`, `FileField`, and `TextField`), then Django will create an additional index that uses an appropriate PostgreSQL operator class for the column. The extra index is necessary to correctly perform lookups that use the `LIKE` operator in their SQL, as is done with the `contains` and `startswith` lookup types.

Migration operation for adding extensions

If you need to add a PostgreSQL extension (like `hstore`, `postgis`, etc.) using a migration, use the *`CreateExtension`* operation.

Speeding up test execution with non-durable settings

You can speed up test execution times by configuring PostgreSQL to be non-durable.

: This is dangerous: it will make your database more susceptible to data loss or corruption in the case of a server crash or power loss. Only use this on a development machine where you can easily restore the entire contents of all databases in the cluster.

6.7.3 MySQL notes

Version support

Django supports MySQL 5.5 and higher.

Django's `inspectdb` feature uses the `information_schema` database, which contains detailed data on all database schemas.

Django expects the database to support Unicode (UTF-8 encoding) and delegates to it the task of enforcing transactions and referential integrity. It is important to be aware of the fact that the two latter ones aren't actually enforced by MySQL when using the MyISAM storage engine, see the next section.

Storage engines

MySQL has several storage engines. You can change the default storage engine in the server configuration.

Until MySQL 5.5.4, the default engine was MyISAM [1]. The main drawbacks of MyISAM are that it doesn't support transactions or enforce foreign-key constraints. On the plus side, it was the only engine that supported full-text indexing and searching until MySQL 5.6.4.

Since MySQL 5.5.5, the default storage engine is InnoDB. This engine is fully transactional and supports foreign key references. It's probably the best choice at this point. However, note that the InnoDB autoincrement counter is lost on a MySQL restart because it does not remember the `AUTO_INCREMENT` value, instead recreating it as "max(id)+1". This may result in an inadvertent reuse of *`AutoField`* values.

If you upgrade an existing project to MySQL 5.5.5 and subsequently add some tables, ensure that your tables are using the same storage engine (i.e. MyISAM vs. InnoDB). Specifically, if tables that have a `ForeignKey` between them use different storage engines, you may see an error like the following when running `migrate`:

```
_mysql_exceptions.OperationalError: (
    1005, "Can't create table '\\db_name\\.#sql-4a8_ab' (errno: 150)"
)
```

MySQL DB API Drivers

The Python Database API is described in **PEP 249**. MySQL has three prominent drivers that implement this API:

- MySQLdb is a native driver that has been developed and supported for over a decade by Andy Dustman.
- mysqlclient is a fork of `MySQLdb` which notably supports Python 3 and can be used as a drop-in replacement for MySQLdb. At the time of this writing, this is **the recommended choice** for using MySQL with Django.
- MySQL Connector/Python is a pure Python driver from Oracle that does not require the MySQL client library or any Python modules outside the standard library.

All these drivers are thread-safe and provide connection pooling. `MySQLdb` is the only one not supporting Python 3 currently.

In addition to a DB API driver, Django needs an adapter to access the database drivers from its ORM. Django provides an adapter for MySQLdb/mysqlclient while MySQL Connector/Python includes its own.

MySQLdb

Django requires MySQLdb version 1.2.1p2 or later.

At the time of writing, the latest release of MySQLdb (1.2.5) doesn't support Python 3. In order to use MySQLdb under Python 3, you'll have to install `mysqlclient` instead.

: There are known issues with the way MySQLdb converts date strings into datetime objects. Specifically, date strings with value `0000-00-00` are valid for MySQL but will be converted into `None` by MySQLdb.

This means you should be careful while using *`loaddata`* and *`dumpdata`* with rows that may have `0000-00-00` values, as they will be converted to `None`.

---

[1] Unless this was changed by the packager of your MySQL package. We've had reports that the Windows Community Server installer sets up InnoDB as the default storage engine, for example.

mysqlclient

Django requires mysqlclient 1.3.3 or later. Note that Python 3.2 is not supported. Except for the Python 3.3+ support, mysqlclient should mostly behave the same as MySQLDB.

MySQL Connector/Python

MySQL Connector/Python is available from the download page. The Django adapter is available in versions 1.1.X and later. It may not support the most recent releases of Django.

Time zone definitions

If you plan on using Django's timezone support, use mysql_tzinfo_to_sql to load time zone tables into the MySQL database. This needs to be done just once for your MySQL server, not per database.

Creating your database

You can create your database using the command-line tools and this SQL:

```
CREATE DATABASE <dbname> CHARACTER SET utf8;
```

This ensures all tables and columns will use UTF-8 by default.

Collation settings

The collation setting for a column controls the order in which data is sorted as well as what strings compare as equal. It can be set on a database-wide level and also per-table and per-column. This is documented thoroughly in the MySQL documentation. In all cases, you set the collation by directly manipulating the database tables; Django doesn't provide a way to set this on the model definition.

By default, with a UTF-8 database, MySQL will use the `utf8_general_ci` collation. This results in all string equality comparisons being done in a *case-insensitive* manner. That is, `"Fred"` and `"freD"` are considered equal at the database level. If you have a unique constraint on a field, it would be illegal to try to insert both `"aa"` and `"AA"` into the same column, since they compare as equal (and, hence, non-unique) with the default collation.

In many cases, this default will not be a problem. However, if you really want case-sensitive comparisons on a particular column or table, you would change the column or table to use the `utf8_bin` collation. The main thing to be aware of in this case is that if you are using MySQLdb 1.2.2, the database backend in Django will then return bytestrings (instead of unicode strings) for any character fields it receive from the database. This is a strong variation from Django's normal practice of *always* returning unicode strings. It is up to you, the developer, to handle the fact that you will receive bytestrings if you configure your table(s) to use `utf8_bin` collation. Django itself should mostly work smoothly with such columns (except for the `contrib.sessions` `Session` and `contrib.admin` `LogEntry` tables described below), but your code must be prepared to call `django.utils.encoding.smart_text()` at times if it really wants to work with consistent data – Django will not do this for you (the database backend layer and the model population layer are separated internally so the database layer doesn't know it needs to make this conversion in this one particular case).

If you're using MySQLdb 1.2.1p2, Django's standard *`CharField`* class will return unicode strings even with `utf8_bin` collation. However, *`TextField`* fields will be returned as an `array.array` instance (from Python's standard `array` module). There isn't a lot Django can do about that, since, again, the information needed to make the necessary conversions isn't available when the data is read in from the database. This problem was fixed in MySQLdb 1.2.2, so if you want to use *`TextField`* with `utf8_bin` collation, upgrading to version 1.2.2 and then dealing with the bytestrings (which shouldn't be too difficult) as described above is the recommended solution.

Should you decide to use `utf8_bin` collation for some of your tables with MySQLdb 1.2.1p2 or 1.2.2, you should still use `utf8_general_ci` (the default) collation for the `django.contrib.sessions.models.Session` table (usually called `django_session`) and the *`django.contrib.admin.models.LogEntry`* table (usually called `django_admin_log`). Those are the two standard tables that use *`TextField`* internally.

Please note that according to MySQL Unicode Character Sets, comparisons for the `utf8_general_ci` collation are faster, but slightly less correct, than comparisons for `utf8_unicode_ci`. If this is acceptable for your application, you should use `utf8_general_ci` because it is faster. If this is not acceptable (for example, if you require German dictionary order), use `utf8_unicode_ci` because it is more accurate.

: Model formsets validate unique fields in a case-sensitive manner. Thus when using a case-insensitive collation, a formset with unique field values that differ only by case will pass validation, but upon calling `save()`, an `IntegrityError` will be raised.

Connecting to the database

Refer to the settings documentation.

Connection settings are used in this order:

1. *`OPTIONS`*.
2. *`NAME`*, *`USER`*, *`PASSWORD`*, *`HOST`*, *`PORT`*
3. MySQL option files.

In other words, if you set the name of the database in *`OPTIONS`*, this will take precedence over *`NAME`*, which would override anything in a MySQL option file.

Here's a sample configuration which uses a MySQL option file:

```
# settings.py
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'OPTIONS': {
            'read_default_file': '/path/to/my.cnf',
        },
    }
}


# my.cnf
[client]
database = NAME
user = USER
password = PASSWORD
default-character-set = utf8
```

Several other MySQLdb connection options may be useful, such as `ssl`, `init_command`, and `sql_mode`. Consult the MySQLdb documentation for more details.

Setting `sql_mode`

From MySQL 5.7 onwards and on fresh installs of MySQL 5.6, the default value of the `sql_mode` option contains `STRICT_TRANS_TABLES`. That option escalates warnings into errors when data are truncated

upon insertion, so Django highly recommends activating a strict mode for MySQL to prevent data loss (either `STRICT_TRANS_TABLES` or `STRICT_ALL_TABLES`).

If you need to customize the SQL mode, you can set the `sql_mode` variable like other MySQL options: either in a config file or with the entry `'init_command': "SET sql_mode='STRICT_TRANS_TABLES'"` in the *`OPTIONS`* part of your database configuration in *`DATABASES`*.

Creating your tables

When Django generates the schema, it doesn't specify a storage engine, so tables will be created with whatever default storage engine your database server is configured for. The easiest solution is to set your database server's default storage engine to the desired engine.

If you're using a hosting service and can't change your server's default storage engine, you have a couple of options.

- After the tables are created, execute an `ALTER TABLE` statement to convert a table to a new storage engine (such as InnoDB):

```
ALTER TABLE <tablename> ENGINE=INNODB;
```

  This can be tedious if you have a lot of tables.

- Another option is to use the `init_command` option for MySQLdb prior to creating your tables:

```
'OPTIONS': {
   'init_command': 'SET default_storage_engine=INNODB',
}
```

  This sets the default storage engine upon connecting to the database. After your tables have been created, you should remove this option as it adds a query that is only needed during table creation to each database connection.

Table names

There are known issues in even the latest versions of MySQL that can cause the case of a table name to be altered when certain SQL statements are executed under certain conditions. It is recommended that you use lowercase table names, if possible, to avoid any problems that might arise from this behavior. Django uses lowercase table names when it auto-generates table names from models, so this is mainly a consideration if you are overriding the table name via the *`db_table`* parameter.

Savepoints

Both the Django ORM and MySQL (when using the InnoDB *storage engine*) support database *savepoints*.

If you use the MyISAM storage engine please be aware of the fact that you will receive database-generated errors if you try to use the *savepoint-related methods of the transactions API*. The reason for this is that detecting the storage engine of a MySQL database/table is an expensive operation so it was decided it isn't worth to dynamically convert these methods in no-op's based in the results of such detection.

Notes on specific fields

Character fields

Any fields that are stored with `VARCHAR` column types have their `max_length` restricted to 255 characters if you are using `unique=True` for the field. This affects *`CharField`*, *`SlugField`* and

CommaSeparatedIntegerField.

`TextField` limitations

MySQL can index only the first N chars of a `BLOB` or `TEXT` column. Since `TextField` doesn't have a defined length, you can't mark it as `unique=True`. MySQL will report: "BLOB/TEXT column '<db_column>' used in key specification without a key length".

Fractional seconds support for Time and DateTime fields

MySQL 5.6.4 and later can store fractional seconds, provided that the column definition includes a fractional indication (e.g. `DATETIME(6)`). Earlier versions do not support them at all. In addition, versions of MySQLdb older than 1.2.5 have a bug that also prevents the use of fractional seconds with MySQL.

Django will not upgrade existing columns to include fractional seconds if the database server supports it. If you want to enable them on an existing database, it's up to you to either manually update the column on the target database, by executing a command like:

```
ALTER TABLE `your_table` MODIFY `your_datetime_column` DATETIME(6)
```

or using a *RunSQL* operation in a *data migration*.

`TIMESTAMP` columns

If you are using a legacy database that contains `TIMESTAMP` columns, you must set *USE_TZ = False* to avoid data corruption. *inspectdb* maps these columns to *DateTimeField* and if you enable timezone support, both MySQL and Django will attempt to convert the values from UTC to local time.

Row locking with `QuerySet.select_for_update()`

MySQL does not support the `NOWAIT` option to the `SELECT ... FOR UPDATE` statement. If `select_for_update()` is used with `nowait=True` then a `DatabaseError` will be raised.

Automatic typecasting can cause unexpected results

When performing a query on a string type, but with an integer value, MySQL will coerce the types of all values in the table to an integer before performing the comparison. If your table contains the values `'abc'`, `'def'` and you query for `WHERE mycolumn=0`, both rows will match. Similarly, `WHERE mycolumn=1` will match the value `'abc1'`. Therefore, string type fields included in Django will always cast the value to a string before using it in a query.

If you implement custom model fields that inherit from *Field* directly, are overriding *get_prep_value()*, or use *RawSQL*, *extra()*, or *raw()*, you should ensure that you perform appropriate typecasting.

6.7.4 SQLite notes

SQLite provides an excellent development alternative for applications that are predominantly read-only or require a smaller installation footprint. As with all database servers, though, there are some differences that are specific to SQLite that you should be aware of.

Substring matching and case sensitivity

For all SQLite versions, there is some slightly counter-intuitive behavior when attempting to match some types of strings. These are triggered when using the *iexact* or *contains* filters in Querysets. The behavior splits into two cases:

1. For substring matching, all matches are done case-insensitively. That is a filter such as `filter(name__contains="aa")` will match a name of `"Aabb"`.

2. For strings containing characters outside the ASCII range, all exact string matches are performed case-sensitively, even when the case-insensitive options are passed into the query. So the *iexact* filter will behave exactly the same as the *exact* filter in these cases.

Some possible workarounds for this are documented at sqlite.org, but they aren't utilized by the default SQLite backend in Django, as incorporating them would be fairly difficult to do robustly. Thus, Django exposes the default SQLite behavior and you should be aware of this when doing case-insensitive or substring filtering.

Old SQLite and CASE expressions

SQLite 3.6.23.1 and older contains a bug when handling query parameters in a `CASE` expression that contains an `ELSE` and arithmetic.

SQLite 3.6.23.1 was released in March 2010, and most current binary distributions for different platforms include a newer version of SQLite, with the notable exception of the Python 2.7 installers for Windows.

As of this writing, the latest release for Windows - Python 2.7.10 - includes SQLite 3.6.21. You can install `pysqlite2` or replace `sqlite3.dll` (by default installed in `C:\Python27\DLLs`) with a newer version from https://www.sqlite.org/ to remedy this issue.

Using newer versions of the SQLite DB-API 2.0 driver

Django will use a `pysqlite2` module in preference to `sqlite3` as shipped with the Python standard library if it finds one is available.

This provides the ability to upgrade both the DB-API 2.0 interface or SQLite 3 itself to versions newer than the ones included with your particular Python binary distribution, if needed.

"Database is locked" errors

SQLite is meant to be a lightweight database, and thus can't support a high level of concurrency. `OperationalError: database is locked` errors indicate that your application is experiencing more concurrency than `sqlite` can handle in default configuration. This error means that one thread or process has an exclusive lock on the database connection and another thread timed out waiting for the lock the be released.

Python's SQLite wrapper has a default timeout value that determines how long the second thread is allowed to wait on the lock before it times out and raises the `OperationalError: database is locked` error.

If you're getting this error, you can solve it by:

- Switching to another database backend. At a certain point SQLite becomes too "lite" for real-world applications, and these sorts of concurrency errors indicate you've reached that point.
- Rewriting your code to reduce concurrency and ensure that database transactions are short-lived.
- Increase the default timeout value by setting the `timeout` database option:

```
    'OPTIONS': {
        # ...
        'timeout': 20,
        # ...
    }
```

This will simply make SQLite wait a bit longer before throwing "database is locked" errors; it won't really do anything to solve them.

`QuerySet.select_for_update()` not supported

SQLite does not support the `SELECT ... FOR UPDATE` syntax. Calling it will have no effect.

"pyformat" parameter style in raw queries not supported

For most backends, raw queries (`Manager.raw()` or `cursor.execute()`) can use the "pyformat" parameter style, where placeholders in the query are given as `'%(name)s'` and the parameters are passed as a dictionary rather than a list. SQLite does not support this.

6.7.5 Oracle notes

Django supports Oracle Database Server versions 11.2 and higher. Version 4.3.1 or higher of the cx_Oracle Python driver is required, although we recommend version 5.1.3 or later as these versions support Python 3.

Note that due to a Unicode-corruption bug in `cx_Oracle` 5.0, that version of the driver should **not** be used with Django; `cx_Oracle` 5.0.1 resolved this issue, so if you'd like to use a more recent `cx_Oracle`, use version 5.0.1.

`cx_Oracle` 5.0.1 or greater can optionally be compiled with the `WITH_UNICODE` environment variable. This is recommended but not required.

In order for the `python manage.py migrate` command to work, your Oracle database user must have privileges to run the following commands:

- CREATE TABLE
- CREATE SEQUENCE
- CREATE PROCEDURE
- CREATE TRIGGER

To run a project's test suite, the user usually needs these *additional* privileges:

- CREATE USER
- DROP USER
- CREATE TABLESPACE
- DROP TABLESPACE
- CREATE SESSION WITH ADMIN OPTION
- CREATE TABLE WITH ADMIN OPTION
- CREATE SEQUENCE WITH ADMIN OPTION
- CREATE PROCEDURE WITH ADMIN OPTION
- CREATE TRIGGER WITH ADMIN OPTION

Note that, while the RESOURCE role has the required CREATE TABLE, CREATE SEQUENCE, CREATE PROCEDURE and CREATE TRIGGER privileges, and a user granted RESOURCE WITH ADMIN OPTION can grant RESOURCE, such a user cannot grant the individual privileges (e.g. CREATE TABLE), and thus RESOURCE WITH ADMIN OPTION is not usually sufficient for running tests.

Some test suites also create views; to run these, the user also needs the CREATE VIEW WITH ADMIN OPTION privilege. In particular, this is needed for Django's own test suite.

All of these privileges are included in the DBA role, which is appropriate for use on a private developer's database.

The Oracle database backend uses the `SYS.DBMS_LOB` and `SYS.DBMS_RANDOM` packages, so your user will require execute permissions on it. It's normally accessible to all users by default, but in case it is not, you'll need to grant permissions like so:

```
GRANT EXECUTE ON SYS.DBMS_LOB TO user;
GRANT EXECUTE ON SYS.DBMS_RANDOM TO user;
```

Connecting to the database

To connect using the service name of your Oracle database, your `settings.py` file should look something like this:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.oracle',
        'NAME': 'xe',
        'USER': 'a_user',
        'PASSWORD': 'a_password',
        'HOST': '',
        'PORT': '',
    }
}
```

In this case, you should leave both *HOST* and *PORT* empty. However, if you don't use a `tnsnames.ora` file or a similar naming method and want to connect using the SID ("xe" in this example), then fill in both *HOST* and *PORT* like so:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.oracle',
        'NAME': 'xe',
        'USER': 'a_user',
        'PASSWORD': 'a_password',
        'HOST': 'dbprod01ned.mycompany.com',
        'PORT': '1540',
    }
}
```

You should either supply both *HOST* and *PORT*, or leave both as empty strings. Django will use a different connect descriptor depending on that choice.

Threaded option

If you plan to run Django in a multithreaded environment (e.g. Apache using the default MPM module on any modern operating system), then you **must** set the `threaded` option of your Oracle database configuration to True:

```
'OPTIONS': {
    'threaded': True,
},
```

Failure to do this may result in crashes and other odd behavior.

INSERT ... RETURNING INTO

By default, the Oracle backend uses a `RETURNING INTO` clause to efficiently retrieve the value of an `AutoField` when inserting new rows. This behavior may result in a `DatabaseError` in certain unusual setups, such as when inserting into a remote table, or into a view with an `INSTEAD OF` trigger. The `RETURNING INTO` clause can be disabled by setting the `use_returning_into` option of the database configuration to False:

```
'OPTIONS': {
    'use_returning_into': False,
},
```

In this case, the Oracle backend will use a separate `SELECT` query to retrieve AutoField values.

Naming issues

Oracle imposes a name length limit of 30 characters. To accommodate this, the backend truncates database identifiers to fit, replacing the final four characters of the truncated name with a repeatable MD5 hash value. Additionally, the backend turns database identifiers to all-uppercase.

To prevent these transformations (this is usually required only when dealing with legacy databases or accessing tables which belong to other users), use a quoted name as the value for `db_table`:

```
class LegacyModel(models.Model):
    class Meta:
        db_table = '"name_left_in_lowercase"'

class ForeignModel(models.Model):
    class Meta:
        db_table = '"OTHER_USER"."NAME_ONLY_SEEMS_OVER_30"'
```

Quoted names can also be used with Django's other supported database backends; except for Oracle, however, the quotes have no effect.

When running `migrate`, an `ORA-06552` error may be encountered if certain Oracle keywords are used as the name of a model field or the value of a `db_column` option. Django quotes all identifiers used in queries to prevent most such problems, but this error can still occur when an Oracle datatype is used as a column name. In particular, take care to avoid using the names `date`, `timestamp`, `number` or `float` as a field name.

NULL and empty strings

Django generally prefers to use the empty string ('') rather than NULL, but Oracle treats both identically. To get around this, the Oracle backend ignores an explicit `null` option on fields that have the empty string as a possible value and generates DDL as if `null=True`. When fetching from the database, it is assumed that a `NULL` value in one of these fields really means the empty string, and the data is silently converted to reflect this assumption.

`TextField` limitations

The Oracle backend stores `TextFields` as `NCLOB` columns. Oracle imposes some limitations on the usage of such LOB columns in general:

- LOB columns may not be used as primary keys.
- LOB columns may not be used in indexes.

- LOB columns may not be used in a `SELECT DISTINCT` list. This means that attempting to use the `QuerySet.distinct` method on a model that includes `TextField` columns will result in an `ORA-00932` error when run against Oracle. As a workaround, use the `QuerySet.defer` method in conjunction with `distinct()` to prevent `TextField` columns from being included in the `SELECT DISTINCT` list.

6.7.6 Using a 3rd-party database backend

In addition to the officially supported databases, there are backends provided by 3rd parties that allow you to use other databases with Django:

- SAP SQL Anywhere
- IBM DB2
- Microsoft SQL Server
- Firebird
- ODBC

The Django versions and ORM features supported by these unofficial backends vary considerably. Queries regarding the specific capabilities of these unofficial backends, along with any support queries, should be directed to the support channels provided by each 3rd party project.

6.8 `django-admin` and `manage.py`

`django-admin` is Django's command-line utility for administrative tasks. This document outlines all it can do.

In addition, `manage.py` is automatically created in each Django project. `manage.py` does the same thing as `django-admin` but takes care of a few things for you:

- It puts your project's package on `sys.path`.
- It sets the *`DJANGO_SETTINGS_MODULE`* environment variable so that it points to your project's `settings.py` file.

The `django-admin` script should be on your system path if you installed Django via its `setup.py` utility. If it's not on your path, you can find it in `site-packages/django/bin` within your Python installation. Consider symlinking it from some place on your path, such as `/usr/local/bin`.

For Windows users, who do not have symlinking functionality available, you can copy `django-admin.exe` to a location on your existing path or edit the `PATH` settings (under `Settings - Control Panel - System - Advanced - Environment...`) to point to its installed location.

Generally, when working on a single Django project, it's easier to use `manage.py` than `django-admin`. If you need to switch between multiple Django settings files, use `django-admin` with *`DJANGO_SETTINGS_MODULE`* or the *`--settings`* command line option.

The command-line examples throughout this document use `django-admin` to be consistent, but any example can use `manage.py` or `python -m django` just as well.

`python -m django` was added.

6.8.1 Usage

```
$ django-admin <command> [options]
$ manage.py <command> [options]
$ python -m django <command> [options]
```

`command` should be one of the commands listed in this document. `options`, which is optional, should be zero or more of the options available for the given command.

Getting runtime help

`django-admin help`

Run `django-admin help` to display usage information and a list of the commands provided by each application.

Run `django-admin help --commands` to display a list of all available commands.

Run `django-admin help <command>` to display a description of the given command and a list of its available options.

App names

Many commands take a list of "app names." An "app name" is the basename of the package containing your models. For example, if your *`INSTALLED_APPS`* contains the string `'mysite.blog'`, the app name is `blog`.

Determining the version

`django-admin version`

Run `django-admin version` to display the current Django version.

The output follows the schema described in **PEP 440**:

```
1.4.dev17026
1.4a1
1.4
```

Displaying debug output

Use *`--verbosity`* to specify the amount of notification and debug information that `django-admin` prints to the console.

6.8.2 Available commands

`check`

`django-admin check [app_label [app_label ...]]`

Uses the system check framework to inspect the entire Django project for common problems.

By default, all apps will be checked. You can check a subset of apps by providing a list of app labels as arguments:

```
django-admin check auth admin myapp
```

If you do not specify any app, all apps will be checked.

`--tag` `TAGS`**`, -t`** `TAGS`

The system check framework performs many different types of checks that are *categorized with tags*. You can use these tags to restrict the checks performed to just those in a particular category. For example, to perform only models and compatibility checks, run:

```
django-admin check --tag models --tag compatibility
```

--list-tags

Lists all available tags.

--deploy

Activates some additional checks that are only relevant in a deployment setting.

You can use this option in your local development environment, but since your local development settings module may not have many of your production settings, you will probably want to point the `check` command at a different settings module, either by setting the `DJANGO_SETTINGS_MODULE` environment variable, or by passing the `--settings` option:

```
django-admin check --deploy --settings=production_settings
```

Or you could run it directly on a production or staging deployment to verify that the correct settings are in use (omitting `--settings`). You could even make it part of your integration test suite.

--fail-level `{CRITICAL,ERROR,WARNING,INFO,DEBUG}`

Specifies the message level that will cause the command to exit with a non-zero status. Default is `ERROR`.

compilemessages

django-admin compilemessages

Compiles `.po` files created by *`makemessages`* to `.mo` files for use with the built-in gettext support. See Internationalization and localization.

--locale `LOCALE`, **-l** `LOCALE`

Specifies the locale(s) to process. If not provided, all locales are processed.

--exclude `EXCLUDE`, **-x** `EXCLUDE`

Specifies the locale(s) to exclude from processing. If not provided, no locales are excluded.

--use-fuzzy, -f

Includes fuzzy translations into compiled files.

`compilemessages` now matches the operation of *`makemessages`*, scanning the project tree for `.po` files to compile.

Example usage:

```
django-admin compilemessages --locale=pt_BR
django-admin compilemessages --locale=pt_BR --locale=fr -f
django-admin compilemessages -l pt_BR
django-admin compilemessages -l pt_BR -l fr --use-fuzzy
django-admin compilemessages --exclude=pt_BR
django-admin compilemessages --exclude=pt_BR --exclude=fr
django-admin compilemessages -x pt_BR
django-admin compilemessages -x pt_BR -x fr
```

createcachetable

django-admin createcachetable

Creates the cache tables for use with the database cache backend using the information from your settings file. See Django's cache framework for more information.

--database DATABASE

Specifies the database in which the cache table(s) will be created. Defaults to `default`.

--dry-run

Prints the SQL that would be run without actually running it, so you can customize it or use the migrations framework.

The `--dry-run` option was added.

dbshell

django-admin dbshell

Runs the command-line client for the database engine specified in your *ENGINE* setting, with the connection parameters specified in your *USER*, *PASSWORD*, etc., settings.

- For PostgreSQL, this runs the `psql` command-line client.
- For MySQL, this runs the `mysql` command-line client.
- For SQLite, this runs the `sqlite3` command-line client.
- For Oracle, this runs the `sqlplus` command-line client.

This command assumes the programs are on your `PATH` so that a simple call to the program name (`psql`, `mysql`, `sqlite3`, `sqlplus`) will find the program in the right place. There's no way to specify the location of the program manually.

--database DATABASE

Specifies the database onto which to open a shell. Defaults to `default`.

diffsettings

django-admin diffsettings

Displays differences between the current settings file and Django's default settings.

Settings that don't appear in the defaults are followed by `"###"`. For example, the default settings don't define *ROOT_URLCONF*, so *ROOT_URLCONF* is followed by `"###"` in the output of `diffsettings`.

--all

Displays all settings, even if they have Django's default value. Such settings are prefixed by `"###"`.

dumpdata

django-admin dumpdata [app_label[.ModelName] [app_label[.ModelName] ...]]

Outputs to standard output all data in the database associated with the named application(s).

If no application name is provided, all installed applications will be dumped.

The output of `dumpdata` can be used as input for *loaddata*.

Note that `dumpdata` uses the default manager on the model for selecting the records to dump. If you're using a *custom manager* as the default manager and it filters some of the available records, not all of the objects will be dumped.

`--all, -a`

Uses Django's base manager, dumping records which might otherwise be filtered or modified by a custom manager.

`--format` `FORMAT`

Specifies the serialization format of the output. Defaults to JSON. Supported formats are listed in *Serialization formats*.

`--indent` `INDENT`

Specifies the number of indentation spaces to use in the output. Defaults to `None` which displays all data on single line.

`--exclude` `EXCLUDE`, **`-e`** `EXCLUDE`

Prevents specific applications or models (specified in the form of `app_label.ModelName`) from being dumped. If you specify a model name, the output will be restricted to that model, rather than the entire application. You can also mix application names and model names.

If you want to exclude multiple applications, pass `--exclude` more than once:

```
django-admin dumpdata --exclude=auth --exclude=contenttypes
```

`--database` `DATABASE`

Specifies the database from which data will be dumped. Defaults to `default`.

`--natural-foreign`

Uses the `natural_key()` model method to serialize any foreign key and many-to-many relationship to objects of the type that defines the method. If you're dumping `contrib.auth Permission` objects or `contrib.contenttypes ContentType` objects, you should probably use this flag. See the *natural keys* documentation for more details on this and the next option.

`--natural-primary`

Omits the primary key in the serialized data of this object since it can be calculated during deserialization.

`--pks` `PRIMARY_KEYS`

Outputs only the objects specified by a comma separated list of primary keys. This is only available when dumping one model. By default, all the records of the model are output.

`--output` `OUTPUT`, **`-o`** `OUTPUT`

Specifies a file to write the serialized data to. By default, the data goes to standard output.

When this option is set and `--verbosity` is greater than 0 (the default), a progress bar is shown in the terminal.

The progress bar in the terminal was added.

flush

django-admin flush

Removes all data from the database and re-executes any post-synchronization handlers. The table of which migrations have been applied is not cleared.

If you would rather start from an empty database and re-run all migrations, you should drop and recreate the database and then run *migrate* instead.

--noinput, --no-input

Suppresses all user prompts.

The `--no-input` alias was added.

--database DATABASE

Specifies the database to flush. Defaults to `default`.

inspectdb

django-admin inspectdb [table [table ...]]

Introspects the database tables in the database pointed-to by the *NAME* setting and outputs a Django model module (a `models.py` file) to standard output. You may choose what tables to inspect by passing their names as arguments.

Use this if you have a legacy database with which you'd like to use Django. The script will inspect the database and create a model for each table within it.

As you might expect, the created models will have an attribute for every field in the table. Note that `inspectdb` has a few special cases in its field-name output:

- If `inspectdb` cannot map a column's type to a model field type, it'll use `TextField` and will insert the Python comment `'This field type is a guess.'` next to the field in the generated model.
- If the database column name is a Python reserved word (such as `'pass'`, `'class'` or `'for'`), `inspectdb` will append `'_field'` to the attribute name. For example, if a table has a column `'for'`, the generated model will have a field `'for_field'`, with the `db_column` attribute set to `'for'`. `inspectdb` will insert the Python comment `'Field renamed because it was a Python reserved word.'` next to the field.

This feature is meant as a shortcut, not as definitive model generation. After you run it, you'll want to look over the generated models yourself to make customizations. In particular, you'll need to rearrange models' order, so that models that refer to other models are ordered properly.

Primary keys are automatically introspected for PostgreSQL, MySQL and SQLite, in which case Django puts in the `primary_key=True` where needed.

`inspectdb` works with PostgreSQL, MySQL and SQLite. Foreign-key detection only works in PostgreSQL and with certain types of MySQL tables.

Django doesn't create database defaults when a *default* is specified on a model field. Similarly, database defaults aren't translated to model field defaults or detected in any fashion by `inspectdb`.

By default, `inspectdb` creates unmanaged models. That is, `managed = False` in the model's `Meta` class tells Django not to manage each table's creation, modification, and deletion. If you do want to allow Django to manage the table's lifecycle, you'll need to change the *managed* option to `True` (or simply remove it because `True` is its default value).

Support for the `table` argument(s) to choose what tables should be inspected was added.

--database DATABASE

Specifies the database to introspect. Defaults to `default`.

loaddata

django-admin loaddata fixture [fixture ...]

Searches for and loads the contents of the named fixture into the database.

--database DATABASE

Specifies the database into which the data will be loaded. Defaults to `default`.

--ignorenonexistent, -i

Ignores fields and models that may have been removed since the fixture was originally generated.

--app APP_LABEL

Specifies a single app to look for fixtures in rather than looking in all apps.

What's a "fixture"?

A *fixture* is a collection of files that contain the serialized contents of the database. Each fixture has a unique name, and the files that comprise the fixture can be distributed over multiple directories, in multiple applications.

Django will search in three locations for fixtures:

1. In the `fixtures` directory of every installed application
2. In any directory named in the *FIXTURE_DIRS* setting
3. In the literal path named by the fixture

Django will load any and all fixtures it finds in these locations that match the provided fixture names.

If the named fixture has a file extension, only fixtures of that type will be loaded. For example:

```
django-admin loaddata mydata.json
```

would only load JSON fixtures called `mydata`. The fixture extension must correspond to the registered name of a *serializer* (e.g., `json` or `xml`).

If you omit the extensions, Django will search all available fixture types for a matching fixture. For example:

```
django-admin loaddata mydata
```

would look for any fixture of any fixture type called `mydata`. If a fixture directory contained `mydata.json`, that fixture would be loaded as a JSON fixture.

The fixtures that are named can include directory components. These directories will be included in the search path. For example:

```
django-admin loaddata foo/bar/mydata.json
```

would search `<app_label>/fixtures/foo/bar/mydata.json` for each installed application, `<dirname>/foo/bar/mydata.json` for each directory in *FIXTURE_DIRS*, and the literal path `foo/bar/mydata.json`.

When fixture files are processed, the data is saved to the database as is. Model defined *save()* methods are not called, and any *pre_save* or *post_save* signals will be called with `raw=True` since the instance only contains attributes that are local to the model. You may, for example, want to disable handlers that access related fields that aren't present during fixture loading and would otherwise raise an exception:

```
from django.db.models.signals import post_save
from .models import MyModel

def my_handler(**kwargs):
    # disable the handler during fixture loading
    if kwargs['raw']:
        return
    ...
```

```
post_save.connect(my_handler, sender=MyModel)
```

You could also write a simple decorator to encapsulate this logic:

```
from functools import wraps

def disable_for_loaddata(signal_handler):
    """
    Decorator that turns off signal handlers when loading fixture data.
    """
    @wraps(signal_handler)
    def wrapper(*args, **kwargs):
        if kwargs['raw']:
            return
        signal_handler(*args, **kwargs)
    return wrapper

@disable_for_loaddata
def my_handler(**kwargs):
    ...
```

Just be aware that this logic will disable the signals whenever fixtures are deserialized, not just during `loaddata`.

Note that the order in which fixture files are processed is undefined. However, all fixture data is installed as a single transaction, so data in one fixture can reference data in another fixture. If the database backend supports row-level constraints, these constraints will be checked at the end of the transaction.

The *dumpdata* command can be used to generate input for `loaddata`.

Compressed fixtures

Fixtures may be compressed in `zip`, `gz`, or `bz2` format. For example:

```
django-admin loaddata mydata.json
```

would look for any of `mydata.json`, `mydata.json.zip`, `mydata.json.gz`, or `mydata.json.bz2`. The first file contained within a zip-compressed archive is used.

Note that if two fixtures with the same name but different fixture type are discovered (for example, if `mydata.json` and `mydata.xml.gz` were found in the same fixture directory), fixture installation will be aborted, and any data installed in the call to `loaddata` will be removed from the database.

MySQL with MyISAM and fixtures

The MyISAM storage engine of MySQL doesn't support transactions or constraints, so if you use MyISAM, you won't get validation of fixture data, or a rollback if multiple transaction files are found.

Database-specific fixtures

If you're in a multi-database setup, you might have fixture data that you want to load onto one database, but not onto another. In this situation, you can add a database identifier into the names of your fixtures.

For example, if your *DATABASES* setting has a 'master' database defined, name the fixture `mydata.master.json` or `mydata.master.json.gz` and the fixture will only be loaded when you specify you want to load data into the `master` database.

makemessages

django-admin makemessages

Runs over the entire source tree of the current directory and pulls out all strings marked for translation. It creates (or updates) a message file in the conf/locale (in the Django tree) or locale (for project and application) directory. After making changes to the messages files you need to compile them with *compilemessages* for use with the builtin gettext support. See the *i18n documentation* for details.

--all, -a

Updates the message files for all available languages.

--extension EXTENSIONS, **-e** EXTENSIONS

Specifies a list of file extensions to examine (default: `html`, `txt`, `py` or `js` if *--domain* is `js`).

Example usage:

```
django-admin makemessages --locale=de --extension xhtml
```

Separate multiple extensions with commas or use `-e` or `--extension` multiple times:

```
django-admin makemessages --locale=de --extension=html,txt --extension xml
```

--locale LOCALE, **-l** LOCALE

Specifies the locale(s) to process.

--exclude EXCLUDE, **-x** EXCLUDE

Specifies the locale(s) to exclude from processing. If not provided, no locales are excluded.

Example usage:

```
django-admin makemessages --locale=pt_BR
django-admin makemessages --locale=pt_BR --locale=fr
django-admin makemessages -l pt_BR
django-admin makemessages -l pt_BR -l fr
django-admin makemessages --exclude=pt_BR
django-admin makemessages --exclude=pt_BR --exclude=fr
django-admin makemessages -x pt_BR
django-admin makemessages -x pt_BR -x fr
```

--domain DOMAIN, **-d** DOMAIN

Specifies the domain of the messages files. Supported options are:

- `django` for all `*.py`, `*.html` and `*.txt` files (default)
- `djangojs` for `*.js` files

--symlinks, -s

Follows symlinks to directories when looking for new translation strings.

Example usage:

```
django-admin makemessages --locale=de --symlinks
```

--ignore PATTERN, **-i** PATTERN

Ignores files or directories matching the given `glob`-style pattern. Use multiple times to ignore more.

These patterns are used by default: `'CVS'`, `'.*'`, `'*~'`, `'*.pyc'`.

Example usage:

```
django-admin makemessages --locale=en_US --ignore=apps/* --ignore=secret/*.html
```

--no-default-ignore

Disables the default values of `--ignore`.

--no-wrap

Disables breaking long message lines into several lines in language files.

--no-location

Suppresses writing '`#: filename:line`' comment lines in language files. Using this option makes it harder for technically skilled translators to understand each message's context.

--keep-pot

Prevents deleting the temporary `.pot` files generated before creating the `.po` file. This is useful for debugging errors which may prevent the final language files from being created.

:

See *Customizing the makemessages command* for instructions on how to customize the keywords that *makemessages* passes to `xgettext`.

makemigrations

django-admin makemigrations [app_label [app_label ...]]

Creates new migrations based on the changes detected to your models. Migrations, their relationship with apps and more are covered in depth in the migrations documentation.

Providing one or more app names as arguments will limit the migrations created to the app(s) specified and any dependencies needed (the table at the other end of a `ForeignKey`, for example).

--noinput, --no-input

Suppresses all user prompts. If a suppressed prompt cannot be resolved automatically, the command will exit with error code 3.

The `--no-input` alias was added.

--empty

Outputs an empty migration for the specified apps, for manual editing. This is for advanced users and should not be used unless you are familiar with the migration format, migration operations, and the dependencies between your migrations.

--dry-run

Shows what migrations would be made without actually writing any migrations files to disk. Using this option along with `--verbosity 3` will also show the complete migrations files that would be written.

--merge

Enables fixing of migration conflicts.

--name `NAME`**, -n** `NAME`

Allows naming the generated migration(s) instead of using a generated name.

--exit, -e

1.10 : Use the `--check` option instead.

Makes `makemigrations` exit with error code 1 when no migrations are created (or would have been created, if combined with `--dry-run`).

`--check`

Makes `makemigrations` exit with a non-zero status when model changes without migrations are detected.

migrate

`django-admin migrate [app_label] [migration_name]`

Synchronizes the database state with the current set of models and migrations. Migrations, their relationship with apps and more are covered in depth in the migrations documentation.

The behavior of this command changes depending on the arguments provided:

- No arguments: All apps have all of their migrations run.
- `<app_label>`: The specified app has its migrations run, up to the most recent migration. This may involve running other apps' migrations too, due to dependencies.
- `<app_label> <migrationname>`: Brings the database schema to a state where the named migration is applied, but no later migrations in the same app are applied. This may involve unapplying migrations if you have previously migrated past the named migration. Use the name `zero` to unapply all migrations for an app.

`--database` `DATABASE`

Specifies the database to migrate. Defaults to `default`.

`--fake`

Tells Django to mark the migrations as having been applied or unapplied, but without actually running the SQL to change your database schema.

This is intended for advanced users to manipulate the current migration state directly if they're manually applying changes; be warned that using `--fake` runs the risk of putting the migration state table into a state where manual recovery will be needed to make migrations run correctly.

`--fake-initial`

Allows Django to skip an app's initial migration if all database tables with the names of all models created by all *`CreateModel`* operations in that migration already exist. This option is intended for use when first running migrations against a database that preexisted the use of migrations. This option does not, however, check for matching database schema beyond matching table names and so is only safe to use if you are confident that your existing schema matches what is recorded in your initial migration.

`--run-syncdb`

Allows creating tables for apps without migrations. While this isn't recommended, the migrations framework is sometimes too slow on large projects with hundreds of models.

`--noinput, --no-input`

Suppresses all user prompts. An example prompt is asking about removing stale content types.

The `--no-input` alias was added.

runserver

`django-admin runserver [addrport]`

Starts a lightweight development Web server on the local machine. By default, the server runs on port 8000 on the IP address `127.0.0.1`. You can pass in an IP address and port number explicitly.

If you run this script as a user with normal privileges (recommended), you might not have access to start a port on a low port number. Low port numbers are reserved for the superuser (root).

This server uses the WSGI application object specified by the *WSGI_APPLICATION* setting.

DO NOT USE THIS SERVER IN A PRODUCTION SETTING. It has not gone through security audits or performance tests. (And that's how it's gonna stay. We're in the business of making Web frameworks, not Web servers, so improving this server to be able to handle a production environment is outside the scope of Django.)

The development server automatically reloads Python code for each request, as needed. You don't need to restart the server for code changes to take effect. However, some actions like adding files don't trigger a restart, so you'll have to restart the server in these cases.

If you are using Linux and install pyinotify, kernel signals will be used to autoreload the server (rather than polling file modification timestamps each second). This offers better scaling to large projects, reduction in response time to code modification, more robust change detection, and battery usage reduction.

When you start the server, and each time you change Python code while the server is running, the system check framework will check your entire Django project for some common errors (see the *check* command). If any errors are found, they will be printed to standard output.

You can run as many concurrent servers as you want, as long as they're on separate ports. Just execute `django-admin runserver` more than once.

Note that the default IP address, `127.0.0.1`, is not accessible from other machines on your network. To make your development server viewable to other machines on the network, use its own IP address (e.g. `192.168.2.1`) or `0.0.0.0` or `::` (with IPv6 enabled).

You can provide an IPv6 address surrounded by brackets (e.g. `[200a::1]:8000`). This will automatically enable IPv6 support.

A hostname containing ASCII-only characters can also be used.

If the staticfiles contrib app is enabled (default in new projects) the *runserver* command will be overridden with its own *runserver* command.

If *migrate* was not previously executed, the table that stores the history of migrations is created at first run of `runserver`.

Logging of each request and response of the server is sent to the *django.server* logger.

In older versions, log messages were written to `sys.stderr` instead of being handled through Python logging.

`--noreload`

Disables the auto-reloader. This means any Python code changes you make while the server is running will *not* take effect if the particular Python modules have already been loaded into memory.

`--nothreading`

Disables use of threading in the development server. The server is multithreaded by default.

`--ipv6, -6`

Uses IPv6 for the development server. This changes the default IP address from `127.0.0.1` to `::1`.

Examples of using different ports and addresses

Port 8000 on IP address `127.0.0.1`:

```
django-admin runserver
```

Port 8000 on IP address `1.2.3.4`:

```
django-admin runserver 1.2.3.4:8000
```

Port 7000 on IP address `127.0.0.1`:

```
django-admin runserver 7000
```

Port 7000 on IP address `1.2.3.4`:

```
django-admin runserver 1.2.3.4:7000
```

Port 8000 on IPv6 address `::1`:

```
django-admin runserver -6
```

Port 7000 on IPv6 address `::1`:

```
django-admin runserver -6 7000
```

Port 7000 on IPv6 address `2001:0db8:1234:5678::9`:

```
django-admin runserver [2001:0db8:1234:5678::9]:7000
```

Port 8000 on IPv4 address of host `localhost`:

```
django-admin runserver localhost:8000
```

Port 8000 on IPv6 address of host `localhost`:

```
django-admin runserver -6 localhost:8000
```

Serving static files with the development server

By default, the development server doesn't serve any static files for your site (such as CSS files, images, things under *MEDIA_URL* and so forth). If you want to configure Django to serve static media, read Managing static files (e.g. images, JavaScript, CSS).

sendtestemail

django-admin sendtestemail [email [email ...]]

Sends a test email (to confirm email sending through Django is working) to the recipient(s) specified. For example:

```
django-admin sendtestemail foo@example.com bar@example.com
```

There are a couple of options, and you may use any combination of them together:

--managers

Mails the email addresses specified in *MANAGERS* using *mail_managers()*.

--admins

Mails the email addresses specified in *ADMINS* using *mail_admins()*.

shell

django-admin shell

Starts the Python interactive interpreter.

--interface {ipython,bpython,python}, **-i** {ipython,bpython,python}

Specifies the shell to use. By default, Django will use IPython or bpython if either is installed. If both are installed, specify which one you want like so:

IPython:

```
django-admin shell -i ipython
```

bpython:

```
django-admin shell -i bpython
```

If you have a "rich" shell installed but want to force use of the "plain" Python interpreter, use `python` as the interface name, like so:

```
django-admin shell -i python
```

1.10 : In older versions, use the `--plain` option instead of `-i python`. This is deprecated and will be removed in Django 2.0.

--nostartup

Disables reading the startup script for the "plain" Python interpreter. By default, the script pointed to by the `PYTHONSTARTUP` environment variable or the `~/.pythonrc.py` script is read.

--command COMMAND, **-c** COMMAND

Lets you pass a command as a string to execute it as Django, like so:

```
django-admin shell --command="import django; print(django.__version__)"
```

showmigrations

django-admin showmigrations [app_label [app_label ...]]

Shows all migrations in a project. You can choose from one of two formats:

--list, -l

Lists all of the apps Django knows about, the migrations available for each app, and whether or not each migration is applied (marked by an `[X]` next to the migration name).

Apps without migrations are also listed, but have `(no migrations)` printed under them.

This is the default output format.

--plan, -p

Shows the migration plan Django will follow to apply migrations. Any supplied app labels are ignored because the plan might go beyond those apps. Like `--list`, applied migrations are marked by an `[X]`. For a `--verbosity` of 2 and above, all dependencies of a migration will also be shown.

--database DATABASE

Specifies the database to examine. Defaults to `default`.

sqlflush

django-admin sqlflush

Prints the SQL statements that would be executed for the *flush* command.

--database DATABASE

Specifies the database for which to print the SQL. Defaults to `default`.

sqlmigrate

django-admin sqlmigrate app_label migration_name

Prints the SQL for the named migration. This requires an active database connection, which it will use to resolve constraint names; this means you must generate the SQL against a copy of the database you wish to later apply it on.

Note that `sqlmigrate` doesn't colorize its output.

--backwards

Generates the SQL for unapplying the migration. By default, the SQL created is for running the migration in the forwards direction.

--database DATABASE

Specifies the database for which to generate the SQL. Defaults to `default`.

To increase the readability of the overall SQL output the SQL code generated for each migration operation is preceded by the operation's description.

sqlsequencereset

django-admin sqlsequencereset app_label [app_label ...]

Prints the SQL statements for resetting sequences for the given app name(s).

Sequences are indexes used by some database engines to track the next available number for automatically incremented fields.

Use this command to generate SQL which will fix cases where a sequence is out of sync with its automatically incremented field data.

--database DATABASE

Specifies the database for which to print the SQL. Defaults to `default`.

squashmigrations

django-admin squashmigrations app_label [start_migration_name] migration_name

Squashes the migrations for `app_label` up to and including `migration_name` down into fewer migrations, if possible. The resulting squashed migrations can live alongside the unsquashed ones safely. For more information, please read *Squashing migrations*.

When `start_migration_name` is given, Django will only include migrations starting from and including this migration. This helps to mitigate the squashing limitation of *RunPython* and *django.db.migrations.operations.RunSQL* migration operations.

--no-optimize

Disables the optimizer when generating a squashed migration. By default, Django will try to optimize the operations in your migrations to reduce the size of the resulting file. Use this option if this process is failing or creating incorrect migrations, though please also file a Django bug report about the behavior, as optimization is meant to be safe.

--noinput, --no-input

Suppresses all user prompts.

The `--no-input` alias was added.

startapp

django-admin startapp name [directory]

Creates a Django app directory structure for the given app name in the current directory or the given destination.

By default the directory created contains a `models.py` file and other app template files. (See the source for more details.) If only the app name is given, the app directory will be created in the current working directory.

If the optional destination is provided, Django will use that existing directory rather than creating a new one. You can use '.' to denote the current working directory.

For example:

```
django-admin startapp myapp /Users/jezdez/Code/myapp
```

--template TEMPLATE

Provides the path to a directory with a custom app template file or a path to a compressed file (`.tar.gz`, `.tar.bz2`, `.tgz`, `.tbz`, `.zip`) containing the app template files.

For example, this would look for an app template in the given directory when creating the `myapp` app:

```
django-admin startapp --template=/Users/jezdez/Code/my_app_template myapp
```

Django will also accept URLs (`http`, `https`, `ftp`) to compressed archives with the app template files, downloading and extracting them on the fly.

For example, taking advantage of GitHub's feature to expose repositories as zip files, you can use a URL like:

```
django-admin startapp --template=https://github.com/githubuser/django-app-template/archive/master.zip
```

--extension EXTENSIONS, **-e** EXTENSIONS

Specifies which file extensions in the app template should be rendered with the template engine. Defaults to `py`.

--name FILES, **-n** FILES

Specifies which files in the app template (in addition to those matching `--extension`) should be rendered with the template engine. Defaults to an empty list.

The *template context* used for all matching files is:

- Any option passed to the `startapp` command (among the command's supported options)
- `app_name` – the app name as passed to the command
- `app_directory` – the full path of the newly created app
- `camel_case_app_name` – the app name in camel case format
- `docs_version` – the version of the documentation: `'dev'` or `'1.x'`

camel_case_app_name was added.

: When the app template files are rendered with the Django template engine (by defa
will also replace all stray template variables contained. For example, if one of the Pytho
explaining a particular feature related to template rendering, it might result in an incorr
To work around this problem, you can use the *templatetag* templatetag to "escap
template syntax.
In addition, to allow Python template files that contain Django template language sy
packaging systems from trying to byte-compile invalid `*.py` files, template files end
renamed to `.py`.
Renaming of `.py-tpl` to `.py` was added.

startproject

django-admin startproject name [directory]

Creates a Django project directory structure for the given project name in the current directory or the given destination.

By default, the new directory contains `manage.py` and a project package (containing a `settings.py` and other files). See the template source for details.

If only the project name is given, both the project directory and project package will be named `<projectname>` and the project directory will be created in the current working directory.

If the optional destination is provided, Django will use that existing directory as the project directory, and create `manage.py` and the project package within it. Use '.' to denote the current working directory.

For example:

```
django-admin startproject myproject /Users/jezdez/Code/myproject_repo
```

--template TEMPLATE

Specifies a directory, file path, or URL of a custom project template. See the *startapp --template* documentation for examples and usage.

--extension EXTENSIONS, **-e** EXTENSIONS

Specifies which file extensions in the project template should be rendered with the template engine. Defaults to `py`.

--name FILES, **-n** FILES

Specifies which files in the project template (in addition to those matching `--extension`) should be rendered with the template engine. Defaults to an empty list.

The *template context* used is:

- Any option passed to the `startproject` command (among the command's supported options)
- `project_name` – the project name as passed to the command
- `project_directory` – the full path of the newly created project
- `secret_key` – a random key for the *SECRET_KEY* setting
- `docs_version` – the version of the documentation: `'dev'` or `'1.x'`

Please also see the *rendering warning* as mentioned for *startapp*.

test

django-admin test [test_label [test_label ...]]

Runs tests for all installed apps. See Testing in Django for more information.

--failfast

Stops running tests and reports the failure immediately after a test fails.

--testrunner TESTRUNNER

Controls the test runner class that is used to execute tests. This value overrides the value provided by the *TEST_RUNNER* setting.

--liveserver LIVESERVER

Overrides the default address where the live server (used with *LiveServerTestCase*) is expected to run from. The default value is `localhost:8081-8179`.

In earlier versions, the default value was `localhost:8081`.

--noinput, --no-input

Suppresses all user prompts. A typical prompt is a warning about deleting an existing test database.

The `--no-input` alias was added.

Test runner options

The `test` command receives options on behalf of the specified *--testrunner*. These are the options of the default test runner: *DiscoverRunner*.

--keepdb, -k

Preserves the test database between test runs. This has the advantage of skipping both the create and destroy actions which can greatly decrease the time to run tests, especially those in a large test suite. If the test database does not exist, it will be created on the first run and then preserved for each subsequent run. Any unapplied migrations will also be applied to the test database before running the test suite.

--reverse, -r

Sorts test cases in the opposite execution order. This may help in debugging the side effects of tests that aren't properly isolated. *Grouping by test class* is preserved when using this option.

--debug-sql, -d

Enables *SQL logging* for failing tests. If `--verbosity` is `2`, then queries in passing tests are also output.

--parallel [N]

Runs tests in separate parallel processes. Since modern processors have multiple cores, this allows running tests significantly faster.

By default `--parallel` runs one process per core according to `multiprocessing.cpu_count()`. You can adjust the number of processes either by providing it as the option's value, e.g. `--parallel=4`, or by setting the `DJANGO_TEST_PROCESSES` environment variable.

Django distributes test cases — `unittest.TestCase` subclasses — to subprocesses. If there are fewer test cases than configured processes, Django will reduce the number of processes accordingly.

Each process gets its own database. You must ensure that different test cases don't access the same resources. For instance, test cases that touch the filesystem should create a temporary directory for their own use.

This option requires the third-party `tblib` package to display tracebacks correctly:

```
$ pip install tblib
```

This feature isn't available on Windows. It doesn't work with the Oracle database backend either.

If you want to use `pdb` while debugging tests, you must disable parallel execution (`--parallel=1`). You'll see something like `bdb.BdbQuit` if you don't.

: When test parallelization is enabled and a test fails, Django may be unable to display the exception traceback. This can make debugging difficult. If you encounter this problem, run the affected test without parallelization to see the traceback of the failure.
This is a known limitation. It arises from the need to serialize objects in order to exchange them between processes. See What can be pickled and unpickled? for details.

--tag TAGS

Runs only tests *marked with the specified tags*. May be specified multiple times and combined with *test --exclude-tag*.

--exclude-tag EXCLUDE_TAGS

Excludes tests *marked with the specified tags*. May be specified multiple times and combined with *test --tag*.

testserver

django-admin testserver [fixture [fixture ...]]

Runs a Django development server (as in *runserver*) using data from the given fixture(s).

For example, this command:

```
django-admin testserver mydata.json
```

...would perform the following steps:

1. Create a test database, as described in *The test database*.
2. Populate the test database with fixture data from the given fixtures. (For more on fixtures, see the documentation for *loaddata* above.)
3. Runs the Django development server (as in *runserver*), pointed at this newly created test database instead of your production database.

This is useful in a number of ways:

- When you're writing unit tests of how your views act with certain fixture data, you can use `testserver` to interact with the views in a Web browser, manually.
- Let's say you're developing your Django application and have a "pristine" copy of a database that you'd like to interact with. You can dump your database to a fixture (using the *dumpdata* command, explained above), then use `testserver` to run your Web application with that data. With this arrangement, you have the flexibility of messing up your data in any way, knowing that whatever data changes you're making are only being made to a test database.

Note that this server does *not* automatically detect changes to your Python source code (as *runserver* does). It does, however, detect changes to templates.

--addrport ADDRPORT

Specifies a different port, or IP address and port, from the default of `127.0.0.1:8000`. This value follows exactly the same format and serves exactly the same function as the argument to the *runserver* command.

Examples:

To run the test server on port 7000 with `fixture1` and `fixture2`:

```
django-admin testserver --addrport 7000 fixture1 fixture2
django-admin testserver fixture1 fixture2 --addrport 7000
```

(The above statements are equivalent. We include both of them to demonstrate that it doesn't matter whether the options come before or after the fixture arguments.)

To run on 1.2.3.4:7000 with a `test` fixture:

```
django-admin testserver --addrport 1.2.3.4:7000 test
```

`--noinput, --no-input`

Suppresses all user prompts. A typical prompt is a warning about deleting an existing test database.

The `--no-input` alias was added.

6.8.3 Commands provided by applications

Some commands are only available when the `django.contrib` application that implements them has been *enabled*. This section describes them grouped by their application.

`django.contrib.auth`

`changepassword`

`django-admin changepassword [<username>]`

This command is only available if Django's authentication system (`django.contrib.auth`) is installed.

Allows changing a user's password. It prompts you to enter a new password twice for the given user. If the entries are identical, this immediately becomes the new password. If you do not supply a user, the command will attempt to change the password whose username matches the current user.

`--database` `DATABASE`

Specifies the database to query for the user. Defaults to `default`.

Example usage:

```
django-admin changepassword ringo
```

`createsuperuser`

`django-admin createsuperuser`

This command is only available if Django's authentication system (`django.contrib.auth`) is installed.

Creates a superuser account (a user who has all permissions). This is useful if you need to create an initial superuser account or if you need to programmatically generate superuser accounts for your site(s).

When run interactively, this command will prompt for a password for the new superuser account. When run non-interactively, no password will be set, and the superuser account will not be able to log in until a password has been manually set for it.

`--username` `USERNAME`

`--email` `EMAIL`

The username and email address for the new account can be supplied by using the `--username` and `--email` arguments on the command line. If either of those is not supplied, `createsuperuser` will prompt for it when running interactively.

`--database` `DATABASE`

Specifies the database into which the superuser object will be saved.

You can subclass the management command and override `get_input_data()` if you want to customize data input and validation. Consult the source code for details on the existing implementation and the method's parameters. For example, it could be useful if you have a `ForeignKey` in *`REQUIRED_FIELDS`* and want to allow creating an instance instead of entering the primary key of an existing instance.

`django.contrib.gis`

`ogrinspect`

This command is only available if GeoDjango (`django.contrib.gis`) is installed.

Please refer to its *`description`* in the GeoDjango documentation.

`django.contrib.sessions`

`clearsessions`

`django-admin clearsessions`

Can be run as a cron job or directly to clean out expired sessions.

`django.contrib.sitemaps`

`ping_google`

This command is only available if the Sitemaps framework (`django.contrib.sitemaps`) is installed.

Please refer to its *`description`* in the Sitemaps documentation.

`django.contrib.staticfiles`

`collectstatic`

This command is only available if the static files application (`django.contrib.staticfiles`) is installed.

Please refer to its *`description`* in the staticfiles documentation.

`findstatic`

This command is only available if the static files application (`django.contrib.staticfiles`) is installed.

Please refer to its *`description`* in the staticfiles documentation.

6.8.4 Default options

Although some commands may allow their own custom options, every command allows for the following options:

--pythonpath PYTHONPATH

Adds the given filesystem path to the Python import search path. If this isn't provided, `django-admin` will use the `PYTHONPATH` environment variable.

This option is unnecessary in `manage.py`, because it takes care of setting the Python path for you.

Example usage:

```
django-admin migrate --pythonpath='/home/djangoprojects/myproject'
```

--settings SETTINGS

Specifies the settings module to use. The settings module should be in Python package syntax, e.g. `mysite.settings`. If this isn't provided, `django-admin` will use the `DJANGO_SETTINGS_MODULE` environment variable.

This option is unnecessary in `manage.py`, because it uses `settings.py` from the current project by default.

Example usage:

```
django-admin migrate --settings=mysite.settings
```

--traceback

Displays a full stack trace when a *`CommandError`* is raised. By default, `django-admin` will show a simple error message when a `CommandError` occurs and a full stack trace for any other exception.

Example usage:

```
django-admin migrate --traceback
```

--verbosity {0,1,2,3}, **--v** {0,1,2,3}

Specifies the amount of notification and debug information that a command should print to the console.

- `0` means no output.
- `1` means normal output (default).
- `2` means verbose output.
- `3` means *very* verbose output.

Example usage:

```
django-admin migrate --verbosity 2
```

--no-color

Disables colorized command output. Some commands format their output to be colorized. For example, errors will be printed to the console in red and SQL statements will be syntax highlighted.

Example usage:

```
django-admin runserver --no-color
```

6.8.5 Extra niceties

Syntax coloring

The `django-admin` / `manage.py` commands will use pretty color-coded output if your terminal supports ANSI-colored output. It won't use the color codes if you're piping the command's output to another program.

Under Windows, the native console doesn't support ANSI escape sequences so by default there is no color output. But you can install the ANSICON third-party tool, the Django commands will detect its presence and will make use of its services to color output just like on Unix-based platforms.

The colors used for syntax highlighting can be customized. Django ships with three color palettes:

- `dark`, suited to terminals that show white text on a black background. This is the default palette.
- `light`, suited to terminals that show black text on a white background.
- `nocolor`, which disables syntax highlighting.

You select a palette by setting a `DJANGO_COLORS` environment variable to specify the palette you want to use. For example, to specify the `light` palette under a Unix or OS/X BASH shell, you would run the following at a command prompt:

```
export DJANGO_COLORS="light"
```

You can also customize the colors that are used. Django specifies a number of roles in which color is used:

- `error` - A major error.
- `notice` - A minor error.
- `success` - A success.
- `warning` - A warning.
- `sql_field` - The name of a model field in SQL.
- `sql_coltype` - The type of a model field in SQL.
- `sql_keyword` - An SQL keyword.
- `sql_table` - The name of a model in SQL.
- `http_info` - A 1XX HTTP Informational server response.
- `http_success` - A 2XX HTTP Success server response.
- `http_not_modified` - A 304 HTTP Not Modified server response.
- `http_redirect` - A 3XX HTTP Redirect server response other than 304.
- `http_not_found` - A 404 HTTP Not Found server response.
- `http_bad_request` - A 4XX HTTP Bad Request server response other than 404.
- `http_server_error` - A 5XX HTTP Server Error response.
- `migrate_heading` - A heading in a migrations management command.
- `migrate_label` - A migration name.

`success` was added.

Each of these roles can be assigned a specific foreground and background color, from the following list:

- `black`
- `red`

- green
- yellow
- blue
- magenta
- cyan
- white

Each of these colors can then be modified by using the following display options:

- bold
- underscore
- blink
- reverse
- conceal

A color specification follows one of the following patterns:

- role=fg
- role=fg/bg
- role=fg,option,option
- role=fg/bg,option,option

where role is the name of a valid color role, fg is the foreground color, bg is the background color and each option is one of the color modifying options. Multiple color specifications are then separated by a semicolon. For example:

```
export DJANGO_COLORS="error=yellow/blue,blink;notice=magenta"
```

would specify that errors be displayed using blinking yellow on blue, and notices displayed using magenta. All other color roles would be left uncolored.

Colors can also be specified by extending a base palette. If you put a palette name in a color specification, all the colors implied by that palette will be loaded. So:

```
export DJANGO_COLORS="light;error=yellow/blue,blink;notice=magenta"
```

would specify the use of all the colors in the light color palette, *except* for the colors for errors and notices which would be overridden as specified.

Bash completion

If you use the Bash shell, consider installing the Django bash completion script, which lives in extras/django_bash_completion in the Django distribution. It enables tab-completion of django-admin and manage.py commands, so you can, for instance...

- Type django-admin.
- Press [TAB] to see all available options.
- Type sql, then [TAB], to see all available options whose names start with sql.

See Writing custom django-admin commands for how to add customized actions.

6.9 Running management commands from your code

`django.core.management.`**`call_command`**(*name*, **args*, ***options*)

To call a management command from code use `call_command`.

`name` the name of the command to call or a command object. Passing the name is preferred unless the object is required for testing.

`*args` a list of arguments accepted by the command. Arguments are passed to the argument parser, so you can use the same style as you would on the command line. For example, `call_command('flush', 'verbosity=0')`.

`**options` named options accepted on the command-line. Options are passed to the command without triggering the argument parser, which means you'll need to pass the correct type. For example, `call_command('flush', verbosity=0)` (zero must be an integer rather than a string).

Examples:

```
from django.core import management
from django.core.management.commands import loaddata

management.call_command('flush', verbosity=0, interactive=False)
management.call_command('loaddata', 'test_data', verbosity=0)
management.call_command(loaddata.Command(), 'test_data', verbosity=0)
```

Note that command options that take no arguments are passed as keywords with `True` or `False`, as you can see with the `interactive` option above.

Named arguments can be passed by using either one of the following syntaxes:

```
# Similar to the command line
management.call_command('dumpdata', '--natural-foreign')

# Named argument similar to the command line minus the initial dashes and
# with internal dashes replaced by underscores
management.call_command('dumpdata', natural_foreign=True)

# `use_natural_foreign_keys` is the option destination variable
management.call_command('dumpdata', use_natural_foreign_keys=True)
```

Command options which take multiple options are passed a list:

```
management.call_command('dumpdata', exclude=['contenttypes', 'auth'])
```

The return value of the `call_command()` function is the same as the return value of the `handle()` method of the command.

`call_command()` now returns the value received from the `command.handle()` method. It now also accepts a command object as the first argument.

6.9.1 Output redirection

Note that you can redirect standard output and error streams as all commands support the `stdout` and `stderr` options. For example, you could write:

```
with open('/path/to/command_output') as f:
    management.call_command('dumpdata', stdout=f)
```

6.10 Django Exceptions

Django raises some of its own exceptions as well as standard Python exceptions.

6.10.1 Django Core Exceptions

Django core exception classes are defined in `django.core.exceptions`.

AppRegistryNotReady

exception AppRegistryNotReady

This exception is raised when attempting to use models before the *app loading process*, which initializes the ORM, is complete.

ObjectDoesNotExist

exception ObjectDoesNotExist

The base class for *DoesNotExist* exceptions; a `try/except` for `ObjectDoesNotExist` will catch *DoesNotExist* exceptions for all models.

See *get()* for further information on *ObjectDoesNotExist* and *DoesNotExist*.

FieldDoesNotExist

exception FieldDoesNotExist

The `FieldDoesNotExist` exception is raised by a model's `_meta.get_field()` method when the requested field does not exist on the model or on the model's parents.

MultipleObjectsReturned

exception MultipleObjectsReturned

The *MultipleObjectsReturned* exception is raised by a query if only one object is expected, but multiple objects are returned. A base version of this exception is provided in *django.core.exceptions*; each model class contains a subclassed version that can be used to identify the specific object type that has returned multiple objects.

See *get()* for further information.

SuspiciousOperation

exception SuspiciousOperation

The *SuspiciousOperation* exception is raised when a user has performed an operation that should be considered suspicious from a security perspective, such as tampering with a session cookie. Subclasses of `SuspiciousOperation` include:

- `DisallowedHost`
- `DisallowedModelAdminLookup`
- `DisallowedModelAdminToField`
- `DisallowedRedirect`

- `InvalidSessionKey`
- `RequestDataTooBig`
- `SuspiciousFileOperation`
- `SuspiciousMultipartForm`
- `SuspiciousSession`
- `TooManyFieldsSent`

If a `SuspiciousOperation` exception reaches the WSGI handler level it is logged at the `Error` level and results in a *`HttpResponseBadRequest`*. See the logging documentation for more information.

PermissionDenied

exception PermissionDenied

The *`PermissionDenied`* exception is raised when a user does not have permission to perform the action requested.

ViewDoesNotExist

exception ViewDoesNotExist

The *`ViewDoesNotExist`* exception is raised by *`django.urls`* when a requested view does not exist.

MiddlewareNotUsed

exception MiddlewareNotUsed

The *`MiddlewareNotUsed`* exception is raised when a middleware is not used in the server configuration.

ImproperlyConfigured

exception ImproperlyConfigured

The *`ImproperlyConfigured`* exception is raised when Django is somehow improperly configured – for example, if a value in `settings.py` is incorrect or unparseable.

FieldError

exception FieldError

The *`FieldError`* exception is raised when there is a problem with a model field. This can happen for several reasons:

- A field in a model clashes with a field of the same name from an abstract base class
- An infinite loop is caused by ordering
- A keyword cannot be parsed from the filter parameters
- A field cannot be determined from a keyword in the query parameters
- A join is not permitted on the specified field
- A field name is invalid
- A query contains invalid order_by arguments

ValidationError

exception ValidationError

The *ValidationError* exception is raised when data fails form or model field validation. For more information about validation, see Form and Field Validation, *Model Field Validation* and the Validator Reference.

NON_FIELD_ERRORS

NON_FIELD_ERRORS

ValidationErrors that don't belong to a particular field in a form or model are classified as NON_FIELD_ERRORS. This constant is used as a key in dictionaries that otherwise map fields to their respective list of errors.

6.10.2 URL Resolver exceptions

URL Resolver exceptions are defined in `django.urls`.

1.10 : In older versions, these exceptions are located in `django.core.urlresolvers`. Importing from the old location will continue to work until Django 2.0.

Resolver404

exception Resolver404

The *Resolver404* exception is raised by *resolve()* if the path passed to `resolve()` doesn't map to a view. It's a subclass of *django.http.Http404*.

NoReverseMatch

exception NoReverseMatch

The *NoReverseMatch* exception is raised by *django.urls* when a matching URL in your URLconf cannot be identified based on the parameters supplied.

6.10.3 Database Exceptions

Database exceptions may be imported from `django.db`.

Django wraps the standard database exceptions so that your Django code has a guaranteed common implementation of these classes.

exception Error

exception InterfaceError

exception DatabaseError

exception DataError

exception OperationalError

exception IntegrityError

exception InternalError

exception ProgrammingError

exception NotSupportedError

The Django wrappers for database exceptions behave exactly the same as the underlying database exceptions. See PEP 249, the Python Database API Specification v2.0, for further information.

As per PEP 3134, a `__cause__` attribute is set with the original (underlying) database exception, allowing access to any additional information provided. (Note that this attribute is available under both Python 2 and Python 3, although PEP 3134 normally only applies to Python 3. To avoid unexpected differences with Python 3, Django will also ensure that the exception made available via `__cause__` has a usable `__traceback__` attribute.)

The `__traceback__` attribute described above was added.

exception `models.`**ProtectedError**

Raised to prevent deletion of referenced objects when using *`django.db.models.PROTECT`*. *`models.ProtectedError`* is a subclass of *`IntegrityError`*.

6.10.4 Http Exceptions

Http exceptions may be imported from `django.http`.

UnreadablePostError

exception UnreadablePostError
`UnreadablePostError` is raised when a user cancels an upload.

6.10.5 Transaction Exceptions

Transaction exceptions are defined in `django.db.transaction`.

TransactionManagementError

exception TransactionManagementError
`TransactionManagementError` is raised for any and all problems related to database transactions.

6.10.6 Testing Framework Exceptions

Exceptions provided by the `django.test` package.

RedirectCycleError

exception `client.`**RedirectCycleError**
`RedirectCycleError` is raised when the test client detects a loop or an overly long chain of redirects.

6.10.7 Python Exceptions

Django raises built-in Python exceptions when appropriate as well. See the Python documentation for further information on the Built-in Exceptions.

6.11 File handling

6.11.1 The `File` object

The *`django.core.files`* module and its submodules contain built-in classes for basic file handling in Django.

The `File` class

class `File`(*file_object*)

The *`File`* class is a thin wrapper around a Python file object with some Django-specific additions. Internally, Django uses this class when it needs to represent a file.

`File` objects have the following attributes and methods:

`name`

The name of the file including the relative path from *`MEDIA_ROOT`*.

`size`

The size of the file in bytes.

`file`

The underlying file object that this class wraps.

Be careful with this attribute in subclasses.

Some subclasses of *`File`*, including *`ContentFile`* and *`FieldFile`*, may replace this attribute with an object other than a Python file object. In these cases, this attribute may itself be a *`File`* subclass (and not necessarily the same subclass). Whenever possible, use the attributes and methods of the subclass itself rather than the those of the subclass's `file` attribute.

`mode`

The read/write mode for the file.

`open`(*mode=None*)

Open or reopen the file (which also does `File.seek(0)`). The `mode` argument allows the same values as Python's built-in `open()`.

When reopening a file, `mode` will override whatever mode the file was originally opened with; `None` means to reopen with the original mode.

`read`(*num_bytes=None*)

Read content from the file. The optional `size` is the number of bytes to read; if not specified, the file will be read to the end.

`__iter__`()

Iterate over the file yielding one line at a time.

`chunks`(*chunk_size=None*)

Iterate over the file yielding "chunks" of a given size. `chunk_size` defaults to 64 KB.

This is especially useful with very large files since it allows them to be streamed off disk and avoids storing the whole file in memory.

`multiple_chunks`(*chunk_size=None*)

Returns `True` if the file is large enough to require multiple chunks to access all of its content give some `chunk_size`.

write(*content*)
Writes the specified content string to the file. Depending on the storage system behind the scenes, this content might not be fully committed until *close()* is called on the file.

close()
Close the file.

In addition to the listed methods, *File* exposes the following attributes and methods of its `file` object: `encoding`, `fileno`, `flush`, `isatty`, `newlines`, `read`, `readinto`, `readlines`, `seek`, `softspace`, `tell`, `truncate`, `writelines`, `xreadlines`. If you are using Python 3, the `seekable` method is also available.

The `seekable` method was added.

The `ContentFile` class

class ContentFile(*File*)
The `ContentFile` class inherits from *File*, but unlike *File* it operates on string content (bytes also supported), rather than an actual file. For example:

```
from __future__ import unicode_literals
from django.core.files.base import ContentFile

f1 = ContentFile("esta sentencia está en español")
f2 = ContentFile(b"these are bytes")
```

The `ImageFile` class

class ImageFile(*file_object*)
Django provides a built-in class specifically for images. *django.core.files.images.ImageFile* inherits all the attributes and methods of *File*, and additionally provides the following:

width
Width of the image in pixels.

height
Height of the image in pixels.

Additional methods on files attached to objects

Any *File* that is associated with an object (as with `Car.photo`, below) will also have a couple of extra methods:

`File.`**save**(*name*, *content*, *save=True*)
Saves a new file with the file name and contents provided. This will not replace the existing file, but will create a new file and update the object to point to it. If `save` is `True`, the model's `save()` method will be called once the file is saved. That is, these two lines:

```
>>> car.photo.save('myphoto.jpg', content, save=False)
>>> car.save()
```

are equivalent to:

```
>>> car.photo.save('myphoto.jpg', content, save=True)
```

Note that the `content` argument must be an instance of either *File* or of a subclass of *File*, such as *ContentFile*.

`File.delete`(*save=True*)
Removes the file from the model instance and deletes the underlying file. If `save` is `True`, the model's `save()` method will be called once the file is deleted.

6.11.2 File storage API

Getting the current storage class

Django provides two convenient ways to access the current storage class:

class `DefaultStorage`
`DefaultStorage` provides lazy access to the current default storage system as defined by *`DEFAULT_FILE_STORAGE`*. *`DefaultStorage`* uses *`get_storage_class()`* internally.

`get_storage_class`(*import_path=None*)
Returns a class or module which implements the storage API.

When called without the `import_path` parameter `get_storage_class` will return the current default storage system as defined by *`DEFAULT_FILE_STORAGE`*. If `import_path` is provided, `get_storage_class` will attempt to import the class or module from the given path and will return it if successful. An exception will be raised if the import is unsuccessful.

The `FileSystemStorage` class

class `FileSystemStorage`(*location=None*, *base_url=None*, *file_permissions_mode=None*, *directory_permissions_mode=None*)
The *`FileSystemStorage`* class implements basic file storage on a local filesystem. It inherits from *`Storage`* and provides implementations for all the public methods thereof.

`location`
Absolute path to the directory that will hold the files. Defaults to the value of your *`MEDIA_ROOT`* setting.

`base_url`
URL that serves the files stored at this location. Defaults to the value of your *`MEDIA_URL`* setting.

`file_permissions_mode`
The file system permissions that the file will receive when it is saved. Defaults to *`FILE_UPLOAD_PERMISSIONS`*.

`directory_permissions_mode`
The file system permissions that the directory will receive when it is saved. Defaults to *`FILE_UPLOAD_DIRECTORY_PERMISSIONS`*.

: The `FileSystemStorage.delete()` method will not raise an exception if the given file name does not exist.

The `Storage` class

class `Storage`
The *`Storage`* class provides a standardized API for storing files, along with a set of default behaviors that all other storage systems can inherit or override as necessary.

: When methods return naive `datetime` objects, the effective timezone used will be the current value of `os.environ['TZ']`; note that this is usually set from Django's *TIME_ZONE*.

accessed_time(*name*)
Returns a naive `datetime` object containing the last accessed time of the file. For storage systems that aren't able to return the last accessed time this will raise `NotImplementedError` instead.

1.10 : Use *get_accessed_time()* instead.

created_time(*name*)
Returns a naive `datetime` object containing the creation time of the file. For storage systems that aren't able to return the creation time this will raise `NotImplementedError` instead.

1.10 : Use *get_created_time()* instead.

delete(*name*)
Deletes the file referenced by `name`. If deletion is not supported on the target storage system this will raise `NotImplementedError` instead

exists(*name*)
Returns `True` if a file referenced by the given name already exists in the storage system, or `False` if the name is available for a new file.

get_accessed_time(*name*)
Returns a `datetime` of the last accessed time of the file. For storage systems unable to return the last accessed time this will raise `NotImplementedError`.

If *USE_TZ* is `True`, returns an aware `datetime`, otherwise returns a naive `datetime` in the local timezone.

get_available_name(*name*, *max_length=None*)
Returns a filename based on the `name` parameter that's free and available for new content to be written to on the target storage system.

The length of the filename will not exceed `max_length`, if provided. If a free unique filename cannot be found, a *SuspiciousFileOperation* exception will be raised.

If a file with `name` already exists, an underscore plus a random 7 character alphanumeric string is appended to the filename before the extension.

get_created_time(*name*)
Returns a `datetime` of the creation time of the file. For storage systems unable to return the creation time this will raise `NotImplementedError`.

If *USE_TZ* is `True`, returns an aware `datetime`, otherwise returns a naive `datetime` in the local timezone.

get_modified_time(*name*)
Returns a `datetime` of the last modified time of the file. For storage systems unable to return the last modified time this will raise `NotImplementedError`.

If *USE_TZ* is `True`, returns an aware `datetime`, otherwise returns a naive `datetime` in the local timezone.

get_valid_name(*name*)
Returns a filename based on the `name` parameter that's suitable for use on the target storage system.

generate_filename(*filename*)
Validates the `filename` by calling *get_valid_name()* and returns a filename to be passed to the *save()* method.

The `filename` argument may include a path as returned by *`FileField.upload_to`*. In that case, the path won't be passed to *`get_valid_name()`* but will be prepended back to the resulting name.

The default implementation uses `os.path` operations. Override this method if that's not appropriate for your storage.

`listdir`(*path*)
Lists the contents of the specified path, returning a 2-tuple of lists; the first item being directories, the second item being files. For storage systems that aren't able to provide such a listing, this will raise a `NotImplementedError` instead.

`modified_time`(*name*)
Returns a naive `datetime` object containing the last modified time. For storage systems that aren't able to return the last modified time, this will raise `NotImplementedError` instead.

1.10 : Use *`get_modified_time()`* instead.

`open`(*name*, *mode='rb'*)
Opens the file given by `name`. Note that although the returned file is guaranteed to be a `File` object, it might actually be some subclass. In the case of remote file storage this means that reading/writing could be quite slow, so be warned.

`path`(*name*)
The local filesystem path where the file can be opened using Python's standard `open()`. For storage systems that aren't accessible from the local filesystem, this will raise `NotImplementedError` instead.

`save`(*name*, *content*, *max_length=None*)
Saves a new file using the storage system, preferably with the name specified. If there already exists a file with this name `name`, the storage system may modify the filename as necessary to get a unique name. The actual name of the stored file will be returned.

The `max_length` argument is passed along to *`get_available_name()`*.

The `content` argument must be an instance of *`django.core.files.File`* or a file-like object that can be wrapped in `File`.

`size`(*name*)
Returns the total size, in bytes, of the file referenced by `name`. For storage systems that aren't able to return the file size this will raise `NotImplementedError` instead.

`url`(*name*)
Returns the URL where the contents of the file referenced by `name` can be accessed. For storage systems that don't support access by URL this will raise `NotImplementedError` instead.

6.11.3 Uploaded Files and Upload Handlers

Uploaded files

class `UploadedFile`

During file uploads, the actual file data is stored in *`request.FILES`*. Each entry in this dictionary is an `UploadedFile` object (or a subclass) – a simple wrapper around an uploaded file. You'll usually use one of these methods to access the uploaded content:

`UploadedFile.`**`read`**`()`
Read the entire uploaded data from the file. Be careful with this method: if the uploaded file is huge it can overwhelm your system if you try to read it into memory. You'll probably want to use `chunks()` instead; see below.

`UploadedFile.`**`multiple_chunks`**`(`*`chunk_size=None`*`)`
Returns `True` if the uploaded file is big enough to require reading in multiple chunks. By default this will be any file larger than 2.5 megabytes, but that's configurable; see below.

`UploadedFile.`**`chunks`**`(`*`chunk_size=None`*`)`
A generator returning chunks of the file. If `multiple_chunks()` is `True`, you should use this method in a loop instead of `read()`.

In practice, it's often easiest simply to use `chunks()` all the time. Looping over `chunks()` instead of using `read()` ensures that large files don't overwhelm your system's memory.

Here are some useful attributes of `UploadedFile`:

`UploadedFile.`**`name`**
The name of the uploaded file (e.g. `my_file.txt`).

`UploadedFile.`**`size`**
The size, in bytes, of the uploaded file.

`UploadedFile.`**`content_type`**
The content-type header uploaded with the file (e.g. *`text/plain`* or *`application/pdf`*). Like any data supplied by the user, you shouldn't trust that the uploaded file is actually this type. You'll still need to validate that the file contains the content that the content-type header claims – "trust but verify."

`UploadedFile.`**`content_type_extra`**
A dictionary containing extra parameters passed to the `content-type` header. This is typically provided by services, such as Google App Engine, that intercept and handle file uploads on your behalf. As a result your handler may not receive the uploaded file content, but instead a URL or other pointer to the file. (see RFC 2388 section 5.3).

`UploadedFile.`**`charset`**
For *`text/*`* content-types, the character set (i.e. `utf8`) supplied by the browser. Again, "trust but verify" is the best policy here.

: Like regular Python files, you can read the file line-by-line simply by iterating over the uploaded file:

```
for line in uploadedfile:
    do_something_with(line)
```

Lines are split using universal newlines. The following are recognized as ending a line: the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old Macintosh convention `'\r'`.

Subclasses of `UploadedFile` include:

class `TemporaryUploadedFile`
A file uploaded to a temporary location (i.e. stream-to-disk). This class is used by the *`TemporaryFileUploadHandler`*. In addition to the methods from *`UploadedFile`*, it has one additional method:

`TemporaryUploadedFile.`**`temporary_file_path`**`()`
Returns the full path to the temporary uploaded file.

class `InMemoryUploadedFile`
A file uploaded into memory (i.e. stream-to-memory). This class is used by the *`MemoryFileUploadHandler`*.

Built-in upload handlers

Together the *MemoryFileUploadHandler* and *TemporaryFileUploadHandler* provide Django's default file upload behavior of reading small files into memory and large ones onto disk. They are located in `django.core.files.uploadhandler`.

class MemoryFileUploadHandler

File upload handler to stream uploads into memory (used for small files).

class TemporaryFileUploadHandler

Upload handler that streams data into a temporary file using *TemporaryUploadedFile*.

Writing custom upload handlers

class FileUploadHandler

All file upload handlers should be subclasses of `django.core.files.uploadhandler.FileUploadHandler`. You can define upload handlers wherever you wish.

Required methods

Custom file upload handlers **must** define the following methods:

`FileUploadHandler.`**`receive_data_chunk`**(*raw_data*, *start*)
Receives a "chunk" of data from the file upload.

`raw_data` is a byte string containing the uploaded data.

`start` is the position in the file where this `raw_data` chunk begins.

The data you return will get fed into the subsequent upload handlers' `receive_data_chunk` methods. In this way, one handler can be a "filter" for other handlers.

Return `None` from `receive_data_chunk` to short-circuit remaining upload handlers from getting this chunk. This is useful if you're storing the uploaded data yourself and don't want future handlers to store a copy of the data.

If you raise a `StopUpload` or a `SkipFile` exception, the upload will abort or the file will be completely skipped.

`FileUploadHandler.`**`file_complete`**(*file_size*)
Called when a file has finished uploading.

The handler should return an `UploadedFile` object that will be stored in `request.FILES`. Handlers may also return `None` to indicate that the `UploadedFile` object should come from subsequent upload handlers.

Optional methods

Custom upload handlers may also define any of the following optional methods or attributes:

`FileUploadHandler.`**`chunk_size`**
Size, in bytes, of the "chunks" Django should store into memory and feed into the handler. That is, this attribute controls the size of chunks fed into `FileUploadHandler.receive_data_chunk`.

For maximum performance the chunk sizes should be divisible by `4` and should not exceed 2 GB (2^{31} bytes) in size. When there are multiple chunk sizes provided by multiple handlers, Django will use the smallest chunk size defined by any handler.

The default is $64*2^{10}$ bytes, or 64 KB.

`FileUploadHandler.`**`new_file`**(*field_name*, *file_name*, *content_type*, *content_length*, *charset*, *content_type_extra*)

Callback signaling that a new file upload is starting. This is called before any data has been fed to any upload handlers.

`field_name` is a string name of the file `<input>` field.

`file_name` is the unicode filename that was provided by the browser.

`content_type` is the MIME type provided by the browser – E.g. `'image/jpeg'`.

`content_length` is the length of the image given by the browser. Sometimes this won't be provided and will be `None`.

`charset` is the character set (i.e. `utf8`) given by the browser. Like `content_length`, this sometimes won't be provided.

`content_type_extra` is extra information about the file from the `content-type` header. See *`UploadedFile.content_type_extra`*.

This method may raise a `StopFutureHandlers` exception to prevent future handlers from handling this file.

`FileUploadHandler.`**`upload_complete`**`()`

Callback signaling that the entire upload (all files) has completed.

`FileUploadHandler.`**`handle_raw_input`**(*input_data*, *META*, *content_length*, *boundary*, *encoding*)

Allows the handler to completely override the parsing of the raw HTTP input.

`input_data` is a file-like object that supports `read()`-ing.

`META` is the same object as `request.META`.

`content_length` is the length of the data in `input_data`. Don't read more than `content_length` bytes from `input_data`.

`boundary` is the MIME boundary for this request.

`encoding` is the encoding of the request.

Return `None` if you want upload handling to continue, or a tuple of `(POST, FILES)` if you want to return the new data structures suitable for the request directly.

6.12 Forms

Detailed form API reference. For introductory material, see the Working with forms topic guide.

6.12.1 The Forms API

About this document

This document covers the gritty details of Django's forms API. You should read the introduction to working with forms first.

Bound and unbound forms

A *Form* instance is either **bound** to a set of data, or **unbound**.

- If it's **bound** to a set of data, it's capable of validating that data and rendering the form as HTML with the data displayed in the HTML.
- If it's **unbound**, it cannot do validation (because there's no data to validate!), but it can still render the blank form as HTML.

class Form

To create an unbound *Form* instance, simply instantiate the class:

```
>>> f = ContactForm()
```

To bind data to a form, pass the data as a dictionary as the first parameter to your *Form* class constructor:

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data)
```

In this dictionary, the keys are the field names, which correspond to the attributes in your *Form* class. The values are the data you're trying to validate. These will usually be strings, but there's no requirement that they be strings; the type of data you pass depends on the *Field*, as we'll see in a moment.

`Form.`**`is_bound`**

If you need to distinguish between bound and unbound form instances at runtime, check the value of the form's *is_bound* attribute:

```
>>> f = ContactForm()
>>> f.is_bound
False
>>> f = ContactForm({'subject': 'hello'})
>>> f.is_bound
True
```

Note that passing an empty dictionary creates a *bound* form with empty data:

```
>>> f = ContactForm({})
>>> f.is_bound
True
```

If you have a bound *Form* instance and want to change the data somehow, or if you want to bind an unbound *Form* instance to some data, create another *Form* instance. There is no way to change data in a *Form* instance. Once a *Form* instance has been created, you should consider its data immutable, whether it has data or not.

Using forms to validate data

`Form.`**`clean`**`()`

Implement a `clean()` method on your `Form` when you must add custom validation for fields that are interdependent. See *Cleaning and validating fields that depend on each other* for example usage.

`Form.`**`is_valid`**`()`

The primary task of a *Form* object is to validate data. With a bound *Form* instance, call the *is_valid()* method to run validation and return a boolean designating whether the data was valid:

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
True
```

Let's try with some invalid data. In this case, `subject` is blank (an error, because all fields are required by default) and `sender` is not a valid email address:

```
>>> data = {'subject': '',
...         'message': 'Hi there',
...         'sender': 'invalid email address',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
False
```

`Form.`**`errors`**

Access the *`errors`* attribute to get a dictionary of error messages:

```
>>> f.errors
{'sender': ['Enter a valid email address.'], 'subject': ['This field is required.']}
```

In this dictionary, the keys are the field names, and the values are lists of Unicode strings representing the error messages. The error messages are stored in lists because a field can have multiple error messages.

You can access *`errors`* without having to call *`is_valid()`* first. The form's data will be validated the first time either you call *`is_valid()`* or access *`errors`*.

The validation routines will only get called once, regardless of how many times you access *`errors`* or call *`is_valid()`*. This means that if validation has side effects, those side effects will only be triggered once.

`Form.errors.`**`as_data`**`()`

Returns a `dict` that maps fields to their original `ValidationError` instances.

```
>>> f.errors.as_data()
{'sender': [ValidationError(['Enter a valid email address.'])],
'subject': [ValidationError(['This field is required.'])]}
```

Use this method anytime you need to identify an error by its `code`. This enables things like rewriting the error's message or writing custom logic in a view when a given error is present. It can also be used to serialize the errors in a custom format (e.g. XML); for instance, *`as_json()`* relies on `as_data()`.

The need for the `as_data()` method is due to backwards compatibility. Previously `ValidationError` instances were lost as soon as their **rendered** error messages were added to the `Form.errors` dictionary. Ideally `Form.errors` would have stored `ValidationError` instances and methods with an `as_` prefix could render them, but it had to be done the other way around in order not to break code that expects rendered error messages in `Form.errors`.

`Form.errors.`**`as_json`**`(`*`escape_html=False`*`)`

Returns the errors serialized as JSON.

```
>>> f.errors.as_json()
{"sender": [{"message": "Enter a valid email address.", "code": "invalid"}],
"subject": [{"message": "This field is required.", "code": "required"}]}
```

By default, `as_json()` does not escape its output. If you are using it for something like AJAX requests to a form view where the client interprets the response and inserts errors into the page, you'll want to be sure to escape the results on the client-side to avoid the possibility of a cross-site scripting attack. It's trivial to do so using a JavaScript library like jQuery - simply use `$(el).text(errorText)` rather than `.html()`.

If for some reason you don't want to use client-side escaping, you can also set `escape_html=True` and error messages will be escaped so you can use them directly in HTML.

`Form.`**`add_error`**(*field*, *error*)

This method allows adding errors to specific fields from within the `Form.clean()` method, or from outside the form altogether; for instance from a view.

The `field` argument is the name of the field to which the errors should be added. If its value is `None` the error will be treated as a non-field error as returned by *`Form.non_field_errors()`*.

The `error` argument can be a simple string, or preferably an instance of `ValidationError`. See *Raising ValidationError* for best practices when defining form errors.

Note that `Form.add_error()` automatically removes the relevant field from `cleaned_data`.

`Form.`**`has_error`**(*field*, *code=None*)

This method returns a boolean designating whether a field has an error with a specific error `code`. If `code` is `None`, it will return `True` if the field contains any errors at all.

To check for non-field errors use *`NON_FIELD_ERRORS`* as the `field` parameter.

`Form.`**`non_field_errors`**`()`

This method returns the list of errors from *`Form.errors`* that aren't associated with a particular field. This includes `ValidationErrors` that are raised in *`Form.clean()`* and errors added using *`Form.add_error(None, "...")`*.

Behavior of unbound forms

It's meaningless to validate a form with no data, but, for the record, here's what happens with unbound forms:

```
>>> f = ContactForm()
>>> f.is_valid()
False
>>> f.errors
{}
```

Dynamic initial values

`Form.`**`initial`**

Use *`initial`* to declare the initial value of form fields at runtime. For example, you might want to fill in a `username` field with the username of the current session.

To accomplish this, use the *`initial`* argument to a *`Form`*. This argument, if given, should be a dictionary mapping field names to initial values. Only include the fields for which you're specifying an initial value; it's not necessary to include every field in your form. For example:

```
>>> f = ContactForm(initial={'subject': 'Hi there!'})
```

These values are only displayed for unbound forms, and they're not used as fallback values if a particular value isn't provided.

If a *Field* defines *initial* *and* you include *initial* when instantiating the Form, then the latter initial will have precedence. In this example, initial is provided both at the field level and at the form instance level, and the latter gets precedence:

```
>>> from django import forms
>>> class CommentForm(forms.Form):
...     name = forms.CharField(initial='class')
...     url = forms.URLField()
...     comment = forms.CharField()
>>> f = CommentForm(initial={'name': 'instance'}, auto_id=False)
>>> print(f)
<tr><th>Name:</th><td><input type="text" name="name" value="instance" required /></td></tr>
<tr><th>Url:</th><td><input type="url" name="url" required /></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" required /></td></tr>
```

Checking which form data has changed

Form.**has_changed**()

Use the has_changed() method on your Form when you need to check if the form data has been changed from the initial data.

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data, initial=data)
>>> f.has_changed()
False
```

When the form is submitted, we reconstruct it and provide the original data so that the comparison can be done:

```
>>> f = ContactForm(request.POST, initial=data)
>>> f.has_changed()
```

has_changed() will be True if the data from request.POST differs from what was provided in *initial* or False otherwise. The result is computed by calling *Field.has_changed()* for each field in the form.

Form.**changed_data**

The changed_data attribute returns a list of the names of the fields whose values in the form's bound data (usually request.POST) differ from what was provided in *initial*. It returns an empty list if no data differs.

```
>>> f = ContactForm(request.POST, initial=data)
>>> if f.has_changed():
...     print("The following fields changed: %s" % ", ".join(f.changed_data))
```

Accessing the fields from the form

Form.**fields**

You can access the fields of *Form* instance from its fields attribute:

```
>>> for row in f.fields.values(): print(row)
...
<django.forms.fields.CharField object at 0x7ffaac632510>
<django.forms.fields.URLField object at 0x7ffaac632f90>
<django.forms.fields.CharField object at 0x7ffaac3aa050>
```

```
>>> f.fields['name']
<django.forms.fields.CharField object at 0x7ffaac6324d0>
```

You can alter the field of *Form* instance to change the way it is presented in the form:

```
>>> f.as_table().split('\n')[0]
'<tr><th>Name:</th><td><input name="name" type="text" value="instance" required /></td></tr>'
>>> f.fields['name'].label = "Username"
>>> f.as_table().split('\n')[0]
'<tr><th>Username:</th><td><input name="name" type="text" value="instance" required /></td></tr>'
```

Beware not to alter the `base_fields` attribute because this modification will influence all subsequent `ContactForm` instances within the same Python process:

```
>>> f.base_fields['name'].label = "Username"
>>> another_f = CommentForm(auto_id=False)
>>> another_f.as_table().split('\n')[0]
'<tr><th>Username:</th><td><input name="name" type="text" value="class" required /></td></tr>'
```

Accessing "clean" data

`Form.`**`cleaned_data`**

Each field in a *Form* class is responsible not only for validating data, but also for "cleaning" it – normalizing it to a consistent format. This is a nice feature, because it allows data for a particular field to be input in a variety of ways, always resulting in consistent output.

For example, *DateField* normalizes input into a Python `datetime.date` object. Regardless of whether you pass it a string in the format `'1994-07-15'`, a `datetime.date` object, or a number of other formats, `DateField` will always normalize it to a `datetime.date` object as long as it's valid.

Once you've created a *Form* instance with a set of data and validated it, you can access the clean data via its `cleaned_data` attribute:

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
True
>>> f.cleaned_data
{'cc_myself': True, 'message': 'Hi there', 'sender': 'foo@example.com', 'subject': 'hello'}
```

Note that any text-based field – such as `CharField` or `EmailField` – always cleans the input into a Unicode string. We'll cover the encoding implications later in this document.

If your data does *not* validate, the `cleaned_data` dictionary contains only the valid fields:

```
>>> data = {'subject': '',
...         'message': 'Hi there',
...         'sender': 'invalid email address',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
False
>>> f.cleaned_data
{'cc_myself': True, 'message': 'Hi there'}
```

`cleaned_data` will always *only* contain a key for fields defined in the `Form`, even if you pass extra data when you define the `Form`. In this example, we pass a bunch of extra fields to the `ContactForm` constructor, but `cleaned_data` contains only the form's fields:

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True,
...         'extra_field_1': 'foo',
...         'extra_field_2': 'bar',
...         'extra_field_3': 'baz'}
>>> f = ContactForm(data)
>>> f.is_valid()
True
>>> f.cleaned_data # Doesn't contain extra_field_1, etc.
{'cc_myself': True, 'message': 'Hi there', 'sender': 'foo@example.com', 'subject': 'hello'}
```

When the `Form` is valid, `cleaned_data` will include a key and value for *all* its fields, even if the data didn't include a value for some optional fields. In this example, the data dictionary doesn't include a value for the `nick_name` field, but `cleaned_data` includes it, with an empty value:

```
>>> from django import forms
>>> class OptionalPersonForm(forms.Form):
...     first_name = forms.CharField()
...     last_name = forms.CharField()
...     nick_name = forms.CharField(required=False)
>>> data = {'first_name': 'John', 'last_name': 'Lennon'}
>>> f = OptionalPersonForm(data)
>>> f.is_valid()
True
>>> f.cleaned_data
{'nick_name': '', 'first_name': 'John', 'last_name': 'Lennon'}
```

In this above example, the `cleaned_data` value for `nick_name` is set to an empty string, because `nick_name` is `CharField`, and `CharField`s treat empty values as an empty string. Each field type knows what its "blank" value is – e.g., for `DateField`, it's `None` instead of the empty string. For full details on each field's behavior in this case, see the "Empty value" note for each field in the "Built-in `Field` classes" section below.

You can write code to perform validation for particular form fields (based on their name) or for the form as a whole (considering combinations of various fields). More information about this is in Form and field validation.

Outputting forms as HTML

The second task of a `Form` object is to render itself as HTML. To do so, simply `print` it:

```
>>> f = ContactForm()
>>> print(f)
<tr><th><label for="id_subject">Subject:</label></th><td><input id="id_subject" type="text" name="sub
<tr><th><label for="id_message">Message:</label></th><td><input type="text" name="message" id="id_mes
<tr><th><label for="id_sender">Sender:</label></th><td><input type="email" name="sender" id="id_sende
<tr><th><label for="id_cc_myself">Cc myself:</label></th><td><input type="checkbox" name="cc_myself"
```

If the form is bound to data, the HTML output will include that data appropriately. For example, if a field is represented by an `<input type="text">`, the data will be in the `value` attribute. If a field is represented by an `<input type="checkbox">`, then that HTML will include `checked="checked"` if appropriate:

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
```

```
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> print(f)
<tr><th><label for="id_subject">Subject:</label></th><td><input id="id_subject" type="text" name="sub
<tr><th><label for="id_message">Message:</label></th><td><input type="text" name="message" id="id_mes
<tr><th><label for="id_sender">Sender:</label></th><td><input type="email" name="sender" id="id_sende
<tr><th><label for="id_cc_myself">Cc myself:</label></th><td><input type="checkbox" name="cc_myself"
```

This default output is a two-column HTML table, with a `<tr>` for each field. Notice the following:

- For flexibility, the output does *not* include the `<table>` and `</table>` tags, nor does it include the `<form>` and `</form>` tags or an `<input type="submit">` tag. It's your job to do that.
- Each field type has a default HTML representation. `CharField` is represented by an `<input type="text">` and `EmailField` by an `<input type="email">`. `BooleanField` is represented by an `<input type="checkbox">`. Note these are merely sensible defaults; you can specify which HTML to use for a given field by using widgets, which we'll explain shortly.
- The HTML `name` for each tag is taken directly from its attribute name in the `ContactForm` class.
- The text label for each field – e.g. `'Subject:'`, `'Message:'` and `'Cc myself:'` is generated from the field name by converting all underscores to spaces and upper-casing the first letter. Again, note these are merely sensible defaults; you can also specify labels manually.
- Each text label is surrounded in an HTML `<label>` tag, which points to the appropriate form field via its `id`. Its `id`, in turn, is generated by prepending `'id_'` to the field name. The `id` attributes and `<label>` tags are included in the output by default, to follow best practices, but you can change that behavior.

Although `<table>` output is the default output style when you `print` a form, other output styles are available. Each style is available as a method on a form object, and each rendering method returns a Unicode object.

as_p()

`Form.as_p()`

`as_p()` renders the form as a series of `<p>` tags, with each `<p>` containing one field:

```
>>> f = ContactForm()
>>> f.as_p()
'<p><label for="id_subject">Subject:</label> <input id="id_subject" type="text" name="subject" maxlen
>>> print(f.as_p())
<p><label for="id_subject">Subject:</label> <input id="id_subject" type="text" name="subject" maxlen
<p><label for="id_message">Message:</label> <input type="text" name="message" id="id_message" require
<p><label for="id_sender">Sender:</label> <input type="email" name="sender" id="id_sender" required
<p><label for="id_cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="id_cc_mys
```

as_ul()

`Form.as_ul()`

`as_ul()` renders the form as a series of `<li>` tags, with each `<li>` containing one field. It does *not* include the `<ul>` or `</ul>`, so that you can specify any HTML attributes on the `<ul>` for flexibility:

```
>>> f = ContactForm()
>>> f.as_ul()
'<li><label for="id_subject">Subject:</label> <input id="id_subject" type="text" name="subject" maxle
>>> print(f.as_ul())
```

```
<li><label for="id_subject">Subject:</label> <input id="id_subject" type="text" name="subject" maxle
<li><label for="id_message">Message:</label> <input type="text" name="message" id="id_message" requi
<li><label for="id_sender">Sender:</label> <input type="email" name="sender" id="id_sender" required
<li><label for="id_cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="id_cc_my
```

as_table()

Form.**as_table**()

Finally, `as_table()` outputs the form as an HTML `<table>`. This is exactly the same as `print`. In fact, when you `print` a form object, it calls its `as_table()` method behind the scenes:

```
>>> f = ContactForm()
>>> f.as_table()
'<tr><th><label for="id_subject">Subject:</label></th><td><input id="id_subject" type="text" name="su
>>> print(f)
<tr><th><label for="id_subject">Subject:</label></th><td><input id="id_subject" type="text" name="sub
<tr><th><label for="id_message">Message:</label></th><td><input type="text" name="message" id="id_mes
<tr><th><label for="id_sender">Sender:</label></th><td><input type="email" name="sender" id="id_sende
<tr><th><label for="id_cc_myself">Cc myself:</label></th><td><input type="checkbox" name="cc_myself"
```

Styling required or erroneous form rows

Form.**error_css_class**

Form.**required_css_class**

It's pretty common to style form rows and fields that are required or have errors. For example, you might want to present required form rows in bold and highlight errors in red.

The *Form* class has a couple of hooks you can use to add `class` attributes to required rows or to rows with errors: simply set the *Form.error_css_class* and/or *Form.required_css_class* attributes:

```
from django import forms

class ContactForm(forms.Form):
    error_css_class = 'error'
    required_css_class = 'required'

    # ... and the rest of your fields here
```

Once you've done that, rows will be given `"error"` and/or `"required"` classes, as needed. The HTML will look something like:

```
>>> f = ContactForm(data)
>>> print(f.as_table())
<tr class="required"><th><label class="required" for="id_subject">Subject:</label>    ...
<tr class="required"><th><label class="required" for="id_message">Message:</label>    ...
<tr class="required error"><th><label class="required" for="id_sender">Sender:</label>      ...
<tr><th><label for="id_cc_myself">Cc myself:<label> ...
>>> f['subject'].label_tag()
<label class="required" for="id_subject">Subject:</label>
>>> f['subject'].label_tag(attrs={'class': 'foo'})
<label for="id_subject" class="foo required">Subject:</label>
```

Configuring form elements' HTML `id` attributes and `<label>` tags

`Form.`**`auto_id`**

By default, the form rendering methods include:

- HTML `id` attributes on the form elements.
- The corresponding `<label>` tags around the labels. An HTML `<label>` tag designates which label text is associated with which form element. This small enhancement makes forms more usable and more accessible to assistive devices. It's always a good idea to use `<label>` tags.

The `id` attribute values are generated by prepending `id_` to the form field names. This behavior is configurable, though, if you want to change the `id` convention or remove HTML `id` attributes and `<label>` tags entirely.

Use the `auto_id` argument to the `Form` constructor to control the `id` and label behavior. This argument must be `True`, `False` or a string.

If `auto_id` is `False`, then the form output will not include `<label>` tags nor `id` attributes:

```
>>> f = ContactForm(auto_id=False)
>>> print(f.as_table())
<tr><th>Subject:</th><td><input type="text" name="subject" maxlength="100" required /></td></tr>
<tr><th>Message:</th><td><input type="text" name="message" required /></td></tr>
<tr><th>Sender:</th><td><input type="email" name="sender" required /></td></tr>
<tr><th>Cc myself:</th><td><input type="checkbox" name="cc_myself" /></td></tr>
>>> print(f.as_ul())
<li>Subject: <input type="text" name="subject" maxlength="100" required /></li>
<li>Message: <input type="text" name="message" required /></li>
<li>Sender: <input type="email" name="sender" required /></li>
<li>Cc myself: <input type="checkbox" name="cc_myself" /></li>
>>> print(f.as_p())
<p>Subject: <input type="text" name="subject" maxlength="100" required /></p>
<p>Message: <input type="text" name="message" required /></p>
<p>Sender: <input type="email" name="sender" required /></p>
<p>Cc myself: <input type="checkbox" name="cc_myself" /></p>
```

If `auto_id` is set to `True`, then the form output *will* include `<label>` tags and will simply use the field name as its `id` for each form field:

```
>>> f = ContactForm(auto_id=True)
>>> print(f.as_table())
<tr><th><label for="subject">Subject:</label></th><td><input id="subject" type="text" name="subject"
<tr><th><label for="message">Message:</label></th><td><input type="text" name="message" id="message"
<tr><th><label for="sender">Sender:</label></th><td><input type="email" name="sender" id="sender" re
<tr><th><label for="cc_myself">Cc myself:</label></th><td><input type="checkbox" name="cc_myself" id=
>>> print(f.as_ul())
<li><label for="subject">Subject:</label> <input id="subject" type="text" name="subject" maxlength="1
<li><label for="message">Message:</label> <input type="text" name="message" id="message" required /><
<li><label for="sender">Sender:</label> <input type="email" name="sender" id="sender" required /></li
<li><label for="cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="cc_myself"
>>> print(f.as_p())
<p><label for="subject">Subject:</label> <input id="subject" type="text" name="subject" maxlength="10
<p><label for="message">Message:</label> <input type="text" name="message" id="message" required /></
<p><label for="sender">Sender:</label> <input type="email" name="sender" id="sender" required /></p>
<p><label for="cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="cc_myself" /
```

If `auto_id` is set to a string containing the format character `'%s'`, then the form output will include `<label>` tags, and will generate `id` attributes based on the format string. For example, for a format string `'field_%s'`, a field named `subject` will get the `id` value `'field_subject'`. Continuing our example:

```
>>> f = ContactForm(auto_id='id_for_%s')
>>> print(f.as_table())
<tr><th><label for="id_for_subject">Subject:</label></th><td><input id="id_for_subject" type="text" n
<tr><th><label for="id_for_message">Message:</label></th><td><input type="text" name="message" id="id
<tr><th><label for="id_for_sender">Sender:</label></th><td><input type="email" name="sender" id="id_f
<tr><th><label for="id_for_cc_myself">Cc myself:</label></th><td><input type="checkbox" name="cc_myse
>>> print(f.as_ul())
<li><label for="id_for_subject">Subject:</label> <input id="id_for_subject" type="text" name="subject
<li><label for="id_for_message">Message:</label> <input type="text" name="message" id="id_for_message
<li><label for="id_for_sender">Sender:</label> <input type="email" name="sender" id="id_for_sender" n
<li><label for="id_for_cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="id_f
>>> print(f.as_p())
<p><label for="id_for_subject">Subject:</label> <input id="id_for_subject" type="text" name="subject"
<p><label for="id_for_message">Message:</label> <input type="text" name="message" id="id_for_message"
<p><label for="id_for_sender">Sender:</label> <input type="email" name="sender" id="id_for_sender" re
<p><label for="id_for_cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="id_fo
```

If `auto_id` is set to any other true value – such as a string that doesn't include `%s` – then the library will act as if `auto_id` is `True`.

By default, `auto_id` is set to the string `'id_%s'`.

`Form.`**`label_suffix`**

A translatable string (defaults to a colon (`:`) in English) that will be appended after any label name when a form is rendered.

It's possible to customize that character, or omit it entirely, using the `label_suffix` parameter:

```
>>> f = ContactForm(auto_id='id_for_%s', label_suffix='')
>>> print(f.as_ul())
<li><label for="id_for_subject">Subject</label> <input id="id_for_subject" type="text" name="subject"
<li><label for="id_for_message">Message</label> <input type="text" name="message" id="id_for_message"
<li><label for="id_for_sender">Sender</label> <input type="email" name="sender" id="id_for_sender" re
<li><label for="id_for_cc_myself">Cc myself</label> <input type="checkbox" name="cc_myself" id="id_fo
>>> f = ContactForm(auto_id='id_for_%s', label_suffix=' ->')
>>> print(f.as_ul())
<li><label for="id_for_subject">Subject -></label> <input id="id_for_subject" type="text" name="subje
<li><label for="id_for_message">Message -></label> <input type="text" name="message" id="id_for_messa
<li><label for="id_for_sender">Sender -></label> <input type="email" name="sender" id="id_for_sender"
<li><label for="id_for_cc_myself">Cc myself -></label> <input type="checkbox" name="cc_myself" id="id
```

Note that the label suffix is added only if the last character of the label isn't a punctuation character (in English, those are `.`, `!`, `?` or `:`).

Fields can also define their own *`label_suffix`*. This will take precedence over *`Form.label_suffix`*. The suffix can also be overridden at runtime using the `label_suffix` parameter to *`label_tag()`*.

`Form.`**`use_required_attribute`**

When set to `True` (the default), required form fields will have the `required` HTML attribute.

Formsets instantiate forms with `use_required_attribute=False` to avoid incorrect browser validation when adding and deleting forms from a formset.

Notes on field ordering

In the `as_p()`, `as_ul()` and `as_table()` shortcuts, the fields are displayed in the order in which you define them in your form class. For example, in the `ContactForm` example, the fields are defined in the order `subject`,

`message`, `sender`, `cc_myself`. To reorder the HTML output, just change the order in which those fields are listed in the class.

There are several other ways to customize the order:

`Form.`**`field_order`**

By default `Form.field_order=None`, which retains the order in which you define the fields in your form class. If `field_order` is a list of field names, the fields are ordered as specified by the list and remaining fields are appended according to the default order. Unknown field names in the list are ignored. This makes it possible to disable a field in a subclass by setting it to `None` without having to redefine ordering.

You can also use the `Form.field_order` argument to a *`Form`* to override the field order. If a *`Form`* defines *`field_order`* *and* you include `field_order` when instantiating the `Form`, then the latter `field_order` will have precedence.

`Form.`**`order_fields`**(*field_order*)

You may rearrange the fields any time using `order_fields()` with a list of field names as in *`field_order`*.

How errors are displayed

If you render a bound `Form` object, the act of rendering will automatically run the form's validation if it hasn't already happened, and the HTML output will include the validation errors as a `<ul class="errorlist">` near the field. The particular positioning of the error messages depends on the output method you're using:

```
>>> data = {'subject': '',
...         'message': 'Hi there',
...         'sender': 'invalid email address',
...         'cc_myself': True}
>>> f = ContactForm(data, auto_id=False)
>>> print(f.as_table())
<tr><th>Subject:</th><td><ul class="errorlist"><li>This field is required.</li></ul><input type="text
<tr><th>Message:</th><td><input type="text" name="message" value="Hi there" required /></td></tr>
<tr><th>Sender:</th><td><ul class="errorlist"><li>Enter a valid email address.</li></ul><input type="
<tr><th>Cc myself:</th><td><input checked="checked" type="checkbox" name="cc_myself" /></td></tr>
>>> print(f.as_ul())
<li><ul class="errorlist"><li>This field is required.</li></ul>Subject: <input type="text" name="subj
<li>Message: <input type="text" name="message" value="Hi there" required /></li>
<li><ul class="errorlist"><li>Enter a valid email address.</li></ul>Sender: <input type="email" name=
<li>Cc myself: <input checked="checked" type="checkbox" name="cc_myself" /></li>
>>> print(f.as_p())
<p><ul class="errorlist"><li>This field is required.</li></ul></p>
<p>Subject: <input type="text" name="subject" maxlength="100" required /></p>
<p>Message: <input type="text" name="message" value="Hi there" required /></p>
<p><ul class="errorlist"><li>Enter a valid email address.</li></ul></p>
<p>Sender: <input type="email" name="sender" value="invalid email address" required /></p>
<p>Cc myself: <input checked="checked" type="checkbox" name="cc_myself" /></p>
```

Customizing the error list format

By default, forms use `django.forms.utils.ErrorList` to format validation errors. If you'd like to use an alternate class for displaying errors, you can pass that in at construction time (replace `__str__` by `__unicode__` on Python 2):

```
>>> from django.forms.utils import ErrorList
>>> class DivErrorList(ErrorList):
```

```
...     def __str__(self):              # __unicode__ on Python 2
...         return self.as_divs()
...     def as_divs(self):
...         if not self: return ''
...         return '<div class="errorlist">%s</div>' % ''.join(['<div class="error">%s</div>' % e for
>>> f = ContactForm(data, auto_id=False, error_class=DivErrorList)
>>> f.as_p()
<div class="errorlist"><div class="error">This field is required.</div></div>
<p>Subject: <input type="text" name="subject" maxlength="100" required /></p>
<p>Message: <input type="text" name="message" value="Hi there" required /></p>
<div class="errorlist"><div class="error">Enter a valid email address.</div></div>
<p>Sender: <input type="email" name="sender" value="invalid email address" required /></p>
<p>Cc myself: <input checked="checked" type="checkbox" name="cc_myself" /></p>
```

More granular output

The `as_p()`, `as_ul()`, and `as_table()` methods are simply shortcuts – they're not the only way a form object can be displayed.

class BoundField

Used to display HTML or access attributes for a single field of a *Form* instance.

The `__str__()` (`__unicode__` on Python 2) method of this object displays the HTML for this field.

To retrieve a single `BoundField`, use dictionary lookup syntax on your form using the field's name as the key:

```
>>> form = ContactForm()
>>> print(form['subject'])
<input id="id_subject" type="text" name="subject" maxlength="100" required />
```

To retrieve all `BoundField` objects, iterate the form:

```
>>> form = ContactForm()
>>> for boundfield in form: print(boundfield)
<input id="id_subject" type="text" name="subject" maxlength="100" required />
<input type="text" name="message" id="id_message" required />
<input type="email" name="sender" id="id_sender" required />
<input type="checkbox" name="cc_myself" id="id_cc_myself" />
```

The field-specific output honors the form object's `auto_id` setting:

```
>>> f = ContactForm(auto_id=False)
>>> print(f['message'])
<input type="text" name="message" required />
>>> f = ContactForm(auto_id='id_%s')
>>> print(f['message'])
<input type="text" name="message" id="id_message" required />
```

Attributes of `BoundField`

`BoundField.`**auto_id**

The HTML ID attribute for this `BoundField`. Returns an empty string if *Form.auto_id* is `False`.

`BoundField.`**data**

This property returns the data for this *BoundField* extracted by the widget's *value_from_datadict()* method, or `None` if it wasn't given:

```
>>> unbound_form = ContactForm()
>>> print(unbound_form['subject'].data)
None
>>> bound_form = ContactForm(data={'subject': 'My Subject'})
>>> print(bound_form['subject'].data)
My Subject
```

BoundField.**errors**
A *list-like object* that is displayed as an HTML <ul class="errorlist"> when printed:

```
>>> data = {'subject': 'hi', 'message': '', 'sender': '', 'cc_myself': ''}
>>> f = ContactForm(data, auto_id=False)
>>> print(f['message'])
<input type="text" name="message" required />
>>> f['message'].errors
['This field is required.']
>>> print(f['message'].errors)
<ul class="errorlist"><li>This field is required.</li></ul>
>>> f['subject'].errors
[]
>>> print(f['subject'].errors)

>>> str(f['subject'].errors)
''
```

BoundField.**field**
The form *Field* instance from the form class that this *BoundField* wraps.

BoundField.**form**
The *Form* instance this *BoundField* is bound to.

BoundField.**help_text**
The *help_text* of the field.

BoundField.**html_name**
The name that will be used in the widget's HTML name attribute. It takes the form *prefix* into account.

BoundField.**id_for_label**
Use this property to render the ID of this field. For example, if you are manually constructing a <label> in your template (despite the fact that *label_tag()* will do this for you):

```
<label for="{{ form.my_field.id_for_label }}">...</label>{{ my_field }}
```

By default, this will be the field's name prefixed by id_ ("id_my_field" for the example above). You may modify the ID by setting *attrs* on the field's widget. For example, declaring a field like this:

```
my_field = forms.CharField(widget=forms.TextInput(attrs={'id': 'myFIELD'}))
```

and using the template above, would render something like:

```
<label for="myFIELD">...</label><input id="myFIELD" type="text" name="my_field" required />
```

BoundField.**is_hidden**
Returns True if this *BoundField*'s widget is hidden.

BoundField.**label**
The *label* of the field. This is used in *label_tag()*.

BoundField.**name**
The name of this field in the form:

```
>>> f = ContactForm()
>>> print(f['subject'].name)
subject
>>> print(f['message'].name)
message
```

Methods of `BoundField`

`BoundField.`**`as_hidden`**(*attrs=None*, ***kwargs*)
Returns a string of HTML for representing this as an `<input type="hidden">`.

`**kwargs` are passed to *`as_widget()`*.

This method is primarily used internally. You should use a widget instead.

`BoundField.`**`as_widget`**(*widget=None*, *attrs=None*, *only_initial=False*)
Renders the field by rendering the passed widget, adding any HTML attributes passed as `attrs`. If no widget is specified, then the field's default widget will be used.

`only_initial` is used by Django internals and should not be set explicitly.

`BoundField.`**`css_classes`**()
When you use Django's rendering shortcuts, CSS classes are used to indicate required form fields or fields that contain errors. If you're manually rendering a form, you can access these CSS classes using the `css_classes` method:

```
>>> f = ContactForm(data={'message': ''})
>>> f['message'].css_classes()
'required'
```

If you want to provide some additional classes in addition to the error and required classes that may be required, you can provide those classes as an argument:

```
>>> f = ContactForm(data={'message': ''})
>>> f['message'].css_classes('foo bar')
'foo bar required'
```

`BoundField.`**`label_tag`**(*contents=None*, *attrs=None*, *label_suffix=None*)
To separately render the label tag of a form field, you can call its `label_tag()` method:

```
>>> f = ContactForm(data={'message': ''})
>>> print(f['message'].label_tag())
<label for="id_message">Message:</label>
```

You can provide the `contents` parameter which will replace the auto-generated label tag. An `attrs` dictionary may contain additional attributes for the `<label>` tag.

The HTML that's generated includes the form's *`label_suffix`* (a colon, by default) or, if set, the current field's *`label_suffix`*. The optional `label_suffix` parameter allows you to override any previously set suffix. For example, you can use an empty string to hide the label on selected fields. If you need to do this in a template, you could write a custom filter to allow passing parameters to `label_tag`.

`BoundField.`**`value`**()
Use this method to render the raw value of this field as it would be rendered by a `Widget`:

```
>>> initial = {'subject': 'welcome'}
>>> unbound_form = ContactForm(initial=initial)
>>> bound_form = ContactForm(data={'subject': 'hi'}, initial=initial)
>>> print(unbound_form['subject'].value())
```

```
    welcome
    >>> print(bound_form['subject'].value())
    hi
```

Customizing BoundField

If you need to access some additional information about a form field in a template and using a subclass of *Field* isn't sufficient, consider also customizing *BoundField*.

A custom form field can override `get_bound_field()`:

`Field.`**`get_bound_field`**(*form*, *field_name*)

Takes an instance of *Form* and the name of the field. The return value will be used when accessing the field in a template. Most likely it will be an instance of a subclass of *BoundField*.

If you have a `GPSCoordinatesField`, for example, and want to be able to access additional information about the coordinates in a template, this could be implemented as follows:

```
class GPSCoordinatesBoundField(BoundField):
    @property
    def country(self):
        """
        Return the country the coordinates lie in or None if it can't be
        determined.
        """
        value = self.value()
        if value:
            return get_country_from_coordinates(value)
        else:
            return None

class GPSCoordinatesField(Field):
    def get_bound_field(self, form, field_name):
        return GPSCoordinatesBoundField(form, self, field_name)
```

Now you can access the country in a template with `{{ form.coordinates.country }}`.

Binding uploaded files to a form

Dealing with forms that have `FileField` and `ImageField` fields is a little more complicated than a normal form.

Firstly, in order to upload files, you'll need to make sure that your `<form>` element correctly defines the `enctype` as `"multipart/form-data"`:

```
<form enctype="multipart/form-data" method="post" action="/foo/">
```

Secondly, when you use the form, you need to bind the file data. File data is handled separately to normal form data, so when your form contains a `FileField` and `ImageField`, you will need to specify a second argument when you bind your form. So if we extend our ContactForm to include an `ImageField` called `mugshot`, we need to bind the file data containing the mugshot image:

```
# Bound form with an image field
>>> from django.core.files.uploadedfile import SimpleUploadedFile
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
```

```
>>> file_data = {'mugshot': SimpleUploadedFile('face.jpg', <file data>)}
>>> f = ContactFormWithMugshot(data, file_data)
```

In practice, you will usually specify `request.FILES` as the source of file data (just like you use `request.POST` as the source of form data):

```
# Bound form with an image field, data from the request
>>> f = ContactFormWithMugshot(request.POST, request.FILES)
```

Constructing an unbound form is the same as always – just omit both form data *and* file data:

```
# Unbound form with an image field
>>> f = ContactFormWithMugshot()
```

Testing for multipart forms

`Form.`**`is_multipart`**`()`

If you're writing reusable views or templates, you may not know ahead of time whether your form is a multipart form or not. The `is_multipart()` method tells you whether the form requires multipart encoding for submission:

```
>>> f = ContactFormWithMugshot()
>>> f.is_multipart()
True
```

Here's an example of how you might use this in a template:

```
{% if form.is_multipart %}
    <form enctype="multipart/form-data" method="post" action="/foo/">
{% else %}
    <form method="post" action="/foo/">
{% endif %}
{{ form }}
</form>
```

Subclassing forms

If you have multiple `Form` classes that share fields, you can use subclassing to remove redundancy.

When you subclass a custom `Form` class, the resulting subclass will include all fields of the parent class(es), followed by the fields you define in the subclass.

In this example, `ContactFormWithPriority` contains all the fields from `ContactForm`, plus an additional field, `priority`. The `ContactForm` fields are ordered first:

```
>>> class ContactFormWithPriority(ContactForm):
...     priority = forms.CharField()
>>> f = ContactFormWithPriority(auto_id=False)
>>> print(f.as_ul())
<li>Subject: <input type="text" name="subject" maxlength="100" required /></li>
<li>Message: <input type="text" name="message" required /></li>
<li>Sender: <input type="email" name="sender" required /></li>
<li>Cc myself: <input type="checkbox" name="cc_myself" /></li>
<li>Priority: <input type="text" name="priority" required /></li>
```

It's possible to subclass multiple forms, treating forms as mixins. In this example, `BeatleForm` subclasses both `PersonForm` and `InstrumentForm` (in that order), and its field list includes the fields from the parent classes:

```
>>> from django import forms
>>> class PersonForm(forms.Form):
...     first_name = forms.CharField()
...     last_name = forms.CharField()
>>> class InstrumentForm(forms.Form):
...     instrument = forms.CharField()
>>> class BeatleForm(InstrumentForm, PersonForm):
...     haircut_type = forms.CharField()
>>> b = BeatleForm(auto_id=False)
>>> print(b.as_ul())
<li>First name: <input type="text" name="first_name" required /></li>
<li>Last name: <input type="text" name="last_name" required /></li>
<li>Instrument: <input type="text" name="instrument" required /></li>
<li>Haircut type: <input type="text" name="haircut_type" required /></li>
```

It's possible to declaratively remove a `Field` inherited from a parent class by setting the name of the field to `None` on the subclass. For example:

```
>>> from django import forms

>>> class ParentForm(forms.Form):
...     name = forms.CharField()
...     age = forms.IntegerField()

>>> class ChildForm(ParentForm):
...     name = None

>>> ChildForm().fields.keys()
... ['age']
```

Prefixes for forms

`Form.`**`prefix`**

You can put several Django forms inside one `<form>` tag. To give each `Form` its own namespace, use the `prefix` keyword argument:

```
>>> mother = PersonForm(prefix="mother")
>>> father = PersonForm(prefix="father")
>>> print(mother.as_ul())
<li><label for="id_mother-first_name">First name:</label> <input type="text" name="mother-first_name"
<li><label for="id_mother-last_name">Last name:</label> <input type="text" name="mother-last_name" id
>>> print(father.as_ul())
<li><label for="id_father-first_name">First name:</label> <input type="text" name="father-first_name"
<li><label for="id_father-last_name">Last name:</label> <input type="text" name="father-last_name" id
```

The prefix can also be specified on the form class:

```
>>> class PersonForm(forms.Form):
...     ...
...     prefix = 'person'
```

The ability to specify `prefix` on the form class was added.

6.12.2 Form fields

class `Field`(***kwargs*)

When you create a `Form` class, the most important part is defining the fields of the form. Each field has custom validation logic, along with a few other hooks.

`Field.`**`clean`**`(`*`value`*`)`

Although the primary way you'll use `Field` classes is in `Form` classes, you can also instantiate them and use them directly to get a better idea of how they work. Each `Field` instance has a `clean()` method, which takes a single argument and either raises a `django.forms.ValidationError` exception or returns the clean value:

```
>>> from django import forms
>>> f = forms.EmailField()
>>> f.clean('foo@example.com')
'foo@example.com'
>>> f.clean('invalid email address')
Traceback (most recent call last):
...
ValidationError: ['Enter a valid email address.']
```

Core field arguments

Each `Field` class constructor takes at least these arguments. Some `Field` classes take additional, field-specific arguments, but the following should *always* be accepted:

required

`Field.`**`required`**

By default, each `Field` class assumes the value is required, so if you pass an empty value – either `None` or the empty string (`""`) – then `clean()` will raise a `ValidationError` exception:

```
>>> from django import forms
>>> f = forms.CharField()
>>> f.clean('foo')
'foo'
>>> f.clean('')
Traceback (most recent call last):
...
ValidationError: ['This field is required.']
>>> f.clean(None)
Traceback (most recent call last):
...
ValidationError: ['This field is required.']
>>> f.clean(' ')
' '
>>> f.clean(0)
'0'
>>> f.clean(True)
'True'
>>> f.clean(False)
'False'
```

To specify that a field is *not* required, pass `required=False` to the `Field` constructor:

```
>>> f = forms.CharField(required=False)
>>> f.clean('foo')
'foo'
>>> f.clean('')
```

```
''
>>> f.clean(None)
''
>>> f.clean(0)
'0'
>>> f.clean(True)
'True'
>>> f.clean(False)
'False'
```

If a `Field` has `required=False` and you pass `clean()` an empty value, then `clean()` will return a *normalized* empty value rather than raising `ValidationError`. For `CharField`, this will be a Unicode empty string. For other `Field` classes, it might be `None`. (This varies from field to field.)

label

Field.label

The `label` argument lets you specify the "human-friendly" label for this field. This is used when the `Field` is displayed in a `Form`.

As explained in "Outputting forms as HTML" above, the default label for a `Field` is generated from the field name by converting all underscores to spaces and upper-casing the first letter. Specify `label` if that default behavior doesn't result in an adequate label.

Here's a full example `Form` that implements `label` for two of its fields. We've specified `auto_id=False` to simplify the output:

```
>>> from django import forms
>>> class CommentForm(forms.Form):
...     name = forms.CharField(label='Your name')
...     url = forms.URLField(label='Your website', required=False)
...     comment = forms.CharField()
>>> f = CommentForm(auto_id=False)
>>> print(f)
<tr><th>Your name:</th><td><input type="text" name="name" required /></td></tr>
<tr><th>Your website:</th><td><input type="url" name="url" required /></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" required /></td></tr>
```

label_suffix

Field.label_suffix

The `label_suffix` argument lets you override the form's *label_suffix* on a per-field basis:

```
>>> class ContactForm(forms.Form):
...     age = forms.IntegerField()
...     nationality = forms.CharField()
...     captcha_answer = forms.IntegerField(label='2 + 2', label_suffix=' =')
>>> f = ContactForm(label_suffix='?')
>>> print(f.as_p())
<p><label for="id_age">Age?</label> <input id="id_age" name="age" type="number" required /></p>
<p><label for="id_nationality">Nationality?</label> <input id="id_nationality" name="nationality" typ
<p><label for="id_captcha_answer">2 + 2 =</label> <input id="id_captcha_answer" name="captcha_answer"
```

initial

Field.**initial**

The `initial` argument lets you specify the initial value to use when rendering this `Field` in an unbound `Form`.

To specify dynamic initial data, see the *`Form.initial`* parameter.

The use-case for this is when you want to display an "empty" form in which a field is initialized to a particular value. For example:

```
>>> from django import forms
>>> class CommentForm(forms.Form):
...     name = forms.CharField(initial='Your name')
...     url = forms.URLField(initial='http://')
...     comment = forms.CharField()
>>> f = CommentForm(auto_id=False)
>>> print(f)
<tr><th>Name:</th><td><input type="text" name="name" value="Your name" required /></td></tr>
<tr><th>Url:</th><td><input type="url" name="url" value="http://" required /></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" required /></td></tr>
```

You may be thinking, why not just pass a dictionary of the initial values as data when displaying the form? Well, if you do that, you'll trigger validation, and the HTML output will include any validation errors:

```
>>> class CommentForm(forms.Form):
...     name = forms.CharField()
...     url = forms.URLField()
...     comment = forms.CharField()
>>> default_data = {'name': 'Your name', 'url': 'http://'}
>>> f = CommentForm(default_data, auto_id=False)
>>> print(f)
<tr><th>Name:</th><td><input type="text" name="name" value="Your name" required /></td></tr>
<tr><th>Url:</th><td><ul class="errorlist"><li>Enter a valid URL.</li></ul><input type="url" name="u
<tr><th>Comment:</th><td><ul class="errorlist"><li>This field is required.</li></ul><input type="text
```

This is why `initial` values are only displayed for unbound forms. For bound forms, the HTML output will use the bound data.

Also note that `initial` values are *not* used as "fallback" data in validation if a particular field's value is not given. `initial` values are *only* intended for initial form display:

```
>>> class CommentForm(forms.Form):
...     name = forms.CharField(initial='Your name')
...     url = forms.URLField(initial='http://')
...     comment = forms.CharField()
>>> data = {'name': '', 'url': '', 'comment': 'Foo'}
>>> f = CommentForm(data)
>>> f.is_valid()
False
# The form does *not* fall back to using the initial values.
>>> f.errors
{'url': ['This field is required.'], 'name': ['This field is required.']}
```

Instead of a constant, you can also pass any callable:

```
>>> import datetime
>>> class DateForm(forms.Form):
...     day = forms.DateField(initial=datetime.date.today)
```

```
>>> print(DateForm())
<tr><th>Day:</th><td><input type="text" name="day" value="12/23/2008" required /><td></tr>
```

The callable will be evaluated only when the unbound form is displayed, not when it is defined.

widget

`Field.widget`

The `widget` argument lets you specify a `Widget` class to use when rendering this `Field`. See Widgets for more information.

help_text

`Field.help_text`

The `help_text` argument lets you specify descriptive text for this `Field`. If you provide `help_text`, it will be displayed next to the `Field` when the `Field` is rendered by one of the convenience `Form` methods (e.g., `as_ul()`).

Like the model field's *help_text*, this value isn't HTML-escaped in automatically-generated forms.

Here's a full example `Form` that implements `help_text` for two of its fields. We've specified `auto_id=False` to simplify the output:

```
>>> from django import forms
>>> class HelpTextContactForm(forms.Form):
...     subject = forms.CharField(max_length=100, help_text='100 characters max.')
...     message = forms.CharField()
...     sender = forms.EmailField(help_text='A valid email address, please.')
...     cc_myself = forms.BooleanField(required=False)
>>> f = HelpTextContactForm(auto_id=False)
>>> print(f.as_table())
<tr><th>Subject:</th><td><input type="text" name="subject" maxlength="100" required /><br /><span cla
<tr><th>Message:</th><td><input type="text" name="message" required /></td></tr>
<tr><th>Sender:</th><td><input type="email" name="sender" required /><br />A valid email address, ple
<tr><th>Cc myself:</th><td><input type="checkbox" name="cc_myself" /></td></tr>
>>> print(f.as_ul()))
<li>Subject: <input type="text" name="subject" maxlength="100" required /> <span class="helptext">100
<li>Message: <input type="text" name="message" required /></li>
<li>Sender: <input type="email" name="sender" required /> A valid email address, please.</li>
<li>Cc myself: <input type="checkbox" name="cc_myself" /></li>
>>> print(f.as_p())
<p>Subject: <input type="text" name="subject" maxlength="100" required /> <span class="helptext">100
<p>Message: <input type="text" name="message" required /></p>
<p>Sender: <input type="email" name="sender" required /> A valid email address, please.</p>
<p>Cc myself: <input type="checkbox" name="cc_myself" /></p>
```

error_messages

`Field.error_messages`

The `error_messages` argument lets you override the default messages that the field will raise. Pass in a dictionary with keys matching the error messages you want to override. For example, here is the default error message:

```
>>> from django import forms
>>> generic = forms.CharField()
>>> generic.clean('')
Traceback (most recent call last):
  ...
ValidationError: ['This field is required.']
```

And here is a custom error message:

```
>>> name = forms.CharField(error_messages={'required': 'Please enter your name'})
>>> name.clean('')
Traceback (most recent call last):
  ...
ValidationError: ['Please enter your name']
```

In the *built-in Field classes* section below, each `Field` defines the error message keys it uses.

validators

`Field.`**`validators`**

The `validators` argument lets you provide a list of validation functions for this field.

See the validators documentation for more information.

localize

`Field.`**`localize`**

The `localize` argument enables the localization of form data input, as well as the rendered output.

See the format localization documentation for more information.

disabled

`Field.`**`disabled`**

The `disabled` boolean argument, when set to `True`, disables a form field using the `disabled` HTML attribute so that it won't be editable by users. Even if a user tampers with the field's value submitted to the server, it will be ignored in favor of the value from the form's initial data.

Checking if the field data has changed

has_changed()

`Field.`**`has_changed`**`()`

The `has_changed()` method is used to determine if the field value has changed from the initial value. Returns `True` or `False`.

See the *`Form.has_changed()`* documentation for more information.

Built-in `Field` classes

Naturally, the `forms` library comes with a set of `Field` classes that represent common validation needs. This section documents each built-in field.

For each field, we describe the default widget used if you don't specify `widget`. We also specify the value returned when you provide an empty value (see the section on `required` above to understand what that means).

`BooleanField`

class `BooleanField`(***kwargs*)

- Default widget: *`CheckboxInput`*
- Empty value: `False`
- Normalizes to: A Python `True` or `False` value.
- Validates that the value is `True` (e.g. the check box is checked) if the field has `required=True`.
- Error message keys: `required`

: Since all `Field` subclasses have `required=True` by default, the validation condition here is important. If you want to include a boolean in your form that can be either `True` or `False` (e.g. a checked or unchecked checkbox), you must remember to pass in `required=False` when creating the `BooleanField`.

`CharField`

class `CharField`(***kwargs*)

- Default widget: *`TextInput`*
- Empty value: `''` (an empty string)
- Normalizes to: A Unicode object.
- Validates `max_length` or `min_length`, if they are provided. Otherwise, all inputs are valid.
- Error message keys: `required`, `max_length`, `min_length`

Has three optional arguments for validation:

`max_length`

`min_length`

If provided, these arguments ensure that the string is at most or at least the given length.

`strip`
If `True` (default), the value will be stripped of leading and trailing whitespace.

`ChoiceField`

class `ChoiceField`(***kwargs*)

- Default widget: *`Select`*
- Empty value: `''` (an empty string)

- Normalizes to: A Unicode object.
- Validates that the given value exists in the list of choices.
- Error message keys: `required`, `invalid_choice`

The `invalid_choice` error message may contain `%(value)s`, which will be replaced with the selected choice.

Takes one extra required argument:

`choices`
Either an iterable (e.g., a list or tuple) of 2-tuples to use as choices for this field, or a callable that returns such an iterable. This argument accepts the same formats as the `choices` argument to a model field. See the *model field reference documentation on choices* for more details. If the argument is a callable, it is evaluated each time the field's form is initialized.

TypedChoiceField

class `TypedChoiceField`(***kwargs*)
Just like a *`ChoiceField`*, except *`TypedChoiceField`* takes two extra arguments, *`coerce`* and *`empty_value`*.

- Default widget: *`Select`*
- Empty value: Whatever you've given as *`empty_value`*.
- Normalizes to: A value of the type provided by the *`coerce`* argument.
- Validates that the given value exists in the list of choices and can be coerced.
- Error message keys: `required`, `invalid_choice`

Takes extra arguments:

`coerce`
A function that takes one argument and returns a coerced value. Examples include the built-in `int`, `float`, `bool` and other types. Defaults to an identity function. Note that coercion happens after input validation, so it is possible to coerce to a value not present in `choices`.

`empty_value`
The value to use to represent "empty." Defaults to the empty string; `None` is another common choice here. Note that this value will not be coerced by the function given in the `coerce` argument, so choose it accordingly.

DateField

class `DateField`(***kwargs*)

- Default widget: *`DateInput`*
- Empty value: `None`
- Normalizes to: A Python `datetime.date` object.
- Validates that the given value is either a `datetime.date`, `datetime.datetime` or string formatted in a particular date format.
- Error message keys: `required`, `invalid`

Takes one optional argument:

input_formats
A list of formats used to attempt to convert a string to a valid `datetime.date` object.

If no `input_formats` argument is provided, the default input formats are:

```
['%Y-%m-%d',       # '2006-10-25'
 '%m/%d/%Y',       # '10/25/2006'
 '%m/%d/%y']       # '10/25/06'
```

Additionally, if you specify *`USE_L10N=False`* in your settings, the following will also be included in the default input formats:

```
['%b %d %Y',       # 'Oct 25 2006'
 '%b %d, %Y',      # 'Oct 25, 2006'
 '%d %b %Y',       # '25 Oct 2006'
 '%d %b, %Y',      # '25 Oct, 2006'
 '%B %d %Y',       # 'October 25 2006'
 '%B %d, %Y',      # 'October 25, 2006'
 '%d %B %Y',       # '25 October 2006'
 '%d %B, %Y']      # '25 October, 2006'
```

See also format localization.

DateTimeField

class DateTimeField(***kwargs*)

- Default widget: *`DateTimeInput`*
- Empty value: `None`
- Normalizes to: A Python `datetime.datetime` object.
- Validates that the given value is either a `datetime.datetime`, `datetime.date` or string formatted in a particular datetime format.
- Error message keys: `required`, `invalid`

Takes one optional argument:

input_formats
A list of formats used to attempt to convert a string to a valid `datetime.datetime` object.

If no `input_formats` argument is provided, the default input formats are:

```
['%Y-%m-%d %H:%M:%S',     # '2006-10-25 14:30:59'
 '%Y-%m-%d %H:%M',        # '2006-10-25 14:30'
 '%Y-%m-%d',              # '2006-10-25'
 '%m/%d/%Y %H:%M:%S',     # '10/25/2006 14:30:59'
 '%m/%d/%Y %H:%M',        # '10/25/2006 14:30'
 '%m/%d/%Y',              # '10/25/2006'
 '%m/%d/%y %H:%M:%S',     # '10/25/06 14:30:59'
 '%m/%d/%y %H:%M',        # '10/25/06 14:30'
 '%m/%d/%y']              # '10/25/06'
```

See also format localization.

DecimalField

class DecimalField(***kwargs*)

- Default widget: *NumberInput* when *Field.localize* is `False`, else *TextInput*.
- Empty value: `None`
- Normalizes to: A Python `decimal`.
- Validates that the given value is a decimal. Leading and trailing whitespace is ignored.
- Error message keys: `required`, `invalid`, `max_value`, `min_value`, `max_digits`, `max_decimal_places`, `max_whole_digits`

The `max_value` and `min_value` error messages may contain `%(limit_value)s`, which will be substituted by the appropriate limit. Similarly, the `max_digits`, `max_decimal_places` and `max_whole_digits` error messages may contain `%(max)s`.

Takes four optional arguments:

max_value

min_value
These control the range of values permitted in the field, and should be given as `decimal.Decimal` values.

max_digits
The maximum number of digits (those before the decimal point plus those after the decimal point, with leading zeros stripped) permitted in the value.

decimal_places
The maximum number of decimal places permitted.

DurationField

class DurationField(***kwargs*)

- Default widget: *TextInput*
- Empty value: `None`
- Normalizes to: A Python `timedelta`.
- Validates that the given value is a string which can be converted into a `timedelta`.
- Error message keys: `required`, `invalid`.

Accepts any format understood by *parse_duration()*.

EmailField

class EmailField(***kwargs*)

- Default widget: *EmailInput*
- Empty value: `''` (an empty string)
- Normalizes to: A Unicode object.
- Validates that the given value is a valid email address, using a moderately complex regular expression.
- Error message keys: `required`, `invalid`

Has two optional arguments for validation, `max_length` and `min_length`. If provided, these arguments ensure that the string is at most or at least the given length.

FileField

class FileField(***kwargs*)

- Default widget: *ClearableFileInput*
- Empty value: `None`
- Normalizes to: An `UploadedFile` object that wraps the file content and file name into a single object.
- Can validate that non-empty file data has been bound to the form.
- Error message keys: `required`, `invalid`, `missing`, `empty`, `max_length`

Has two optional arguments for validation, `max_length` and `allow_empty_file`. If provided, these ensure that the file name is at most the given length, and that validation will succeed even if the file content is empty.

To learn more about the `UploadedFile` object, see the file uploads documentation.

When you use a `FileField` in a form, you must also remember to *bind the file data to the form*.

The `max_length` error refers to the length of the filename. In the error message for that key, `%(max)d` will be replaced with the maximum filename length and `%(length)d` will be replaced with the current filename length.

FilePathField

class FilePathField(***kwargs*)

- Default widget: *Select*
- Empty value: `None`
- Normalizes to: A unicode object
- Validates that the selected choice exists in the list of choices.
- Error message keys: `required`, `invalid_choice`

The field allows choosing from files inside a certain directory. It takes five extra arguments; only `path` is required:

path
The absolute path to the directory whose contents you want listed. This directory must exist.

recursive
If `False` (the default) only the direct contents of `path` will be offered as choices. If `True`, the directory will be descended into recursively and all descendants will be listed as choices.

match
A regular expression pattern; only files with names matching this expression will be allowed as choices.

allow_files
Optional. Either `True` or `False`. Default is `True`. Specifies whether files in the specified location should be included. Either this or *allow_folders* must be `True`.

allow_folders
Optional. Either `True` or `False`. Default is `False`. Specifies whether folders in the specified location should be included. Either this or *allow_files* must be `True`.

FloatField

class FloatField(**kwargs)

- Default widget: *NumberInput* when *Field.localize* is `False`, else *TextInput*.
- Empty value: `None`
- Normalizes to: A Python float.
- Validates that the given value is a float. Leading and trailing whitespace is allowed, as in Python's `float()` function.
- Error message keys: `required`, `invalid`, `max_value`, `min_value`

Takes two optional arguments for validation, `max_value` and `min_value`. These control the range of values permitted in the field.

ImageField

class ImageField(**kwargs)

- Default widget: *ClearableFileInput*
- Empty value: `None`
- Normalizes to: An `UploadedFile` object that wraps the file content and file name into a single object.
- Validates that file data has been bound to the form, and that the file is of an image format understood by Pillow.
- Error message keys: `required`, `invalid`, `missing`, `empty`, `invalid_image`

Using an `ImageField` requires that Pillow is installed with support for the image formats you use. If you encounter a `corrupt image` error when you upload an image, it usually means that Pillow doesn't understand its format. To fix this, install the appropriate library and reinstall Pillow.

When you use an `ImageField` on a form, you must also remember to *bind the file data to the form*.

After the field has been cleaned and validated, the `UploadedFile` object will have an additional `image` attribute containing the Pillow Image instance used to check if the file was a valid image. Also, `UploadedFile.content_type` will be updated with the image's content type if Pillow can determine it, otherwise it will be set to `None`.

IntegerField

class IntegerField(**kwargs)

- Default widget: *NumberInput* when *Field.localize* is `False`, else *TextInput*.
- Empty value: `None`
- Normalizes to: A Python integer or long integer.
- Validates that the given value is an integer. Leading and trailing whitespace is allowed, as in Python's `int()` function.
- Error message keys: `required`, `invalid`, `max_value`, `min_value`

The `max_value` and `min_value` error messages may contain `%(limit_value)s`, which will be substituted by the appropriate limit.

Takes two optional arguments for validation:

max_value

min_value

These control the range of values permitted in the field.

GenericIPAddressField

class GenericIPAddressField(***kwargs*)

A field containing either an IPv4 or an IPv6 address.

- Default widget: *TextInput*
- Empty value: `''` (an empty string)
- Normalizes to: A Unicode object. IPv6 addresses are normalized as described below.
- Validates that the given value is a valid IP address.
- Error message keys: `required`, `invalid`

The IPv6 address normalization follows **RFC 4291#section-2.2** section 2.2, including using the IPv4 format suggested in paragraph 3 of that section, like `::ffff:192.0.2.0`. For example, `2001:0::0:01` would be normalized to `2001::1`, and `::ffff:0a0a:0a0a` to `::ffff:10.10.10.10`. All characters are converted to lowercase.

Takes two optional arguments:

protocol
: Limits valid inputs to the specified protocol. Accepted values are `both` (default), `IPv4` or `IPv6`. Matching is case insensitive.

unpack_ipv4
: Unpacks IPv4 mapped addresses like `::ffff:192.0.2.1`. If this option is enabled that address would be unpacked to `192.0.2.1`. Default is disabled. Can only be used when `protocol` is set to `'both'`.

MultipleChoiceField

class MultipleChoiceField(***kwargs*)

- Default widget: *SelectMultiple*
- Empty value: `[]` (an empty list)
- Normalizes to: A list of Unicode objects.
- Validates that every value in the given list of values exists in the list of choices.
- Error message keys: `required`, `invalid_choice`, `invalid_list`

The `invalid_choice` error message may contain `%(value)s`, which will be replaced with the selected choice.

Takes one extra required argument, `choices`, as for *ChoiceField*.

TypedMultipleChoiceField

class TypedMultipleChoiceField(***kwargs*)

Just like a *MultipleChoiceField*, except *TypedMultipleChoiceField* takes two extra arguments, `coerce` and `empty_value`.

- Default widget: *SelectMultiple*
- Empty value: Whatever you've given as `empty_value`
- Normalizes to: A list of values of the type provided by the `coerce` argument.
- Validates that the given values exists in the list of choices and can be coerced.
- Error message keys: `required`, `invalid_choice`

The `invalid_choice` error message may contain `%(value)s`, which will be replaced with the selected choice.

Takes two extra arguments, `coerce` and `empty_value`, as for *TypedChoiceField*.

NullBooleanField

class NullBooleanField(***kwargs*)

- Default widget: *NullBooleanSelect*
- Empty value: `None`
- Normalizes to: A Python `True`, `False` or `None` value.
- Validates nothing (i.e., it never raises a `ValidationError`).

RegexField

class RegexField(***kwargs*)

- Default widget: *TextInput*
- Empty value: `''` (an empty string)
- Normalizes to: A Unicode object.
- Validates that the given value matches against a certain regular expression.
- Error message keys: `required`, `invalid`

Takes one required argument:

regex
: A regular expression specified either as a string or a compiled regular expression object.

Also takes `max_length`, `min_length`, and `strip`, which work just as they do for *CharField*.

strip
: Defaults to `False`. If enabled, stripping will be applied before the regex validation.

SlugField

class SlugField(***kwargs*)

- Default widget: *TextInput*
- Empty value: `''` (an empty string)
- Normalizes to: A Unicode object.
- Validates that the given value contains only letters, numbers, underscores, and hyphens.
- Error messages: `required`, `invalid`

This field is intended for use in representing a model *SlugField* in forms.

Takes an optional parameter:

allow_unicode
A boolean instructing the field to accept Unicode letters in addition to ASCII letters. Defaults to `False`.

TimeField

class TimeField(***kwargs*)

- Default widget: *TextInput*
- Empty value: `None`
- Normalizes to: A Python `datetime.time` object.
- Validates that the given value is either a `datetime.time` or string formatted in a particular time format.
- Error message keys: `required`, `invalid`

Takes one optional argument:

input_formats
A list of formats used to attempt to convert a string to a valid `datetime.time` object.

If no `input_formats` argument is provided, the default input formats are:

```
'%H:%M:%S',     # '14:30:59'
'%H:%M',        # '14:30'
```

URLField

class URLField(***kwargs*)

- Default widget: *URLInput*
- Empty value: `''` (an empty string)
- Normalizes to: A Unicode object.
- Validates that the given value is a valid URL.
- Error message keys: `required`, `invalid`

Takes the following optional arguments:

max_length

min_length

These are the same as `CharField.max_length` and `CharField.min_length`.

`UUIDField`

class `UUIDField`(***kwargs*)

- Default widget: *`TextInput`*
- Empty value: `''` (an empty string)
- Normalizes to: A `UUID` object.
- Error message keys: `required`, `invalid`

This field will accept any string format accepted as the `hex` argument to the `UUID` constructor.

Slightly complex built-in `Field` classes

`ComboField`

class `ComboField`(***kwargs*)

- Default widget: *`TextInput`*
- Empty value: `''` (an empty string)
- Normalizes to: A Unicode object.
- Validates the given value against each of the fields specified as an argument to the `ComboField`.
- Error message keys: `required`, `invalid`

Takes one extra required argument:

`fields`
The list of fields that should be used to validate the field's value (in the order in which they are provided).

```
>>> from django.forms import ComboField
>>> f = ComboField(fields=[CharField(max_length=20), EmailField()])
>>> f.clean('test@example.com')
'test@example.com'
>>> f.clean('longemailaddress@example.com')
Traceback (most recent call last):
...
ValidationError: ['Ensure this value has at most 20 characters (it has 28).']
```

`MultiValueField`

class `MultiValueField`(*fields=()*, ***kwargs*)

- Default widget: *`TextInput`*
- Empty value: `''` (an empty string)
- Normalizes to: the type returned by the `compress` method of the subclass.
- Validates the given value against each of the fields specified as an argument to the `MultiValueField`.
- Error message keys: `required`, `invalid`, `incomplete`

Aggregates the logic of multiple fields that together produce a single value.

This field is abstract and must be subclassed. In contrast with the single-value fields, subclasses of *MultiValueField* must not implement *clean()* but instead - implement *compress()*.

Takes one extra required argument:

fields
A tuple of fields whose values are cleaned and subsequently combined into a single value. Each value of the field is cleaned by the corresponding field in `fields` – the first value is cleaned by the first field, the second value is cleaned by the second field, etc. Once all fields are cleaned, the list of clean values is combined into a single value by *compress()*.

Also takes one extra optional argument:

require_all_fields
Defaults to `True`, in which case a `required` validation error will be raised if no value is supplied for any field.

When set to `False`, the *Field.required* attribute can be set to `False` for individual fields to make them optional. If no value is supplied for a required field, an `incomplete` validation error will be raised.

A default `incomplete` error message can be defined on the *MultiValueField* subclass, or different messages can be defined on each individual field. For example:

```
from django.core.validators import RegexValidator

class PhoneField(MultiValueField):
    def __init__(self, *args, **kwargs):
        # Define one message for all fields.
        error_messages = {
            'incomplete': 'Enter a country calling code and a phone number.',
        }
        # Or define a different message for each field.
        fields = (
            CharField(
                error_messages={'incomplete': 'Enter a country calling code.'},
                validators=[
                    RegexValidator(r'^[0-9]+$', 'Enter a valid country calling code.'),
                ],
            ),
            CharField(
                error_messages={'incomplete': 'Enter a phone number.'},
                validators=[RegexValidator(r'^[0-9]+$', 'Enter a valid phone number.')],
            ),
            CharField(
                validators=[RegexValidator(r'^[0-9]+$', 'Enter a valid extension.')],
                required=False,
            ),
        )
        super(PhoneField, self).__init__(
            error_messages=error_messages, fields=fields,
            require_all_fields=False, *args, **kwargs
        )
```

widget
Must be a subclass of *django.forms.MultiWidget*. Default value is *TextInput*, which probably is not very useful in this case.

compress(*data_list*)
Takes a list of valid values and returns a "compressed" version of those values – in a single value. For

example, *SplitDateTimeField* is a subclass which combines a time field and a date field into a `datetime` object.

This method must be implemented in the subclasses.

SplitDateTimeField

class SplitDateTimeField(***kwargs*)

- Default widget: *SplitDateTimeWidget*
- Empty value: `None`
- Normalizes to: A Python `datetime.datetime` object.
- Validates that the given value is a `datetime.datetime` or string formatted in a particular datetime format.
- Error message keys: `required`, `invalid`, `invalid_date`, `invalid_time`

Takes two optional arguments:

input_date_formats
A list of formats used to attempt to convert a string to a valid `datetime.date` object.

If no `input_date_formats` argument is provided, the default input formats for *DateField* are used.

input_time_formats
A list of formats used to attempt to convert a string to a valid `datetime.time` object.

If no `input_time_formats` argument is provided, the default input formats for *TimeField* are used.

Fields which handle relationships

Two fields are available for representing relationships between models: *ModelChoiceField* and *ModelMultipleChoiceField*. Both of these fields require a single `queryset` parameter that is used to create the choices for the field. Upon form validation, these fields will place either one model object (in the case of `ModelChoiceField`) or multiple model objects (in the case of `ModelMultipleChoiceField`) into the `cleaned_data` dictionary of the form.

For more complex uses, you can specify `queryset=None` when declaring the form field and then populate the `queryset` in the form's `__init__()` method:

```
class FooMultipleChoiceForm(forms.Form):
    foo_select = forms.ModelMultipleChoiceField(queryset=None)

    def __init__(self, *args, **kwargs):
        super(FooMultipleChoiceForm, self).__init__(*args, **kwargs)
        self.fields['foo_select'].queryset = ...
```

ModelChoiceField

class ModelChoiceField(***kwargs*)

- Default widget: *Select*
- Empty value: `None`
- Normalizes to: A model instance.

- Validates that the given id exists in the queryset.
- Error message keys: `required`, `invalid_choice`

Allows the selection of a single model object, suitable for representing a foreign key. Note that the default widget for `ModelChoiceField` becomes impractical when the number of entries increases. You should avoid using it for more than 100 items.

A single argument is required:

`queryset`
A `QuerySet` of model objects from which the choices for the field will be derived, and which will be used to validate the user's selection.

`ModelChoiceField` also takes two optional arguments:

`empty_label`
By default the `<select>` widget used by `ModelChoiceField` will have an empty choice at the top of the list. You can change the text of this label (which is `"---------"` by default) with the `empty_label` attribute, or you can disable the empty label entirely by setting `empty_label` to `None`:

```
# A custom empty label
field1 = forms.ModelChoiceField(queryset=..., empty_label="(Nothing)")

# No empty label
field2 = forms.ModelChoiceField(queryset=..., empty_label=None)
```

Note that if a `ModelChoiceField` is required and has a default initial value, no empty choice is created (regardless of the value of `empty_label`).

`to_field_name`
This optional argument is used to specify the field to use as the value of the choices in the field's widget. Be sure it's a unique field for the model, otherwise the selected value could match more than one object. By default it is set to `None`, in which case the primary key of each object will be used. For example:

```
# No custom to_field_name
field1 = forms.ModelChoiceField(queryset=...)
```

would yield:

```
<select id="id_field1" name="field1">
<option value="obj1.pk">Object1</option>
<option value="obj2.pk">Object2</option>
...
</select>
```

and:

```
# to_field_name provided
field2 = forms.ModelChoiceField(queryset=..., to_field_name="name")
```

would yield:

```
<select id="id_field2" name="field2">
<option value="obj1.name">Object1</option>
<option value="obj2.name">Object2</option>
...
</select>
```

The `__str__` (`__unicode__` on Python 2) method of the model will be called to generate string representations of the objects for use in the field's choices; to provide customized representations, subclass

`ModelChoiceField` and override `label_from_instance`. This method will receive a model object, and should return a string suitable for representing it. For example:

```
from django.forms import ModelChoiceField

class MyModelChoiceField(ModelChoiceField):
    def label_from_instance(self, obj):
        return "My Object #%i" % obj.id
```

ModelMultipleChoiceField

class ModelMultipleChoiceField(***kwargs*)

- Default widget: *SelectMultiple*
- Empty value: An empty `QuerySet` (self.queryset.none())
- Normalizes to: A `QuerySet` of model instances.
- Validates that every id in the given list of values exists in the queryset.
- Error message keys: `required`, `list`, `invalid_choice`, `invalid_pk_value`

The `invalid_choice` message may contain `%(value)s` and the `invalid_pk_value` message may contain `%(pk)s`, which will be substituted by the appropriate values.

Allows the selection of one or more model objects, suitable for representing a many-to-many relation. As with *ModelChoiceField*, you can use `label_from_instance` to customize the object representations.

A single argument is required:

queryset
Same as *ModelChoiceField.queryset*.

Takes one optional argument:

to_field_name
Same as *ModelChoiceField.to_field_name*.

Creating custom fields

If the built-in `Field` classes don't meet your needs, you can easily create custom `Field` classes. To do this, just create a subclass of `django.forms.Field`. Its only requirements are that it implement a `clean()` method and that its `__init__()` method accept the core arguments mentioned above (`required`, `label`, `initial`, `widget`, `help_text`).

You can also customize how a field will be accessed by overriding *get_bound_field()*.

6.12.3 Model Form Functions

Model Form API reference. For introductory material about model forms, see the Creating forms from models topic guide.

modelform_factory

modelform_factory(*model*, *form=ModelForm*, *fields=None*, *exclude=None*, *formfield_callback=None*, *widgets=None*, *localized_fields=None*, *labels=None*, *help_texts=None*, *error_messages=None*, *field_classes=None*)

Returns a *ModelForm* class for the given `model`. You can optionally pass a `form` argument to use as a starting point for constructing the `ModelForm`.

`fields` is an optional list of field names. If provided, only the named fields will be included in the returned fields.

`exclude` is an optional list of field names. If provided, the named fields will be excluded from the returned fields, even if they are listed in the `fields` argument.

`formfield_callback` is a callable that takes a model field and returns a form field.

`widgets` is a dictionary of model field names mapped to a widget.

`localized_fields` is a list of names of fields which should be localized.

`labels` is a dictionary of model field names mapped to a label.

`help_texts` is a dictionary of model field names mapped to a help text.

`error_messages` is a dictionary of model field names mapped to a dictionary of error messages.

`field_classes` is a dictionary of model field names mapped to a form field class.

See *ModelForm factory function* for example usage.

You must provide the list of fields explicitly, either via keyword arguments `fields` or `exclude`, or the corresponding attributes on the form's inner `Meta` class. See *Selecting the fields to use* for more information. Omitting any definition of the fields to use will result in an *ImproperlyConfigured* exception.

The `field_classes` keyword argument was added.

modelformset_factory

modelformset_factory(*model*, *form=ModelForm*, *formfield_callback=None*, *formset=BaseModelFormSet*, *extra=1*, *can_delete=False*, *can_order=False*, *max_num=None*, *fields=None*, *exclude=None*, *widgets=None*, *validate_max=False*, *localized_fields=None*, *labels=None*, *help_texts=None*, *error_messages=None*, *min_num=None*, *validate_min=False*, *field_classes=None*)

Returns a `FormSet` class for the given `model` class.

Arguments `model`, `form`, `fields`, `exclude`, `formfield_callback`, `widgets`, `localized_fields`, `labels`, `help_texts`, `error_messages`, and `field_classes` are all passed through to *modelform_factory()*.

Arguments `formset`, `extra`, `max_num`, `can_order`, `can_delete` and `validate_max` are passed through to *formset_factory()*. See formsets for details.

See *Model formsets* for example usage.

The `field_classes` keyword argument was added.

inlineformset_factory

inlineformset_factory(*parent_model*, *model*, *form=ModelForm*, *formset=BaseInlineFormSet*, *fk_name=None*, *fields=None*, *exclude=None*, *extra=3*, *can_order=False*, *can_delete=True*, *max_num=None*, *formfield_callback=None*, *widgets=None*, *validate_max=False*, *localized_fields=None*, *labels=None*, *help_texts=None*, *error_messages=None*, *min_num=None*, *validate_min=False*, *field_classes=None*)

Returns an `InlineFormSet` using *`modelformset_factory()`* with defaults of `formset=`*`BaseInlineFormSet`*, `can_delete=True`, and `extra=3`.

If your model has more than one *`ForeignKey`* to the `parent_model`, you must specify a `fk_name`.

See *Inline formsets* for example usage.

The `field_classes` keyword argument was added.

6.12.4 Formset Functions

Formset API reference. For introductory material about formsets, see the Formsets topic guide.

formset_factory

formset_factory(*form*, *formset=BaseFormSet*, *extra=1*, *can_order=False*, *can_delete=False*, *max_num=None*, *validate_max=False*, *min_num=None*, *validate_min=False*)

Returns a `FormSet` class for the given `form` class.

See formsets for example usage.

6.12.5 Widgets

A widget is Django's representation of an HTML input element. The widget handles the rendering of the HTML, and the extraction of data from a GET/POST dictionary that corresponds to the widget.

: Widgets should not be confused with the form fields. Form fields deal with the logic of input validation and are used directly in templates. Widgets deal with rendering of HTML form input elements on the web page and extraction of raw submitted data. However, widgets do need to be *assigned* to form fields.

Specifying widgets

Whenever you specify a field on a form, Django will use a default widget that is appropriate to the type of data that is to be displayed. To find which widget is used on which field, see the documentation about *Built-in Field classes*.

However, if you want to use a different widget for a field, you can just use the *`widget`* argument on the field definition. For example:

```
from django import forms

class CommentForm(forms.Form):
    name = forms.CharField()
    url = forms.URLField()
    comment = forms.CharField(widget=forms.Textarea)
```

This would specify a form with a comment that uses a larger *Textarea* widget, rather than the default *TextInput* widget.

Setting arguments for widgets

Many widgets have optional extra arguments; they can be set when defining the widget on the field. In the following example, the *years* attribute is set for a *SelectDateWidget*:

```
from django import forms

BIRTH_YEAR_CHOICES = ('1980', '1981', '1982')
FAVORITE_COLORS_CHOICES = (
    ('blue', 'Blue'),
    ('green', 'Green'),
    ('black', 'Black'),
)

class SimpleForm(forms.Form):
    birth_year = forms.DateField(widget=forms.SelectDateWidget(years=BIRTH_YEAR_CHOICES))
    favorite_colors = forms.MultipleChoiceField(
        required=False,
        widget=forms.CheckboxSelectMultiple,
        choices=FAVORITE_COLORS_CHOICES,
    )
```

See the *Built-in widgets* for more information about which widgets are available and which arguments they accept.

Widgets inheriting from the `Select` widget

Widgets inheriting from the *Select* widget deal with choices. They present the user with a list of options to choose from. The different widgets present this choice differently; the *Select* widget itself uses a `<select>` HTML list representation, while *RadioSelect* uses radio buttons.

Select widgets are used by default on *ChoiceField* fields. The choices displayed on the widget are inherited from the *ChoiceField* and changing *ChoiceField.choices* will update *Select.choices*. For example:

```
>>> from django import forms
>>> CHOICES = (('1', 'First',), ('2', 'Second',))
>>> choice_field = forms.ChoiceField(widget=forms.RadioSelect, choices=CHOICES)
>>> choice_field.choices
[('1', 'First'), ('2', 'Second')]
>>> choice_field.widget.choices
[('1', 'First'), ('2', 'Second')]
>>> choice_field.widget.choices = ()
>>> choice_field.choices = (('1', 'First and only',),)
>>> choice_field.widget.choices
[('1', 'First and only')]
```

Widgets which offer a *choices* attribute can however be used with fields which are not based on choice – such as a *CharField* – but it is recommended to use a *ChoiceField*-based field when the choices are inherent to the model and not just the representational widget.

Customizing widget instances

When Django renders a widget as HTML, it only renders very minimal markup - Django doesn't add class names, or any other widget-specific attributes. This means, for example, that all *TextInput* widgets will appear the same on

your Web pages.

There are two ways to customize widgets: *per widget instance* and *per widget class*.

Styling widget instances

If you want to make one widget instance look different from another, you will need to specify additional attributes at the time when the widget object is instantiated and assigned to a form field (and perhaps add some rules to your CSS files).

For example, take the following simple form:

```
from django import forms

class CommentForm(forms.Form):
    name = forms.CharField()
    url = forms.URLField()
    comment = forms.CharField()
```

This form will include three default *TextInput* widgets, with default rendering – no CSS class, no extra attributes. This means that the input boxes provided for each widget will be rendered exactly the same:

```
>>> f = CommentForm(auto_id=False)
>>> f.as_table()
<tr><th>Name:</th><td><input type="text" name="name" required /></td></tr>
<tr><th>Url:</th><td><input type="url" name="url" required /></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" required /></td></tr>
```

On a real Web page, you probably don't want every widget to look the same. You might want a larger input element for the comment, and you might want the 'name' widget to have some special CSS class. It is also possible to specify the 'type' attribute to take advantage of the new HTML5 input types. To do this, you use the *Widget.attrs* argument when creating the widget:

```
class CommentForm(forms.Form):
    name = forms.CharField(widget=forms.TextInput(attrs={'class': 'special'}))
    url = forms.URLField()
    comment = forms.CharField(widget=forms.TextInput(attrs={'size': '40'}))
```

Django will then include the extra attributes in the rendered output:

```
>>> f = CommentForm(auto_id=False)
>>> f.as_table()
<tr><th>Name:</th><td><input type="text" name="name" class="special" required /></td></tr>
<tr><th>Url:</th><td><input type="url" name="url" required /></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" size="40" required /></td></tr>
```

You can also set the HTML `id` using *attrs*. See *BoundField.id_for_label* for an example.

Styling widget classes

With widgets, it is possible to add assets (`css` and `javascript`) and more deeply customize their appearance and behavior.

In a nutshell, you will need to subclass the widget and either *define a "Media" inner class* or *create a "media" property*.

These methods involve somewhat advanced Python programming and are described in detail in the Form Assets topic guide.

Base widget classes

Base widget classes *Widget* and *MultiWidget* are subclassed by all the *built-in widgets* and may serve as a foundation for custom widgets.

Widget

class Widget(*attrs=None*)

This abstract class cannot be rendered, but provides the basic attribute *attrs*. You may also implement or override the *render()* method on custom widgets.

attrs

A dictionary containing HTML attributes to be set on the rendered widget.

```
>>> from django import forms
>>> name = forms.TextInput(attrs={'size': 10, 'title': 'Your name',})
>>> name.render('name', 'A name')
'<input title="Your name" type="text" name="name" value="A name" size="10" required />'
```

If you assign a value of `True` or `False` to an attribute, it will be rendered as an HTML5 boolean attribute:

```
>>> name = forms.TextInput(attrs={'required': True})
>>> name.render('name', 'A name')
'<input name="name" type="text" value="A name" required />'
>>>
>>> name = forms.TextInput(attrs={'required': False})
>>> name.render('name', 'A name')
'<input name="name" type="text" value="A name" />'
```

supports_microseconds

An attribute that defaults to `True`. If set to `False`, the microseconds part of `datetime` and `time` values will be set to `0`.

In older versions, this attribute was only defined on the date and time widgets (as `False`).

format_value(*value*)

Cleans and returns a value for use in the widget template. `value` isn't guaranteed to be valid input, therefore subclass implementations should program defensively.

In older versions, this method is a private API named `_format_value()`. The old name will work until Django 2.0.

id_for_label(*self*, *id_*)

Returns the HTML ID attribute of this widget for use by a `<label>`, given the ID of the field. Returns `None` if an ID isn't available.

This hook is necessary because some widgets have multiple HTML elements and, thus, multiple IDs. In that case, this method should return an ID value that corresponds to the first ID in the widget's tags.

render(*name*, *value*, *attrs=None*)

Returns HTML for the widget, as a Unicode string. This method must be implemented by the subclass, otherwise `NotImplementedError` will be raised.

The 'value' given is not guaranteed to be valid input, therefore subclass implementations should program defensively.

value_from_datadict(*data*, *files*, *name*)

Given a dictionary of data and this widget's name, returns the value of this widget. `files` may contain data coming from *request.FILES*. Returns `None` if a value wasn't provided. Note also that

`value_from_datadict` may be called more than once during handling of form data, so if you customize it and add expensive processing, you should implement some caching mechanism yourself.

value_omitted_from_data(*data*, *files*, *name*)
Given `data` and `files` dictionaries and this widget's name, returns whether or not there's data or files for the widget.

The method's result affects whether or not a field in a model form *falls back to its default*.

Special cases are *`CheckboxInput`* and *`CheckboxSelectMultiple`*, which always return `False` because an unchecked checkbox doesn't appear in the data of an HTML form submission, so it's unknown whether or not the user actually submitted a value.

use_required_attribute(*initial*)
Given a form field's `initial` value, returns whether or not the widget can be rendered with the `required` HTML attribute. Forms use this method along with *`Field.required`* and *`Form.use_required_attribute`* to determine whether or not to display the `required` attribute for each field.

By default, returns `False` for hidden widgets and `True` otherwise. Special cases are *`ClearableFileInput`*, which returns `False` when `initial` is not set, and *`CheckboxSelectMultiple`*, which always returns `False` because browser validation would require all checkboxes to be checked instead of at least one.

Override this method in custom widgets that aren't compatible with browser validation. For example, a WSYSIWG text editor widget backed by a hidden `textarea` element may want to always return `False` to avoid browser validation on the hidden field.

MultiWidget

class MultiWidget(*widgets*, *attrs=None*)
A widget that is composed of multiple widgets. *`MultiWidget`* works hand in hand with the *`MultiValueField`*.

`MultiWidget` has one required argument:

widgets
An iterable containing the widgets needed.

And one required method:

decompress(*value*)
This method takes a single "compressed" value from the field and returns a list of "decompressed" values. The input value can be assumed valid, but not necessarily non-empty.

This method **must be implemented** by the subclass, and since the value may be empty, the implementation must be defensive.

The rationale behind "decompression" is that it is necessary to "split" the combined value of the form field into the values for each widget.

An example of this is how *`SplitDateTimeWidget`* turns a `datetime` value into a list with date and time split into two separate values:

```
from django.forms import MultiWidget

class SplitDateTimeWidget(MultiWidget):

    # ...
```

```
        def decompress(self, value):
            if value:
                return [value.date(), value.time().replace(microsecond=0)]
            return [None, None]
```

: Note that *MultiValueField* has a complementary method *compress()* with the opposite responsibility - to combine cleaned values of all member fields into one.

Other methods that may be useful to override include:

render(*name*, *value*, *attrs=None*)

Argument `value` is handled differently in this method from the subclasses of *Widget* because it has to figure out how to split a single value for display in multiple widgets.

The `value` argument used when rendering can be one of two things:

- A `list`.
- A single value (e.g., a string) that is the "compressed" representation of a `list` of values.

If `value` is a list, the output of *render()* will be a concatenation of rendered child widgets. If `value` is not a list, it will first be processed by the method *decompress()* to create the list and then rendered.

When `render()` executes its HTML rendering, each value in the list is rendered with the corresponding widget – the first value is rendered in the first widget, the second value is rendered in the second widget, etc.

Unlike in the single value widgets, method *render()* need not be implemented in the subclasses.

format_output(*rendered_widgets*)

Given a list of rendered widgets (as strings), returns a Unicode string representing the HTML for the whole lot.

This hook allows you to format the HTML design of the widgets any way you'd like.

Here's an example widget which subclasses *MultiWidget* to display a date with the day, month, and year in different select boxes. This widget is intended to be used with a *DateField* rather than a *MultiValueField*, thus we have implemented *value_from_datadict()*:

```
from datetime import date
from django.forms import widgets

class DateSelectorWidget(widgets.MultiWidget):
    def __init__(self, attrs=None):
        # create choices for days, months, years
        # example below, the rest snipped for brevity.
        years = [(year, year) for year in (2011, 2012, 2013)]
        _widgets = (
            widgets.Select(attrs=attrs, choices=days),
            widgets.Select(attrs=attrs, choices=months),
            widgets.Select(attrs=attrs, choices=years),
        )
        super(DateSelectorWidget, self).__init__(_widgets, attrs)

    def decompress(self, value):
        if value:
            return [value.day, value.month, value.year]
        return [None, None, None]
```

```
        def format_output(self, rendered_widgets):
            return ''.join(rendered_widgets)

        def value_from_datadict(self, data, files, name):
            datelist = [
                widget.value_from_datadict(data, files, name + '_%s' % i)
                for i, widget in enumerate(self.widgets)]
            try:
                D = date(
                    day=int(datelist[0]),
                    month=int(datelist[1]),
                    year=int(datelist[2]),
                )
            except ValueError:
                return ''
            else:
                return str(D)
```

The constructor creates several *Select* widgets in a tuple. The `super` class uses this tuple to setup the widget.

The *format_output()* method is fairly vanilla here (in fact, it's the same as what's been implemented as the default for `MultiWidget`), but the idea is that you could add custom HTML between the widgets should you wish.

The required method *decompress()* breaks up a `datetime.date` value into the day, month, and year values corresponding to each widget. Note how the method handles the case where `value` is `None`.

The default implementation of *value_from_datadict()* returns a list of values corresponding to each `Widget`. This is appropriate when using a `MultiWidget` with a *MultiValueField*, but since we want to use this widget with a *DateField* which takes a single value, we have overridden this method to combine the data of all the subwidgets into a `datetime.date`. The method extracts data from the `POST` dictionary and constructs and validates the date. If it is valid, we return the string, otherwise, we return an empty string which will cause `form.is_valid` to return `False`.

Built-in widgets

Django provides a representation of all the basic HTML widgets, plus some commonly used groups of widgets in the `django.forms.widgets` module, including *the input of text*, *various checkboxes and selectors*, *uploading files*, and *handling of multi-valued input*.

Widgets handling input of text

These widgets make use of the HTML elements `input` and `textarea`.

TextInput
class TextInput
Text input: `<input type="text" ...>`

NumberInput
class NumberInput
Text input: `<input type="number" ...>`

Beware that not all browsers support entering localized numbers in `number` input types. Django itself avoids using them for fields having their *localize* property set to `True`.

EmailInput
class EmailInput
Text input: `<input type="email" ...>`

URLInput
class URLInput
Text input: `<input type="url" ...>`

PasswordInput
class PasswordInput
Password input: `<input type='password' ...>`

Takes one optional argument:

render_value
Determines whether the widget will have a value filled in when the form is re-displayed after a validation error (default is `False`).

HiddenInput
class HiddenInput
Hidden input: `<input type='hidden' ...>`

Note that there also is a *MultipleHiddenInput* widget that encapsulates a set of hidden input elements.

DateInput
class DateInput
Date input as a simple text box: `<input type='text' ...>`

Takes same arguments as *TextInput*, with one more optional argument:

format
The format in which this field's initial value will be displayed.

If no `format` argument is provided, the default format is the first format found in *DATE_INPUT_FORMATS* and respects Format localization.

DateTimeInput
class DateTimeInput
Date/time input as a simple text box: `<input type='text' ...>`

Takes same arguments as *TextInput*, with one more optional argument:

format
The format in which this field's initial value will be displayed.

If no `format` argument is provided, the default format is the first format found in *DATETIME_INPUT_FORMATS* and respects Format localization.

By default, the microseconds part of the time value is always set to `0`. If microseconds are required, use a subclass with the *supports_microseconds* attribute set to `True`.

TimeInput
class TimeInput
Time input as a simple text box: `<input type='text' ...>`

Takes same arguments as *TextInput*, with one more optional argument:

format
The format in which this field's initial value will be displayed.

If no `format` argument is provided, the default format is the first format found in *TIME_INPUT_FORMATS* and respects Format localization.

For the treatment of microseconds, see *DateTimeInput*.

Textarea
class Textarea
Text area: `<textarea>...</textarea>`

Selector and checkbox widgets

CheckboxInput
class CheckboxInput
Checkbox: `<input type='checkbox' ...>`

Takes one optional argument:

check_test
A callable that takes the value of the `CheckboxInput` and returns `True` if the checkbox should be checked for that value.

Select
class Select
Select widget: `<select><option ...>...</select>`

choices
This attribute is optional when the form field does not have a `choices` attribute. If it does, it will override anything you set here when the attribute is updated on the *Field*.

NullBooleanSelect
class NullBooleanSelect
Select widget with options 'Unknown', 'Yes' and 'No'

SelectMultiple
class SelectMultiple
Similar to *Select*, but allows multiple selection: `<select multiple='multiple'>...</select>`

RadioSelect
class RadioSelect
Similar to *Select*, but rendered as a list of radio buttons within `<li>` tags:

```
<ul>
  <li><input type='radio' name='...'></li>
  ...
</ul>
```

For more granular control over the generated markup, you can loop over the radio buttons in the template. Assuming a form `myform` with a field `beatles` that uses a `RadioSelect` as its widget:

```
{% for radio in myform.beatles %}
<div class="myradio">
    {{ radio }}
</div>
{% endfor %}
```

This would generate the following HTML:

```
<div class="myradio">
    <label for="id_beatles_0"><input id="id_beatles_0" name="beatles" type="radio" value="john"
</div>
<div class="myradio">
    <label for="id_beatles_1"><input id="id_beatles_1" name="beatles" type="radio" value="paul"
</div>
<div class="myradio">
    <label for="id_beatles_2"><input id="id_beatles_2" name="beatles" type="radio" value="george
</div>
<div class="myradio">
    <label for="id_beatles_3"><input id="id_beatles_3" name="beatles" type="radio" value="ringo"
</div>
```

That included the `<label>` tags. To get more granular, you can use each radio button's `tag`, `choice_label` and `id_for_label` attributes. For example, this template...

```
{% for radio in myform.beatles %}
    <label for="{{ radio.id_for_label }}">
        {{ radio.choice_label }}
        <span class="radio">{{ radio.tag }}</span>
    </label>
{% endfor %}
```

...will result in the following HTML:

```
<label for="id_beatles_0">
    John
    <span class="radio"><input id="id_beatles_0" name="beatles" type="radio" value="john" requir
</label>

<label for="id_beatles_1">
    Paul
    <span class="radio"><input id="id_beatles_1" name="beatles" type="radio" value="paul" requir
</label>

<label for="id_beatles_2">
    George
    <span class="radio"><input id="id_beatles_2" name="beatles" type="radio" value="george" requ
</label>

<label for="id_beatles_3">
    Ringo
    <span class="radio"><input id="id_beatles_3" name="beatles" type="radio" value="ringo" requi
</label>
```

If you decide not to loop over the radio buttons – e.g., if your template simply includes `{{ myform.beatles }}` – they'll be output in a `<ul>` with `<li>` tags, as above.

The outer `<ul>` container receives the `id` attribute of the widget, if defined, or *`BoundField.auto_id`* otherwise.

When looping over the radio buttons, the `label` and `input` tags include `for` and `id` attributes, respectively.

Each radio button has an `id_for_label` attribute to output the element's ID.

CheckboxSelectMultiple
class CheckboxSelectMultiple
Similar to *SelectMultiple*, but rendered as a list of check buttons:

```
<ul>
  <li><input type='checkbox' name='...' ></li>
  ...
</ul>
```

The outer `<ul>` container receives the `id` attribute of the widget, if defined, or *BoundField.auto_id* otherwise.

Like *RadioSelect*, you can loop over the individual checkboxes for the widget's choices. Unlike *RadioSelect*, the checkboxes won't include the `required` HTML attribute if the field is required because browser validation would require all checkboxes to be checked instead of at least one.

When looping over the checkboxes, the `label` and `input` tags include `for` and `id` attributes, respectively. Each checkbox has an `id_for_label` attribute to output the element's ID.

File upload widgets

FileInput
class FileInput
File upload input: `<input type='file' ...>`

ClearableFileInput
class ClearableFileInput
File upload input: `<input type='file' ...>`, with an additional checkbox input to clear the field's value, if the field is not required and has initial data.

Composite widgets

MultipleHiddenInput
class MultipleHiddenInput
Multiple `<input type='hidden' ...>` widgets.

A widget that handles multiple hidden widgets for fields that have a list of values.

choices
This attribute is optional when the form field does not have a `choices` attribute. If it does, it will override anything you set here when the attribute is updated on the *Field*.

SplitDateTimeWidget
class SplitDateTimeWidget
Wrapper (using *MultiWidget*) around two widgets: *DateInput* for the date, and *TimeInput* for the time. Must be used with *SplitDateTimeField* rather than *DateTimeField*.

`SplitDateTimeWidget` has two optional attributes:

date_format
Similar to *DateInput.format*

time_format
Similar to *TimeInput.format*

SplitHiddenDateTimeWidget
class SplitHiddenDateTimeWidget
Similar to *SplitDateTimeWidget*, but uses *HiddenInput* for both date and time.

SelectDateWidget
class SelectDateWidget
Wrapper around three *Select* widgets: one each for month, day, and year.

Takes several optional arguments:

years
An optional list/tuple of years to use in the "year" select box. The default is a list containing the current year and the next 9 years.

months
An optional dict of months to use in the "months" select box.

The keys of the dict correspond to the month number (1-indexed) and the values are the displayed months:

```
MONTHS = {
    1:_('jan'), 2:_('feb'), 3:_('mar'), 4:_('apr'),
    5:_('may'), 6:_('jun'), 7:_('jul'), 8:_('aug'),
    9:_('sep'), 10:_('oct'), 11:_('nov'), 12:_('dec')
}
```

empty_label
If the *DateField* is not required, *SelectDateWidget* will have an empty choice at the top of the list (which is `---` by default). You can change the text of this label with the `empty_label` attribute. `empty_label` can be a `string`, `list`, or `tuple`. When a string is used, all select boxes will each have an empty choice with this label. If `empty_label` is a `list` or `tuple` of 3 string elements, the select boxes will have their own custom label. The labels should be in this order `('year_label', 'month_label', 'day_label')`.

```
# A custom empty label with string
field1 = forms.DateField(widget=SelectDateWidget(empty_label="Nothing"))

# A custom empty label with tuple
field1 = forms.DateField(
    widget=SelectDateWidget(
        empty_label=("Choose Year", "Choose Month", "Choose Day"),
    ),
)
```

This widget used to be located in the `django.forms.extras.widgets` package. It is now defined in `django.forms.widgets` and like the other widgets it can be imported directly from `django.forms`.

6.12.6 Form and field validation

Form validation happens when the data is cleaned. If you want to customize this process, there are various places to make changes, each one serving a different purpose. Three types of cleaning methods are run during form processing. These are normally executed when you call the `is_valid()` method on a form. There are other things that can also trigger cleaning and validation (accessing the `errors` attribute or calling `full_clean()` directly), but normally they won't be needed.

In general, any cleaning method can raise `ValidationError` if there is a problem with the data it is processing, passing the relevant information to the `ValidationError` constructor. *See below* for the best practice in raising `ValidationError`. If no `ValidationError` is raised, the method should return the cleaned (normalized) data as a Python object.

Most validation can be done using *validators* - simple helpers that can be reused easily. Validators are simple functions (or callables) that take a single argument and raise `ValidationError` on invalid input. Validators are run after the field's `to_python` and `validate` methods have been called.

Validation of a form is split into several steps, which can be customized or overridden:

- The `to_python()` method on a `Field` is the first step in every validation. It coerces the value to a correct datatype and raises `ValidationError` if that is not possible. This method accepts the raw value from the widget and returns the converted value. For example, a `FloatField` will turn the data into a Python `float` or raise a `ValidationError`.
- The `validate()` method on a `Field` handles field-specific validation that is not suitable for a validator. It takes a value that has been coerced to a correct datatype and raises `ValidationError` on any error. This method does not return anything and shouldn't alter the value. You should override it to handle validation logic that you can't or don't want to put in a validator.
- The `run_validators()` method on a `Field` runs all of the field's validators and aggregates all the errors into a single `ValidationError`. You shouldn't need to override this method.
- The `clean()` method on a `Field` subclass is responsible for running `to_python()`, `validate()`, and `run_validators()` in the correct order and propagating their errors. If, at any time, any of the methods raise `ValidationError`, the validation stops and that error is raised. This method returns the clean data, which is then inserted into the `cleaned_data` dictionary of the form.
- The `clean_<fieldname>()` method is called on a form subclass – where `<fieldname>` is replaced with the name of the form field attribute. This method does any cleaning that is specific to that particular attribute, unrelated to the type of field that it is. This method is not passed any parameters. You will need to look up the value of the field in `self.cleaned_data` and remember that it will be a Python object at this point, not the original string submitted in the form (it will be in `cleaned_data` because the general field `clean()` method, above, has already cleaned the data once).

  For example, if you wanted to validate that the contents of a `CharField` called `serialnumber` was unique, `clean_serialnumber()` would be the right place to do this. You don't need a specific field (it's just a `CharField`), but you want a formfield-specific piece of validation and, possibly, cleaning/normalizing the data.

  This method should return the cleaned value obtained from `cleaned_data`, regardless of whether it changed anything or not.
- The form subclass's `clean()` method can perform validation that requires access to multiple form fields. This is where you might put in checks such as "if field `A` is supplied, field `B` must contain a valid email address". This method can return a completely different dictionary if it wishes, which will be used as the `cleaned_data`.

  Since the field validation methods have been run by the time `clean()` is called, you also have access to the form's `errors` attribute which contains all the errors raised by cleaning of individual fields.

  Note that any errors raised by your *`Form.clean()`* override will not be associated with any field in particular. They go into a special "field" (called `__all__`), which you can access via the *`non_field_errors()`* method if you need to. If you want to attach errors to a specific field in the form, you need to call *`add_error()`*.

  Also note that there are special considerations when overriding the `clean()` method of a `ModelForm` subclass. (see the *ModelForm documentation* for more information)

These methods are run in the order given above, one field at a time. That is, for each field in the form (in the order they are declared in the form definition), the `Field.clean()` method (or its override) is run, then

`clean_<fieldname>()`. Finally, once those two methods are run for every field, the *`Form.clean()`* method, or its override, is executed whether or not the previous methods have raised errors.

Examples of each of these methods are provided below.

As mentioned, any of these methods can raise a `ValidationError`. For any field, if the `Field.clean()` method raises a `ValidationError`, any field-specific cleaning method is not called. However, the cleaning methods for all remaining fields are still executed.

Raising `ValidationError`

In order to make error messages flexible and easy to override, consider the following guidelines:

- Provide a descriptive error `code` to the constructor:

```
    # Good
    ValidationError(_('Invalid value'), code='invalid')

    # Bad
    ValidationError(_('Invalid value'))
```

- Don't coerce variables into the message; use placeholders and the `params` argument of the constructor:

```
    # Good
    ValidationError(
        _('Invalid value: %(value)s'),
        params={'value': '42'},
    )

    # Bad
    ValidationError(_('Invalid value: %s') % value)
```

- Use mapping keys instead of positional formatting. This enables putting the variables in any order or omitting them altogether when rewriting the message:

```
    # Good
    ValidationError(
        _('Invalid value: %(value)s'),
        params={'value': '42'},
    )

    # Bad
    ValidationError(
        _('Invalid value: %s'),
        params=('42',),
    )
```

- Wrap the message with `gettext` to enable translation:

```
    # Good
    ValidationError(_('Invalid value'))

    # Bad
    ValidationError('Invalid value')
```

Putting it all together:

```
raise ValidationError(
    _('Invalid value: %(value)s'),
    code='invalid',
```

```
    params={'value': '42'},
)
```

Following these guidelines is particularly necessary if you write reusable forms, form fields, and model fields.

While not recommended, if you are at the end of the validation chain (i.e. your form `clean()` method) and you know you will *never* need to override your error message you can still opt for the less verbose:

```
ValidationError(_('Invalid value: %s') % value)
```

The *`Form.errors.as_data()`* and *`Form.errors.as_json()`* methods greatly benefit from fully featured `ValidationErrors` (with a `code` name and a `params` dictionary).

Raising multiple errors

If you detect multiple errors during a cleaning method and wish to signal all of them to the form submitter, it is possible to pass a list of errors to the `ValidationError` constructor.

As above, it is recommended to pass a list of `ValidationError` instances with `codes` and `params` but a list of strings will also work:

```
# Good
raise ValidationError([
    ValidationError(_('Error 1'), code='error1'),
    ValidationError(_('Error 2'), code='error2'),
])

# Bad
raise ValidationError([
    _('Error 1'),
    _('Error 2'),
])
```

Using validation in practice

The previous sections explained how validation works in general for forms. Since it can sometimes be easier to put things into place by seeing each feature in use, here are a series of small examples that use each of the previous features.

Using validators

Django's form (and model) fields support use of simple utility functions and classes known as validators. A validator is merely a callable object or function that takes a value and simply returns nothing if the value is valid or raises a *`ValidationError`* if not. These can be passed to a field's constructor, via the field's `validators` argument, or defined on the *`Field`* class itself with the `default_validators` attribute.

Simple validators can be used to validate values inside the field, let's have a look at Django's `SlugField`:

```
from django.forms import CharField
from django.core import validators

class SlugField(CharField):
    default_validators = [validators.validate_slug]
```

As you can see, `SlugField` is just a `CharField` with a customized validator that validates that submitted text obeys to some character rules. This can also be done on field definition so:

```
slug = forms.SlugField()
```

is equivalent to:

```
slug = forms.CharField(validators=[validators.validate_slug])
```

Common cases such as validating against an email or a regular expression can be handled using existing validator classes available in Django. For example, `validators.validate_slug` is an instance of a *`RegexValidator`* constructed with the first argument being the pattern: `^[-a-zA-Z0-9_]+$`. See the section on writing validators to see a list of what is already available and for an example of how to write a validator.

Form field default cleaning

Let's first create a custom form field that validates its input is a string containing comma-separated email addresses. The full class looks like this:

```
from django import forms
from django.core.validators import validate_email

class MultiEmailField(forms.Field):
    def to_python(self, value):
        """Normalize data to a list of strings."""
        # Return an empty list if no input was given.
        if not value:
            return []
        return value.split(',')

    def validate(self, value):
        """Check if value consists only of valid emails."""
        # Use the parent's handling of required fields, etc.
        super(MultiEmailField, self).validate(value)
        for email in value:
            validate_email(email)
```

Every form that uses this field will have these methods run before anything else can be done with the field's data. This is cleaning that is specific to this type of field, regardless of how it is subsequently used.

Let's create a simple `ContactForm` to demonstrate how you'd use this field:

```
class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField()
    sender = forms.EmailField()
    recipients = MultiEmailField()
    cc_myself = forms.BooleanField(required=False)
```

Simply use `MultiEmailField` like any other form field. When the `is_valid()` method is called on the form, the `MultiEmailField.clean()` method will be run as part of the cleaning process and it will, in turn, call the custom `to_python()` and `validate()` methods.

Cleaning a specific field attribute

Continuing on from the previous example, suppose that in our `ContactForm`, we want to make sure that the `recipients` field always contains the address `"fred@example.com"`. This is validation that is specific to

our form, so we don't want to put it into the general `MultiEmailField` class. Instead, we write a cleaning method that operates on the `recipients` field, like so:

```
from django import forms

class ContactForm(forms.Form):
    # Everything as before.
    ...

    def clean_recipients(self):
        data = self.cleaned_data['recipients']
        if "fred@example.com" not in data:
            raise forms.ValidationError("You have forgotten about Fred!")

        # Always return the cleaned data, whether you have changed it or
        # not.
        return data
```

Cleaning and validating fields that depend on each other

Suppose we add another requirement to our contact form: if the `cc_myself` field is `True`, the `subject` must contain the word `"help"`. We are performing validation on more than one field at a time, so the form's *`clean()`* method is a good spot to do this. Notice that we are talking about the `clean()` method on the form here, whereas earlier we were writing a `clean()` method on a field. It's important to keep the field and form difference clear when working out where to validate things. Fields are single data points, forms are a collection of fields.

By the time the form's `clean()` method is called, all the individual field clean methods will have been run (the previous two sections), so `self.cleaned_data` will be populated with any data that has survived so far. So you also need to remember to allow for the fact that the fields you are wanting to validate might not have survived the initial individual field checks.

There are two ways to report any errors from this step. Probably the most common method is to display the error at the top of the form. To create such an error, you can raise a `ValidationError` from the `clean()` method. For example:

```
from django import forms

class ContactForm(forms.Form):
    # Everything as before.
    ...

    def clean(self):
        cleaned_data = super(ContactForm, self).clean()
        cc_myself = cleaned_data.get("cc_myself")
        subject = cleaned_data.get("subject")

        if cc_myself and subject:
            # Only do something if both fields are valid so far.
            if "help" not in subject:
                raise forms.ValidationError(
                    "Did not send for 'help' in the subject despite "
                    "CC'ing yourself."
                )
```

In this code, if the validation error is raised, the form will display an error message at the top of the form (normally) describing the problem.

The call to `super(ContactForm, self).clean()` in the example code ensures that any validation logic in

parent classes is maintained. If your form inherits another that doesn't return a `cleaned_data` dictionary in its `clean()` method (doing so is optional), then don't assign `cleaned_data` to the result of the `super()` call and use `self.cleaned_data` instead:

```
def clean(self):
    super(ContactForm, self).clean()
    cc_myself = self.cleaned_data.get("cc_myself")
    ...
```

The second approach for reporting validation errors might involve assigning the error message to one of the fields. In this case, let's assign an error message to both the "subject" and "cc_myself" rows in the form display. Be careful when doing this in practice, since it can lead to confusing form output. We're showing what is possible here and leaving it up to you and your designers to work out what works effectively in your particular situation. Our new code (replacing the previous sample) looks like this:

```
from django import forms

class ContactForm(forms.Form):
    # Everything as before.
    ...

    def clean(self):
        cleaned_data = super(ContactForm, self).clean()
        cc_myself = cleaned_data.get("cc_myself")
        subject = cleaned_data.get("subject")

        if cc_myself and subject and "help" not in subject:
            msg = "Must put 'help' in subject when cc'ing yourself."
            self.add_error('cc_myself', msg)
            self.add_error('subject', msg)
```

The second argument of `add_error()` can be a simple string, or preferably an instance of `ValidationError`. See *Raising ValidationError* for more details. Note that `add_error()` automatically removes the field from `cleaned_data`.

6.13 Middleware

This document explains all middleware components that come with Django. For information on how to use them and how to write your own middleware, see the middleware usage guide.

6.13.1 Available middleware

Cache middleware

class UpdateCacheMiddleware

class FetchFromCacheMiddleware

Enable the site-wide cache. If these are enabled, each Django-powered page will be cached for as long as the *CACHE_MIDDLEWARE_SECONDS* setting defines. See the cache documentation.

"Common" middleware

class CommonMiddleware

Adds a few conveniences for perfectionists:

- Forbids access to user agents in the *DISALLOWED_USER_AGENTS* setting, which should be a list of compiled regular expression objects.
- Performs URL rewriting based on the *APPEND_SLASH* and *PREPEND_WWW* settings.

  If *APPEND_SLASH* is `True` and the initial URL doesn't end with a slash, and it is not found in the URLconf, then a new URL is formed by appending a slash at the end. If this new URL is found in the URLconf, then Django redirects the request to this new URL. Otherwise, the initial URL is processed as usual.

  For example, `foo.com/bar` will be redirected to `foo.com/bar/` if you don't have a valid URL pattern for `foo.com/bar` but *do* have a valid pattern for `foo.com/bar/`.

  If *PREPEND_WWW* is `True`, URLs that lack a leading "www." will be redirected to the same URL with a leading "www."

  Both of these options are meant to normalize URLs. The philosophy is that each URL should exist in one, and only one, place. Technically a URL `foo.com/bar` is distinct from `foo.com/bar/` – a search-engine indexer would treat them as separate URLs – so it's best practice to normalize URLs.
- Handles ETags based on the *USE_ETAGS* setting. If *USE_ETAGS* is set to `True`, Django will calculate an ETag for each request by MD5-hashing the page content, and it'll take care of sending `Not Modified` responses, if appropriate.

`CommonMiddleware.`**`response_redirect_class`**

Defaults to *HttpResponsePermanentRedirect*. Subclass `CommonMiddleware` and override the attribute to customize the redirects issued by the middleware.

class `BrokenLinkEmailsMiddleware`

- Sends broken link notification emails to *MANAGERS* (see Error reporting).

Exception middleware

class `ExceptionMiddleware`

Catches exceptions raised during the request/response cycle and returns the appropriate response.

- *Http404* is processed by *handler404* (or a more friendly debug page if *DEBUG=True*).
- *PermissionDenied* is processed by *handler403*.
- `MultiPartParserError` is processed by *handler400*.
- *SuspiciousOperation* is processed by *handler400* (or a more friendly debug page if *DEBUG=True*).
- Any other exception is processed by *handler500* (or a more friendly debug page if *DEBUG=True*).

Django uses this middleware regardless of whether or not you include it in *MIDDLEWARE*, however, you may want to subclass if your own middleware needs to transform any of these exceptions into the appropriate responses. *LocaleMiddleware* does this, for example.

GZip middleware

class `GZipMiddleware`

: Security researchers recently revealed that when compression techniques (including `GZipMiddleware`) are used on a website, the site may become exposed to a number of possible attacks. Before using `GZipMiddleware` on your site, you should consider very carefully whether you are subject to these attacks. If you're in *any* doubt about whether you're affected, you should avoid using `GZipMiddleware`. For more details, see the the BREACH paper (PDF) and breachattack.com.

Compresses content for browsers that understand GZip compression (all modern browsers).

This middleware should be placed before any other middleware that need to read or write the response body so that compression happens afterward.

It will NOT compress content if any of the following are true:

- The content body is less than 200 bytes long.
- The response has already set the `Content-Encoding` header.
- The request (the browser) hasn't sent an `Accept-Encoding` header containing `gzip`.

You can apply GZip compression to individual views using the *`gzip_page()`* decorator.

In older versions, Django's CSRF protection mechanism was vulnerable to BREACH attacks when compression was used. This is no longer the case, but you should still take care not to compromise your own secrets this way.

Conditional GET middleware

class `ConditionalGetMiddleware`

Handles conditional GET operations. If the response has a `ETag` or `Last-Modified` header, and the request has `If-None-Match` or `If-Modified-Since`, the response is replaced by an *`HttpResponseNotModified`*.

Also sets the `Date` and `Content-Length` response-headers.

Locale middleware

class `LocaleMiddleware`

Enables language selection based on data from the request. It customizes content for each user. See the internationalization documentation.

`LocaleMiddleware.`**`response_redirect_class`**

Defaults to *`HttpResponseRedirect`*. Subclass `LocaleMiddleware` and override the attribute to customize the redirects issued by the middleware.

Message middleware

class `MessageMiddleware`

Enables cookie- and session-based message support. See the messages documentation.

Security middleware

: If your deployment situation allows, it's usually a good idea to have your front-end Web server perform the functionality provided by the `SecurityMiddleware`. That way, if there are requests that aren't served by Django (such as static media or user-uploaded files), they will have the same protections as requests to your Django application.

class `SecurityMiddleware`

The `django.middleware.security.SecurityMiddleware` provides several security enhancements to the request/response cycle. Each one can be independently enabled or disabled with a setting.

- *SECURE_BROWSER_XSS_FILTER*
- *SECURE_CONTENT_TYPE_NOSNIFF*
- *SECURE_HSTS_INCLUDE_SUBDOMAINS*
- *SECURE_HSTS_SECONDS*
- *SECURE_REDIRECT_EXEMPT*
- *SECURE_SSL_HOST*
- *SECURE_SSL_REDIRECT*

HTTP Strict Transport Security

For sites that should only be accessed over HTTPS, you can instruct modern browsers to refuse to connect to your domain name via an insecure connection (for a given period of time) by setting the "Strict-Transport-Security" header. This reduces your exposure to some SSL-stripping man-in-the-middle (MITM) attacks.

`SecurityMiddleware` will set this header for you on all HTTPS responses if you set the *SECURE_HSTS_SECONDS* setting to a non-zero integer value.

When enabling HSTS, it's a good idea to first use a small value for testing, for example, *SECURE_HSTS_SECONDS = 3600* for one hour. Each time a Web browser sees the HSTS header from your site, it will refuse to communicate non-securely (using HTTP) with your domain for the given period of time. Once you confirm that all assets are served securely on your site (i.e. HSTS didn't break anything), it's a good idea to increase this value so that infrequent visitors will be protected (31536000 seconds, i.e. 1 year, is common).

Additionally, if you set the *SECURE_HSTS_INCLUDE_SUBDOMAINS* setting to `True`, `SecurityMiddleware` will add the `includeSubDomains` directive to the `Strict-Transport-Security` header. This is recommended (assuming all subdomains are served exclusively using HTTPS), otherwise your site may still be vulnerable via an insecure connection to a subdomain.

: The HSTS policy applies to your entire domain, not just the URL of the response that you set the header on. Therefore, you should only use it if your entire domain is served via HTTPS only.
Browsers properly respecting the HSTS header will refuse to allow users to bypass warnings and connect to a site with an expired, self-signed, or otherwise invalid SSL certificate. If you use HSTS, make sure your certificates are in good shape and stay that way!

: If you are deployed behind a load-balancer or reverse-proxy server, and the `Strict-Transport-Security` header is not being added to your responses, it may be because Django doesn't realize that it's on a secure connection; you may need to set the *SECURE_PROXY_SSL_HEADER* setting.

X-Content-Type-Options: nosniff

Some browsers will try to guess the content types of the assets that they fetch, overriding the `Content-Type` header. While this can help display sites with improperly configured servers, it can also pose a security risk.

If your site serves user-uploaded files, a malicious user could upload a specially-crafted file that would be interpreted as HTML or JavaScript by the browser when you expected it to be something harmless.

To learn more about this header and how the browser treats it, you can read about it on the IE Security Blog.

To prevent the browser from guessing the content type and force it to always use the type provided in the `Content-Type` header, you can pass the `X-Content-Type-Options: nosniff` header. `SecurityMiddleware` will do this for all responses if the *SECURE_CONTENT_TYPE_NOSNIFF* setting is `True`.

Note that in most deployment situations where Django isn't involved in serving user-uploaded files, this setting won't help you. For example, if your *MEDIA_URL* is served directly by your front-end Web server (nginx, Apache, etc.) then you'd want to set this header there. On the other hand, if you are using Django to do something like require authorization in order to download files and you cannot set the header using your Web server, this setting will be useful.

X-XSS-Protection: 1; mode=block

Some browsers have the ability to block content that appears to be an XSS attack. They work by looking for JavaScript content in the GET or POST parameters of a page. If the JavaScript is replayed in the server's response, the page is blocked from rendering and an error page is shown instead.

The X-XSS-Protection header is used to control the operation of the XSS filter.

To enable the XSS filter in the browser, and force it to always block suspected XSS attacks, you can pass the `X-XSS-Protection: 1; mode=block` header. `SecurityMiddleware` will do this for all responses if the *SECURE_BROWSER_XSS_FILTER* setting is `True`.

: The browser XSS filter is a useful defense measure, but must not be relied upon exclusively. It cannot detect all XSS attacks and not all browsers support the header. Ensure you are still *validating and sanitizing* all input to prevent XSS attacks.

SSL Redirect

If your site offers both HTTP and HTTPS connections, most users will end up with an unsecured connection by default. For best security, you should redirect all HTTP connections to HTTPS.

If you set the *SECURE_SSL_REDIRECT* setting to True, `SecurityMiddleware` will permanently (HTTP 301) redirect all HTTP connections to HTTPS.

: For performance reasons, it's preferable to do these redirects outside of Django, in a front-end load balancer or reverse-proxy server such as nginx. *SECURE_SSL_REDIRECT* is intended for the deployment situations where this isn't an option.

If the *SECURE_SSL_HOST* setting has a value, all redirects will be sent to that host instead of the originally-requested host.

If there are a few pages on your site that should be available over HTTP, and not redirected to HTTPS, you can list regular expressions to match those URLs in the *SECURE_REDIRECT_EXEMPT* setting.

: If you are deployed behind a load-balancer or reverse-proxy server and Django can't seem to tell when a request actually is already secure, you may need to set the *SECURE_PROXY_SSL_HEADER* setting.

Session middleware

class SessionMiddleware

Enables session support. See the session documentation.

Site middleware

class CurrentSiteMiddleware

Adds the `site` attribute representing the current site to every incoming `HttpRequest` object. See the *sites documentation*.

Authentication middleware

class AuthenticationMiddleware

Adds the `user` attribute, representing the currently-logged-in user, to every incoming `HttpRequest` object. See *Authentication in Web requests*.

class RemoteUserMiddleware

Middleware for utilizing Web server provided authentication. See Authentication using REMOTE_USER for usage details.

class PersistentRemoteUserMiddleware

Middleware for utilizing Web server provided authentication when enabled only on the login page. See *Using REMOTE_USER on login pages only* for usage details.

CSRF protection middleware

class CsrfViewMiddleware

Adds protection against Cross Site Request Forgeries by adding hidden form fields to POST forms and checking requests for the correct value. See the Cross Site Request Forgery protection documentation.

`X-Frame-Options` middleware

class XFrameOptionsMiddleware

Simple clickjacking protection via the X-Frame-Options header.

6.13.2 Middleware ordering

Here are some hints about the ordering of various Django middleware classes:

1. *SecurityMiddleware*

   It should go near the top of the list if you're going to turn on the SSL redirect as that avoids running through a bunch of other unnecessary middleware.

2. *UpdateCacheMiddleware*

   Before those that modify the `Vary` header (`SessionMiddleware`, `GZipMiddleware`, `LocaleMiddleware`).

3. *GZipMiddleware*

   Before any middleware that may change or use the response body.

   After `UpdateCacheMiddleware`: Modifies `Vary` header.

4. *ConditionalGetMiddleware*

   Before `CommonMiddleware`: uses its `ETag` header when *USE_ETAGS* = `True`.

5. *SessionMiddleware*

   After `UpdateCacheMiddleware`: Modifies `Vary` header.

6. *LocaleMiddleware*

   One of the topmost, after `SessionMiddleware` (uses session data) and `UpdateCacheMiddleware` (modifies `Vary` header).

7. *CommonMiddleware*

   Before any middleware that may change the response (it calculates `ETags`).

   After `GZipMiddleware` so it won't calculate an `ETag` header on gzipped contents.

   Close to the top: it redirects when *APPEND_SLASH* or *PREPEND_WWW* are set to `True`.

8. *CsrfViewMiddleware*

   Before any view middleware that assumes that CSRF attacks have been dealt with.

9. *AuthenticationMiddleware*

   After `SessionMiddleware`: uses session storage.

10. *MessageMiddleware*

    After `SessionMiddleware`: can use session-based storage.

11. *FetchFromCacheMiddleware*

    After any middleware that modifies the `Vary` header: that header is used to pick a value for the cache hash-key.

12. *FlatpageFallbackMiddleware*

    Should be near the bottom as it's a last-resort type of middleware.

13. *RedirectFallbackMiddleware*

    Should be near the bottom as it's a last-resort type of middleware.

6.14 Migration Operations

Migration files are composed of one or more `Operations`, objects that declaratively record what the migration should do to your database.

Django also uses these `Operation` objects to work out what your models looked like historically, and to calculate what changes you've made to your models since the last migration so it can automatically write your migrations; that's why they're declarative, as it means Django can easily load them all into memory and run through them without touching the database to work out what your project should look like.

There are also more specialized `Operation` objects which are for things like *data migrations* and for advanced manual database manipulation. You can also write your own `Operation` classes if you want to encapsulate a custom change you commonly make.

If you need an empty migration file to write your own `Operation` objects into, just use `python manage.py makemigrations --empty yourappname`, but be aware that manually adding schema-altering operations can confuse the migration autodetector and make resulting runs of *`makemigrations`* output incorrect code.

All of the core Django operations are available from the `django.db.migrations.operations` module.

For introductory material, see the migrations topic guide.

6.14.1 Schema Operations

`CreateModel`

class `CreateModel`(*name*, *fields*, *options=None*, *bases=None*, *managers=None*)

Creates a new model in the project history and a corresponding table in the database to match it.

`name` is the model name, as would be written in the `models.py` file.

`fields` is a list of 2-tuples of `(field_name, field_instance)`. The field instance should be an unbound field (so just `models.CharField(...)`, rather than a field taken from another model).

`options` is an optional dictionary of values from the model's `Meta` class.

`bases` is an optional list of other classes to have this model inherit from; it can contain both class objects as well as strings in the format `"appname.ModelName"` if you want to depend on another model (so you inherit from the historical version). If it's not supplied, it defaults to just inheriting from the standard `models.Model`.

`managers` takes a list of 2-tuples of `(manager_name, manager_instance)`. The first manager in the list will be the default manager for this model during migrations.

`DeleteModel`

class `DeleteModel`(*name*)

Deletes the model from the project history and its table from the database.

`RenameModel`

class `RenameModel`(*old_name*, *new_name*)

Renames the model from an old name to a new one.

You may have to manually add this if you change the model's name and quite a few of its fields at once; to the autodetector, this will look like you deleted a model with the old name and added a new one with a different name, and the migration it creates will lose any data in the old table.

AlterModelTable

class AlterModelTable(*name*, *table*)

Changes the model's table name (the *db_table* option on the `Meta` subclass).

AlterUniqueTogether

class AlterUniqueTogether(*name*, *unique_together*)

Changes the model's set of unique constraints (the *unique_together* option on the `Meta` subclass).

AlterIndexTogether

class AlterIndexTogether(*name*, *index_together*)

Changes the model's set of custom indexes (the *index_together* option on the `Meta` subclass).

AlterOrderWithRespectTo

class AlterOrderWithRespectTo(*name*, *order_with_respect_to*)

Makes or deletes the `_order` column needed for the *order_with_respect_to* option on the `Meta` subclass.

AlterModelOptions

class AlterModelOptions(*name*, *options*)

Stores changes to miscellaneous model options (settings on a model's `Meta`) like `permissions` and `verbose_name`. Does not affect the database, but persists these changes for *RunPython* instances to use. `options` should be a dictionary mapping option names to values.

AlterModelManagers

class AlterModelManagers(*name*, *managers*)

Alters the managers that are available during migrations.

AddField

class AddField(*model_name*, *name*, *field*, *preserve_default=True*)

Adds a field to a model. `model_name` is the model's name, `name` is the field's name, and `field` is an unbound Field instance (the thing you would put in the field declaration in `models.py` - for example, `models.IntegerField(null=True)`.

The `preserve_default` argument indicates whether the field's default value is permanent and should be baked into the project state (`True`), or if it is temporary and just for this migration (`False`) - usually because the migration is adding a non-nullable field to a table and needs a default value to put into existing rows. It does not affect the behavior of setting defaults in the database directly - Django never sets database defaults and always applies them in the Django ORM code.

RemoveField

class RemoveField(*model_name*, *name*)

Removes a field from a model.

Bear in mind that when reversed, this is actually adding a field to a model. The operation is reversible (apart from any data loss, which of course is irreversible) if the field is nullable or if it has a default value that can be used to populate the recreated column. If the field is not nullable and does not have a default value, the operation is irreversible.

AlterField

class AlterField(*model_name*, *name*, *field*, *preserve_default=True*)

Alters a field's definition, including changes to its type, `null`, `unique`, `db_column` and other field attributes.

The `preserve_default` argument indicates whether the field's default value is permanent and should be baked into the project state (`True`), or if it is temporary and just for this migration (`False`) - usually because the migration is altering a nullable field to a non-nullable one and needs a default value to put into existing rows. It does not affect the behavior of setting defaults in the database directly - Django never sets database defaults and always applies them in the Django ORM code.

Note that not all changes are possible on all databases - for example, you cannot change a text-type field like `models.TextField()` into a number-type field like `models.IntegerField()` on most databases.

RenameField

class RenameField(*model_name*, *old_name*, *new_name*)

Changes a field's name (and, unless `db_column` is set, its column name).

6.14.2 Special Operations

RunSQL

class RunSQL(*sql*, *reverse_sql=None*, *state_operations=None*, *hints=None*, *elidable=False*)

Allows running of arbitrary SQL on the database - useful for more advanced features of database backends that Django doesn't support directly, like partial indexes.

`sql`, and `reverse_sql` if provided, should be strings of SQL to run on the database. On most database backends (all but PostgreSQL), Django will split the SQL into individual statements prior to executing them. This requires installing the sqlparse Python library.

You can also pass a list of strings or 2-tuples. The latter is used for passing queries and parameters in the same way as *cursor.execute()*. These three operations are equivalent:

```
migrations.RunSQL("INSERT INTO musician (name) VALUES ('Reinhardt');")
migrations.RunSQL([("INSERT INTO musician (name) VALUES ('Reinhardt');", None)])
migrations.RunSQL([("INSERT INTO musician (name) VALUES (%s);", ['Reinhardt'])])
```

If you want to include literal percent signs in the query, you have to double them if you are passing parameters.

The `reverse_sql` queries are executed when the migration is unapplied, so you can reverse the changes done in the forwards queries:

```
migrations.RunSQL(
    [("INSERT INTO musician (name) VALUES (%s);", ['Reinhardt'])],
    [("DELETE FROM musician where name=%s;", ['Reinhardt'])],
)
```

The `state_operations` argument is so you can supply operations that are equivalent to the SQL in terms of project state; for example, if you are manually creating a column, you should pass in a list containing an `AddField` operation here so that the autodetector still has an up-to-date state of the model (otherwise, when you next run `makemigrations`, it won't see any operation that adds that field and so will try to run it again). For example:

```
migrations.RunSQL(
    "ALTER TABLE musician ADD COLUMN name varchar(255) NOT NULL;",
    state_operations=[
        migrations.AddField(
            'musician',
            'name',
            models.CharField(max_length=255),
        ),
    ],
)
```

The optional `hints` argument will be passed as `**hints` to the *`allow_migrate()`* method of database routers to assist them in making routing decisions. See *Hints* for more details on database hints.

The optional `elidable` argument determines whether or not the operation will be removed (elided) when *squashing migrations*.

`RunSQL.noop`
: Pass the `RunSQL.noop` attribute to `sql` or `reverse_sql` when you want the operation not to do anything in the given direction. This is especially useful in making the operation reversible.

The `elidable` argument was added.

RunPython

class RunPython(*code*, *reverse_code=None*, *atomic=None*, *hints=None*, *elidable=False*)

Runs custom Python code in a historical context. `code` (and `reverse_code` if supplied) should be callable objects that accept two arguments; the first is an instance of `django.apps.registry.Apps` containing historical models that match the operation's place in the project history, and the second is an instance of *`SchemaEditor`*.

The `reverse_code` argument is called when unapplying migrations. This callable should undo what is done in the `code` callable so that the migration is reversible.

The optional `hints` argument will be passed as `**hints` to the *`allow_migrate()`* method of database routers to assist them in making a routing decision. See *Hints* for more details on database hints.

The optional `elidable` argument determines whether or not the operation will be removed (elided) when *squashing migrations*.

You are advised to write the code as a separate function above the `Migration` class in the migration file, and just pass it to `RunPython`. Here's an example of using `RunPython` to create some initial objects on a `Country` model:

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.db import migrations, models

def forwards_func(apps, schema_editor):
```

```
    # We get the model from the versioned app registry;
    # if we directly import it, it'll be the wrong version
    Country = apps.get_model("myapp", "Country")
    db_alias = schema_editor.connection.alias
    Country.objects.using(db_alias).bulk_create([
        Country(name="USA", code="us"),
        Country(name="France", code="fr"),
    ])

def reverse_func(apps, schema_editor):
    # forwards_func() creates two Country instances,
    # so reverse_func() should delete them.
    Country = apps.get_model("myapp", "Country")
    db_alias = schema_editor.connection.alias
    Country.objects.using(db_alias).filter(name="USA", code="us").delete()
    Country.objects.using(db_alias).filter(name="France", code="fr").delete()

class Migration(migrations.Migration):

    dependencies = []

    operations = [
        migrations.RunPython(forwards_func, reverse_func),
    ]
```

This is generally the operation you would use to create *data migrations*, run custom data updates and alterations, and anything else you need access to an ORM and/or Python code for.

If you're upgrading from South, this is basically the South pattern as an operation - one or two methods for forwards and backwards, with an ORM and schema operations available. Most of the time, you should be able to translate the `orm.Model` or `orm["appname", "Model"]` references from South directly into `apps.get_model("appname", "Model")` references here and leave most of the rest of the code unchanged for data migrations. However, `apps` will only have references to models in the current app unless migrations in other apps are added to the migration's dependencies.

Much like *`RunSQL`*, ensure that if you change schema inside here you're either doing it outside the scope of the Django model system (e.g. triggers) or that you use *`SeparateDatabaseAndState`* to add in operations that will reflect your changes to the model state - otherwise, the versioned ORM and the autodetector will stop working correctly.

By default, `RunPython` will run its contents inside a transaction on databases that do not support DDL transactions (for example, MySQL and Oracle). This should be safe, but may cause a crash if you attempt to use the `schema_editor` provided on these backends; in this case, pass `atomic=False` to the `RunPython` operation.

On databases that do support DDL transactions (SQLite and PostgreSQL), `RunPython` operations do not have any transactions automatically added besides the transactions created for each migration. Thus, on PostgreSQL, for example, you should avoid combining schema changes and `RunPython` operations in the same migration or you may hit errors like `OperationalError: cannot ALTER TABLE "mytable" because it has pending trigger events`.

If you have a different database and aren't sure if it supports DDL transactions, check the `django.db.connection.features.can_rollback_ddl` attribute.

If the `RunPython` operation is part of a *non-atomic migration*, the operation will only be executed in a transaction if `atomic=True` is passed to the `RunPython` operation.

: `RunPython` does not magically alter the connection of the models for you; any model methods you call will go to the default database unless you give them the current database alias (available from `schema_editor.connection.alias`, where `schema_editor` is the second argument to your function).

static `RunPython.`**`noop`**`()`

Pass the `RunPython.noop` method to `code` or `reverse_code` when you want the operation not to do anything in the given direction. This is especially useful in making the operation reversible.

The `elidable` argument was added.

The `atomic` argument default was changed to `None`, indicating that the atomicity is controlled by the `atomic` attribute of the migration.

SeparateDatabaseAndState

class **`SeparateDatabaseAndState`**(*database_operations=None*, *state_operations=None*)

A highly specialized operation that let you mix and match the database (schema-changing) and state (autodetector-powering) aspects of operations.

It accepts two list of operations, and when asked to apply state will use the state list, and when asked to apply changes to the database will use the database list. Do not use this operation unless you're very sure you know what you're doing.

6.14.3 Writing your own

Operations have a relatively simple API, and they're designed so that you can easily write your own to supplement the built-in Django ones. The basic structure of an `Operation` looks like this:

```
from django.db.migrations.operations.base import Operation

class MyCustomOperation(Operation):

    # If this is False, it means that this operation will be ignored by
    # sqlmigrate; if true, it will be run and the SQL collected for its output.
    reduces_to_sql = False

    # If this is False, Django will refuse to reverse past this operation.
    reversible = False

    def __init__(self, arg1, arg2):
        # Operations are usually instantiated with arguments in migration
        # files. Store the values of them on self for later use.
        pass

    def state_forwards(self, app_label, state):
        # The Operation should take the 'state' parameter (an instance of
        # django.db.migrations.state.ProjectState) and mutate it to match
        # any schema changes that have occurred.
        pass

    def database_forwards(self, app_label, schema_editor, from_state, to_state):
        # The Operation should use schema_editor to apply any changes it
        # wants to make to the database.
        pass

    def database_backwards(self, app_label, schema_editor, from_state, to_state):
        # If reversible is True, this is called when the operation is reversed.
        pass

    def describe(self):
```

```
        # This is used to describe what the operation does in console output.
        return "Custom Operation"
```

You can take this template and work from it, though we suggest looking at the built-in Django operations in `django.db.migrations.operations` - they're easy to read and cover a lot of the example usage of semi-internal aspects of the migration framework like `ProjectState` and the patterns used to get historical models, as well as `ModelState` and the patterns used to mutate historical models in `state_forwards()`.

Some things to note:

- You don't need to learn too much about `ProjectState` to just write simple migrations; just know that it has an `apps` property that gives access to an app registry (which you can then call `get_model` on).
- `database_forwards` and `database_backwards` both get two states passed to them; these just represent the difference the `state_forwards` method would have applied, but are given to you for convenience and speed reasons.
- `to_state` in the database_backwards method is the *older* state; that is, the one that will be the current state once the migration has finished reversing.
- You might see implementations of `references_model` on the built-in operations; this is part of the autodetection code and does not matter for custom operations.

: For performance reasons, the *`Field`* instances in `ModelState.fields` are reused across migrations. You must never change the attributes on these instances. If you need to mutate a field in `state_forwards()`, you must remove the old instance from `ModelState.fields` and add a new instance in its place. The same is true for the *`Manager`* instances in `ModelState.managers`.

As a simple example, let's make an operation that loads PostgreSQL extensions (which contain some of PostgreSQL's more exciting features). It's simple enough; there's no model state changes, and all it does is run one command:

```
from django.db.migrations.operations.base import Operation

class LoadExtension(Operation):

    reversible = True

    def __init__(self, name):
        self.name = name

    def state_forwards(self, app_label, state):
        pass

    def database_forwards(self, app_label, schema_editor, from_state, to_state):
        schema_editor.execute("CREATE EXTENSION IF NOT EXISTS %s" % self.name)

    def database_backwards(self, app_label, schema_editor, from_state, to_state):
        schema_editor.execute("DROP EXTENSION %s" % self.name)

    def describe(self):
        return "Creates extension %s" % self.name
```

6.15 Models

Model API reference. For introductory material, see Models.

6.15.1 Model field reference

This document contains all the API references of *Field* including the *field options* and *field types* Django offers.

:

If the built-in fields don't do the trick, you can try django-localflavor (documentation), which contains assorted pieces of code that are useful for particular countries and cultures.

Also, you can easily write your own custom model fields.

: Technically, these models are defined in *django.db.models.fields*, but for convenience they're imported into *django.db.models*; the standard convention is to use `from django.db import models` and refer to fields as `models.<Foo>Field`.

Field options

The following arguments are available to all field types. All are optional.

null

`Field.`**`null`**

If `True`, Django will store empty values as `NULL` in the database. Default is `False`.

Avoid using *null* on string-based fields such as *CharField* and *TextField* because empty string values will always be stored as empty strings, not as `NULL`. If a string-based field has `null=True`, that means it has two possible values for "no data": `NULL`, and the empty string. In most cases, it's redundant to have two possible values for "no data;" the Django convention is to use the empty string, not `NULL`.

For both string-based and non-string-based fields, you will also need to set `blank=True` if you wish to permit empty values in forms, as the *null* parameter only affects database storage (see *blank*).

: When using the Oracle database backend, the value `NULL` will be stored to denote the empty string regardless of this attribute.

If you want to accept *null* values with *BooleanField*, use *NullBooleanField* instead.

blank

`Field.`**`blank`**

If `True`, the field is allowed to be blank. Default is `False`.

Note that this is different than *null*. *null* is purely database-related, whereas *blank* is validation-related. If a field has `blank=True`, form validation will allow entry of an empty value. If a field has `blank=False`, the field will be required.

choices

`Field.`**`choices`**

An iterable (e.g., a list or tuple) consisting itself of iterables of exactly two items (e.g. `[(A, B), (A, B) ...]`) to use as choices for this field. If this is given, the default form widget will be a select box with these choices instead of the standard text field.

The first element in each tuple is the actual value to be set on the model, and the second element is the human-readable name. For example:

```
YEAR_IN_SCHOOL_CHOICES = (
    ('FR', 'Freshman'),
    ('SO', 'Sophomore'),
    ('JR', 'Junior'),
    ('SR', 'Senior'),
)
```

Generally, it's best to define choices inside a model class, and to define a suitably-named constant for each value:

```
from django.db import models

class Student(models.Model):
    FRESHMAN = 'FR'
    SOPHOMORE = 'SO'
    JUNIOR = 'JR'
    SENIOR = 'SR'
    YEAR_IN_SCHOOL_CHOICES = (
        (FRESHMAN, 'Freshman'),
        (SOPHOMORE, 'Sophomore'),
        (JUNIOR, 'Junior'),
        (SENIOR, 'Senior'),
    )
    year_in_school = models.CharField(
        max_length=2,
        choices=YEAR_IN_SCHOOL_CHOICES,
        default=FRESHMAN,
    )

    def is_upperclass(self):
        return self.year_in_school in (self.JUNIOR, self.SENIOR)
```

Though you can define a choices list outside of a model class and then refer to it, defining the choices and names for each choice inside the model class keeps all of that information with the class that uses it, and makes the choices easy to reference (e.g, `Student.SOPHOMORE` will work anywhere that the `Student` model has been imported).

You can also collect your available choices into named groups that can be used for organizational purposes:

```
MEDIA_CHOICES = (
    ('Audio', (
            ('vinyl', 'Vinyl'),
            ('cd', 'CD'),
        )
    ),
    ('Video', (
            ('vhs', 'VHS Tape'),
            ('dvd', 'DVD'),
        )
    ),
    ('unknown', 'Unknown'),
)
```

The first element in each tuple is the name to apply to the group. The second element is an iterable of 2-tuples, with each 2-tuple containing a value and a human-readable name for an option. Grouped options may be combined with

ungrouped options within a single list (such as the *unknown* option in this example).

For each model field that has *choices* set, Django will add a method to retrieve the human-readable name for the field's current value. See *get_FOO_display()* in the database API documentation.

Note that choices can be any iterable object – not necessarily a list or tuple. This lets you construct choices dynamically. But if you find yourself hacking *choices* to be dynamic, you're probably better off using a proper database table with a *ForeignKey*. *choices* is meant for static data that doesn't change much, if ever.

Unless *blank=False* is set on the field along with a *default* then a label containing `"---------"` will be rendered with the select box. To override this behavior, add a tuple to `choices` containing `None`; e.g. `(None, 'Your String For Display')`. Alternatively, you can use an empty string instead of `None` where this makes sense - such as on a *CharField*.

db_column

`Field.`**db_column**

The name of the database column to use for this field. If this isn't given, Django will use the field's name.

If your database column name is an SQL reserved word, or contains characters that aren't allowed in Python variable names – notably, the hyphen – that's OK. Django quotes column and table names behind the scenes.

db_index

`Field.`**db_index**

If `True`, a database index will be created for this field.

db_tablespace

`Field.`**db_tablespace**

The name of the database tablespace to use for this field's index, if this field is indexed. The default is the project's *DEFAULT_INDEX_TABLESPACE* setting, if set, or the *db_tablespace* of the model, if any. If the backend doesn't support tablespaces for indexes, this option is ignored.

default

`Field.`**default**

The default value for the field. This can be a value or a callable object. If callable it will be called every time a new object is created.

The default can't be a mutable object (model instance, `list`, `set`, etc.), as a reference to the same instance of that object would be used as the default value in all new model instances. Instead, wrap the desired default in a callable. For example, if you want to specify a default `dict` for *JSONField*, use a function:

```
def contact_default():
    return {"email": "to1@example.com"}

contact_info = JSONField("ContactInfo", default=contact_default)
```

`lambda`s can't be used for field options like `default` because they can't be *serialized by migrations*. See that documentation for other caveats.

For fields like *ForeignKey* that map to model instances, defaults should be the value of the field they reference (pk unless *to_field* is set) instead of model instances.

The default value is used when new model instances are created and a value isn't provided for the field. When the field is a primary key, the default is also used when the field is set to None.

editable

Field.**editable**

If False, the field will not be displayed in the admin or any other *ModelForm*. They are also skipped during *model validation*. Default is True.

error_messages

Field.**error_messages**

The error_messages argument lets you override the default messages that the field will raise. Pass in a dictionary with keys matching the error messages you want to override.

Error message keys include null, blank, invalid, invalid_choice, unique, and unique_for_date. Additional error message keys are specified for each field in the *Field types* section below.

help_text

Field.**help_text**

Extra "help" text to be displayed with the form widget. It's useful for documentation even if your field isn't used on a form.

Note that this value is *not* HTML-escaped in automatically-generated forms. This lets you include HTML in *help_text* if you so desire. For example:

```
help_text="Please use the following format: <em>YYYY-MM-DD</em>."
```

Alternatively you can use plain text and django.utils.html.escape() to escape any HTML special characters. Ensure that you escape any help text that may come from untrusted users to avoid a cross-site scripting attack.

primary_key

Field.**primary_key**

If True, this field is the primary key for the model.

If you don't specify primary_key=True for any field in your model, Django will automatically add an *AutoField* to hold the primary key, so you don't need to set primary_key=True on any of your fields unless you want to override the default primary-key behavior. For more, see *Automatic primary key fields*.

primary_key=True implies *null=False* and *unique=True*. Only one primary key is allowed on an object.

The primary key field is read-only. If you change the value of the primary key on an existing object and then save it, a new object will be created alongside the old one.

unique

Field.**unique**

If `True`, this field must be unique throughout the table.

This is enforced at the database level and by model validation. If you try to save a model with a duplicate value in a *unique* field, a *django.db.IntegrityError* will be raised by the model's *save()* method.

This option is valid on all field types except *ManyToManyField*, *OneToOneField*, and *FileField*.

Note that when `unique` is `True`, you don't need to specify *db_index*, because `unique` implies the creation of an index.

unique_for_date

Field.**unique_for_date**

Set this to the name of a *DateField* or *DateTimeField* to require that this field be unique for the value of the date field.

For example, if you have a field `title` that has `unique_for_date="pub_date"`, then Django wouldn't allow the entry of two records with the same `title` and `pub_date`.

Note that if you set this to point to a *DateTimeField*, only the date portion of the field will be considered. Besides, when *USE_TZ* is `True`, the check will be performed in the *current time zone* at the time the object gets saved.

This is enforced by *Model.validate_unique()* during model validation but not at the database level. If any *unique_for_date* constraint involves fields that are not part of a *ModelForm* (for example, if one of the fields is listed in `exclude` or has *editable=False*), *Model.validate_unique()* will skip validation for that particular constraint.

unique_for_month

Field.**unique_for_month**

Like *unique_for_date*, but requires the field to be unique with respect to the month.

unique_for_year

Field.**unique_for_year**

Like *unique_for_date* and *unique_for_month*.

verbose_name

Field.**verbose_name**

A human-readable name for the field. If the verbose name isn't given, Django will automatically create it using the field's attribute name, converting underscores to spaces. See *Verbose field names*.

validators

Field.**validators**

A list of validators to run for this field. See the validators documentation for more information.

Registering and fetching lookups `Field` implements the *lookup registration API*. The API can be used to customize which lookups are available for a field class, and how lookups are fetched from a field.

Field types

AutoField

class AutoField(***options*)

An *IntegerField* that automatically increments according to available IDs. You usually won't need to use this directly; a primary key field will automatically be added to your model if you don't specify otherwise. See *Automatic primary key fields*.

BigAutoField

class BigAutoField(***options*)

A 64-bit integer, much like an *AutoField* except that it is guaranteed to fit numbers from `1` to `9223372036854775807`.

BigIntegerField

class BigIntegerField(***options*)

A 64-bit integer, much like an *IntegerField* except that it is guaranteed to fit numbers from `-9223372036854775808` to `9223372036854775807`. The default form widget for this field is a *TextInput*.

BinaryField

class BinaryField(***options*)

A field to store raw binary data. It only supports `bytes` assignment. Be aware that this field has limited functionality. For example, it is not possible to filter a queryset on a `BinaryField` value. It is also not possible to include a `BinaryField` in a *ModelForm*.

Abusing BinaryField

Although you might think about storing files in the database, consider that it is bad design in 99% of the cases. This field is *not* a replacement for proper static files handling.

BooleanField

class BooleanField(***options*)

A true/false field.

The default form widget for this field is a *CheckboxInput*.

If you need to accept *null* values then use *NullBooleanField* instead.

The default value of BooleanField is None when *Field.default* isn't defined.

CharField

class CharField(*max_length=None*, ***options*)

A string field, for small- to large-sized strings.

For large amounts of text, use *TextField*.

The default form widget for this field is a *TextInput*.

CharField has one extra required argument:

CharField.**max_length**
: The maximum length (in characters) of the field. The max_length is enforced at the database level and in Django's validation.

: If you are writing an application that must be portable to multiple database backends, you should be aware that there are restrictions on max_length for some backends. Refer to the database backend notes for details.

MySQL users

If you are using this field with MySQLdb 1.2.2 and the utf8_bin collation (which is *not* the default), there are some issues to be aware of. Refer to the *MySQL database notes* for details.

CommaSeparatedIntegerField

class CommaSeparatedIntegerField(*max_length=None*, ***options*)

1.9 : This field is deprecated in favor of *CharField* with validators=[*validate_comma_separated_integer_list*].

A field of integers separated by commas. As in *CharField*, the *max_length* argument is required and the note about database portability mentioned there should be heeded.

DateField

class DateField(*auto_now=False*, *auto_now_add=False*, ***options*)

A date, represented in Python by a datetime.date instance. Has a few extra, optional arguments:

DateField.**auto_now**
: Automatically set the field to now every time the object is saved. Useful for "last-modified" timestamps. Note that the current date is *always* used; it's not just a default value that you can override.

The field is only automatically updated when calling *Model.save()*. The field isn't updated when making updates to other fields in other ways such as *QuerySet.update()*, though you can specify a custom value for the field in an update like that.

DateField.**auto_now_add**
Automatically set the field to now when the object is first created. Useful for creation of timestamps. Note that the current date is *always* used; it's not just a default value that you can override. So even if you set a value for this field when creating the object, it will be ignored. If you want to be able to modify this field, set the following instead of auto_now_add=True:

- For *DateField*: default=date.today - from datetime.date.today()
- For *DateTimeField*: default=timezone.now - from *django.utils.timezone.now()*

The default form widget for this field is a *TextInput*. The admin adds a JavaScript calendar, and a shortcut for "Today". Includes an additional invalid_date error message key.

The options auto_now_add, auto_now, and default are mutually exclusive. Any combination of these options will result in an error.

: As currently implemented, setting auto_now or auto_now_add to True will cause the field to have editable=False and blank=True set.

: The auto_now and auto_now_add options will always use the date in the *default timezone* at the moment of creation or update. If you need something different, you may want to consider simply using your own callable default or overriding save() instead of using auto_now or auto_now_add; or using a DateTimeField instead of a DateField and deciding how to handle the conversion from datetime to date at display time.

DateTimeField

class DateTimeField(*auto_now=False*, *auto_now_add=False*, ***options*)

A date and time, represented in Python by a datetime.datetime instance. Takes the same extra arguments as *DateField*.

The default form widget for this field is a single *TextInput*. The admin uses two separate *TextInput* widgets with JavaScript shortcuts.

DecimalField

class DecimalField(*max_digits=None*, *decimal_places=None*, ***options*)

A fixed-precision decimal number, represented in Python by a Decimal instance. Has two **required** arguments:

DecimalField.**max_digits**
The maximum number of digits allowed in the number. Note that this number must be greater than or equal to decimal_places.

DecimalField.**decimal_places**
The number of decimal places to store with the number.

For example, to store numbers up to 999 with a resolution of 2 decimal places, you'd use:

```
models.DecimalField(..., max_digits=5, decimal_places=2)
```

And to store numbers up to approximately one billion with a resolution of 10 decimal places:

```
models.DecimalField(..., max_digits=19, decimal_places=10)
```

The default form widget for this field is a *NumberInput* when *localize* is `False` or *TextInput* otherwise.

: For more information about the differences between the *FloatField* and *DecimalField* classes, please see *FloatField vs. DecimalField*.

DurationField

class DurationField(***options*)

A field for storing periods of time - modeled in Python by `timedelta`. When used on PostgreSQL, the data type used is an `interval` and on Oracle the data type is `INTERVAL DAY(9) TO SECOND(6)`. Otherwise a `bigint` of microseconds is used.

: Arithmetic with `DurationField` works in most cases. However on all databases other than PostgreSQL, comparing the value of a `DurationField` to arithmetic on `DateTimeField` instances will not work as expected.

EmailField

class EmailField(*max_length=254*, ***options*)

A *CharField* that checks that the value is a valid email address. It uses *EmailValidator* to validate the input.

FileField

class FileField(*upload_to=None*, *max_length=100*, ***options*)

A file-upload field.

: The `primary_key` and `unique` arguments are not supported, and will raise a `TypeError` if used.

Has two optional arguments:

`FileField.`**upload_to**

This attribute provides a way of setting the upload directory and file name, and can be set in two ways. In both cases, the value is passed to the *Storage.save()* method.

If you specify a string value, it may contain `strftime()` formatting, which will be replaced by the date/time of the file upload (so that uploaded files don't fill up the given directory). For example:

```
class MyModel(models.Model):
    # file will be uploaded to MEDIA_ROOT/uploads
    upload = models.FileField(upload_to='uploads/')
    # or...
    # file will be saved to MEDIA_ROOT/uploads/2015/01/30
    upload = models.FileField(upload_to='uploads/%Y/%m/%d/')
```

If you are using the default *FileSystemStorage*, the string value will be appended to your *MEDIA_ROOT* path to form the location on the local filesystem where uploaded files will be stored. If you are using a different storage, check that storage's documentation to see how it handles `upload_to`.

`upload_to` may also be a callable, such as a function. This will be called to obtain the upload path, including the filename. This callable must accept two arguments and return a Unix-style path (with forward slashes) to be passed along to the storage system. The two arguments are:

Argument	Description
`instance`	An instance of the model where the `FileField` is defined. More specifically, this is the particular instance where the current file is being attached. In most cases, this object will not have been saved to the database yet, so if it uses the default `AutoField`, *it might not yet have a value for its primary key field*.
`filename`	The filename that was originally given to the file. This may or may not be taken into account when determining the final destination path.

For example:

```
def user_directory_path(instance, filename):
    # file will be uploaded to MEDIA_ROOT/user_<id>/<filename>
    return 'user_{0}/{1}'.format(instance.user.id, filename)

class MyModel(models.Model):
    upload = models.FileField(upload_to=user_directory_path)
```

`FileField.`**`storage`**

A storage object, which handles the storage and retrieval of your files. See Managing files for details on how to provide this object.

The default form widget for this field is a *ClearableFileInput*.

Using a *FileField* or an *ImageField* (see below) in a model takes a few steps:

1. In your settings file, you'll need to define *MEDIA_ROOT* as the full path to a directory where you'd like Django to store uploaded files. (For performance, these files are not stored in the database.) Define *MEDIA_URL* as the base public URL of that directory. Make sure that this directory is writable by the Web server's user account.
2. Add the *FileField* or *ImageField* to your model, defining the *upload_to* option to specify a subdirectory of *MEDIA_ROOT* to use for uploaded files.
3. All that will be stored in your database is a path to the file (relative to *MEDIA_ROOT*). You'll most likely want to use the convenience *url* attribute provided by Django. For example, if your *ImageField* is called `mug_shot`, you can get the absolute path to your image in a template with `{{ object.mug_shot.url }}`.

For example, say your *MEDIA_ROOT* is set to `'/home/media'`, and *upload_to* is set to `'photos/%Y/%m/%d'`. The `'%Y/%m/%d'` part of *upload_to* is `strftime()` formatting; `'%Y'` is the four-digit year, `'%m'` is the two-digit month and `'%d'` is the two-digit day. If you upload a file on Jan. 15, 2007, it will be saved in the directory `/home/media/photos/2007/01/15`.

If you wanted to retrieve the uploaded file's on-disk filename, or the file's size, you could use the *name* and *size* attributes respectively; for more information on the available attributes and methods, see the *File* class reference and the Managing files topic guide.

: The file is saved as part of saving the model in the database, so the actual file name used on disk cannot be relied on until after the model has been saved.

The uploaded file's relative URL can be obtained using the *url* attribute. Internally, this calls the *url()* method of the underlying *Storage* class. Note that whenever you deal with uploaded files, you should pay close attention to where you're uploading them and what type of files they are, to avoid security holes. *Validate all uploaded files* so that you're sure the files are what you think they are. For example, if you blindly let somebody upload files, without validation, to a directory that's within your Web server's document root, then somebody could upload a CGI or PHP script and execute that script by visiting its URL on your site. Don't allow that.

Also note that even an uploaded HTML file, since it can be executed by the browser (though not by the server), can pose security threats that are equivalent to XSS or CSRF attacks.

FileField instances are created in your database as `varchar` columns with a default max length of 100 characters. As with other fields, you can change the maximum length using the *max_length* argument.

`FileField` and `FieldFile`

class `FieldFile`

When you access a *FileField* on a model, you are given an instance of *FieldFile* as a proxy for accessing the underlying file.

The API of *FieldFile* mirrors that of *File*, with one key difference: *The object wrapped by the class is not necessarily a wrapper around Python's built-in file object.* Instead, it is a wrapper around the result of the *Storage.open()* method, which may be a *File* object, or it may be a custom storage's implementation of the *File* API.

In addition to the API inherited from *File* such as *read()* and *write()*, *FieldFile* includes several methods that can be used to interact with the underlying file:

: Two methods of this class, *save()* and *delete()*, default to saving the model object of the associated `FieldFile` in the database.

`FieldFile.`**`name`**

The name of the file including the relative path from the root of the *Storage* of the associated *FileField*.

`FieldFile.`**`size`**

The result of the underlying *Storage.size()* method.

`FieldFile.`**`url`**

A read-only property to access the file's relative URL by calling the *url()* method of the underlying *Storage* class.

`FieldFile.`**`open`**(*mode='rb'*)

Opens or reopens the file associated with this instance in the specified `mode`. Unlike the standard Python `open()` method, it doesn't return a file descriptor.

Since the underlying file is opened implicitly when accessing it, it may be unnecessary to call this method except to reset the pointer to the underlying file or to change the `mode`.

`FieldFile.`**`close`**`()`

Behaves like the standard Python `file.close()` method and closes the file associated with this instance.

`FieldFile.`**`save`**(*name*, *content*, *save=True*)

This method takes a filename and file contents and passes them to the storage class for the field, then associates the stored file with the model field. If you want to manually associate file data with *FileField* instances on your model, the `save()` method is used to persist that file data.

Takes two required arguments: `name` which is the name of the file, and `content` which is an object containing the file's contents. The optional `save` argument controls whether or not the model instance is saved after the file associated with this field has been altered. Defaults to `True`.

Note that the `content` argument should be an instance of *django.core.files.File*, not Python's built-in file object. You can construct a *File* from an existing Python file object like this:

```
from django.core.files import File
# Open an existing file using Python's built-in open()
f = open('/path/to/hello.world')
myfile = File(f)
```

Or you can construct one from a Python string like this:

```
from django.core.files.base import ContentFile
myfile = ContentFile("hello world")
```

For more information, see Managing files.

`FieldFile.`**`delete`**(*save=True*)

Deletes the file associated with this instance and clears all attributes on the field. Note: This method will close the file if it happens to be open when `delete()` is called.

The optional `save` argument controls whether or not the model instance is saved after the file associated with this field has been deleted. Defaults to `True`.

Note that when a model is deleted, related files are not deleted. If you need to cleanup orphaned files, you'll need to handle it yourself (for instance, with a custom management command that can be run manually or scheduled to run periodically via e.g. cron).

FilePathField

class `FilePathField`(*path=None*, *match=None*, *recursive=False*, *max_length=100*, ***options*)

A *CharField* whose choices are limited to the filenames in a certain directory on the filesystem. Has three special arguments, of which the first is **required**:

`FilePathField.`**`path`**
: Required. The absolute filesystem path to a directory from which this *FilePathField* should get its choices. Example: `"/home/images"`.

`FilePathField.`**`match`**
: Optional. A regular expression, as a string, that *FilePathField* will use to filter filenames. Note that the regex will be applied to the base filename, not the full path. Example: `"foo.*\.txt$"`, which will match a file called `foo23.txt` but not `bar.txt` or `foo23.png`.

`FilePathField.`**`recursive`**
: Optional. Either `True` or `False`. Default is `False`. Specifies whether all subdirectories of *path* should be included

`FilePathField.`**`allow_files`**
: Optional. Either `True` or `False`. Default is `True`. Specifies whether files in the specified location should be included. Either this or *allow_folders* must be `True`.

`FilePathField.`**`allow_folders`**
: Optional. Either `True` or `False`. Default is `False`. Specifies whether folders in the specified location should be included. Either this or *`allow_files`* must be `True`.

Of course, these arguments can be used together.

The one potential gotcha is that *`match`* applies to the base filename, not the full path. So, this example:

```
FilePathField(path="/home/images", match="foo.*", recursive=True)
```

...will match `/home/images/foo.png` but not `/home/images/foo/bar.png` because the *`match`* applies to the base filename (`foo.png` and `bar.png`).

`FilePathField` instances are created in your database as `varchar` columns with a default max length of 100 characters. As with other fields, you can change the maximum length using the *`max_length`* argument.

FloatField

class `FloatField`(***options*)

A floating-point number represented in Python by a `float` instance.

The default form widget for this field is a *`NumberInput`* when *`localize`* is `False` or *`TextInput`* otherwise.

`FloatField` vs. `DecimalField`

The *`FloatField`* class is sometimes mixed up with the *`DecimalField`* class. Although they both represent real numbers, they represent those numbers differently. `FloatField` uses Python's `float` type internally, while `DecimalField` uses Python's `Decimal` type. For information on the difference between the two, see Python's documentation for the `decimal` module.

ImageField

class `ImageField`(*upload_to=None*, *height_field=None*, *width_field=None*, *max_length=100*, ***options*)

Inherits all attributes and methods from *`FileField`*, but also validates that the uploaded object is a valid image.

In addition to the special attributes that are available for *`FileField`*, an *`ImageField`* also has `height` and `width` attributes.

To facilitate querying on those attributes, *`ImageField`* has two extra optional arguments:

`ImageField.`**`height_field`**
: Name of a model field which will be auto-populated with the height of the image each time the model instance is saved.

`ImageField.`**`width_field`**
: Name of a model field which will be auto-populated with the width of the image each time the model instance is saved.

Requires the Pillow library.

`ImageField` instances are created in your database as `varchar` columns with a default max length of 100 characters. As with other fields, you can change the maximum length using the *`max_length`* argument.

The default form widget for this field is a *`ClearableFileInput`*.

IntegerField

class IntegerField(***options*)

An integer. Values from `-2147483648` to `2147483647` are safe in all databases supported by Django. The default form widget for this field is a *NumberInput* when *localize* is `False` or *TextInput* otherwise.

GenericIPAddressField

class GenericIPAddressField(*protocol='both'*, *unpack_ipv4=False*, ***options*)

An IPv4 or IPv6 address, in string format (e.g. `192.0.2.30` or `2a02:42fe::4`). The default form widget for this field is a *TextInput*.

The IPv6 address normalization follows **RFC 4291#section-2.2** section 2.2, including using the IPv4 format suggested in paragraph 3 of that section, like `::ffff:192.0.2.0`. For example, `2001:0::0:01` would be normalized to `2001::1`, and `::ffff:0a0a:0a0a` to `::ffff:10.10.10.10`. All characters are converted to lowercase.

GenericIPAddressField.**protocol**
: Limits valid inputs to the specified protocol. Accepted values are `'both'` (default), `'IPv4'` or `'IPv6'`. Matching is case insensitive.

GenericIPAddressField.**unpack_ipv4**
: Unpacks IPv4 mapped addresses like `::ffff:192.0.2.1`. If this option is enabled that address would be unpacked to `192.0.2.1`. Default is disabled. Can only be used when `protocol` is set to `'both'`.

If you allow for blank values, you have to allow for null values since blank values are stored as null.

NullBooleanField

class NullBooleanField(***options*)

Like a *BooleanField*, but allows `NULL` as one of the options. Use this instead of a *BooleanField* with `null=True`. The default form widget for this field is a *NullBooleanSelect*.

PositiveIntegerField

class PositiveIntegerField(***options*)

Like an *IntegerField*, but must be either positive or zero (`0`). Values from `0` to `2147483647` are safe in all databases supported by Django. The value `0` is accepted for backward compatibility reasons.

PositiveSmallIntegerField

class PositiveSmallIntegerField(***options*)

Like a *PositiveIntegerField*, but only allows values under a certain (database-dependent) point. Values from `0` to `32767` are safe in all databases supported by Django.

SlugField

class SlugField(*max_length=50*, ***options*)

Slug is a newspaper term. A slug is a short label for something, containing only letters, numbers, underscores or hyphens. They're generally used in URLs.

Like a CharField, you can specify *max_length* (read the note about database portability and *max_length* in that section, too). If *max_length* is not specified, Django will use a default length of 50.

Implies setting *Field.db_index* to `True`.

It is often useful to automatically prepopulate a SlugField based on the value of some other value. You can do this automatically in the admin using *prepopulated_fields*.

`SlugField.`**`allow_unicode`**
If `True`, the field accepts Unicode letters in addition to ASCII letters. Defaults to `False`.

SmallIntegerField

class `SmallIntegerField`(***options*)

Like an *IntegerField*, but only allows values under a certain (database-dependent) point. Values from `-32768` to `32767` are safe in all databases supported by Django.

TextField

class `TextField`(***options*)

A large text field. The default form widget for this field is a *Textarea*.

If you specify a `max_length` attribute, it will be reflected in the *Textarea* widget of the auto-generated form field. However it is not enforced at the model or database level. Use a *CharField* for that.

MySQL users

If you are using this field with MySQLdb 1.2.1p2 and the `utf8_bin` collation (which is *not* the default), there are some issues to be aware of. Refer to the *MySQL database notes* for details.

TimeField

class `TimeField`(*auto_now=False*, *auto_now_add=False*, ***options*)

A time, represented in Python by a `datetime.time` instance. Accepts the same auto-population options as *DateField*.

The default form widget for this field is a *TextInput*. The admin adds some JavaScript shortcuts.

URLField

class `URLField`(*max_length=200*, ***options*)

A *CharField* for a URL.

The default form widget for this field is a *TextInput*.

Like all *CharField* subclasses, *URLField* takes the optional *max_length* argument. If you don't specify *max_length*, a default of 200 is used.

UUIDField

class UUIDField(***options*)

A field for storing universally unique identifiers. Uses Python's `UUID` class. When used on PostgreSQL, this stores in a `uuid` datatype, otherwise in a `char(32)`.

Universally unique identifiers are a good alternative to *`AutoField`* for *`primary_key`*. The database will not generate the UUID for you, so it is recommended to use *`default`*:

```
import uuid
from django.db import models

class MyUUIDModel(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    # other fields
```

Note that a callable (with the parentheses omitted) is passed to `default`, not an instance of `UUID`.

Relationship fields

Django also defines a set of fields that represent relations.

ForeignKey

class ForeignKey(*othermodel*, *on_delete*, ***options*)

A many-to-one relationship. Requires a positional argument: the class to which the model is related.

`on_delete` can now be used as the second positional argument (previously it was typically only passed as a keyword argument). It will be a required argument in Django 2.0. To create a recursive relationship – an object that has a many-to-one relationship with itself – use `models.ForeignKey('self', on_delete=models.CASCADE)`. If you need to create a relationship on a model that has not yet been defined, you can use the name of the model, rather than the model object itself:

```
from django.db import models

class Car(models.Model):
    manufacturer = models.ForeignKey(
        'Manufacturer',
        on_delete=models.CASCADE,
    )
    # ...

class Manufacturer(models.Model):
    # ...
    pass
```

Relationships defined this way on *abstract models* are resolved when the model is subclassed as a concrete model and are not relative to the abstract model's `app_label`:

products/models.py

```
from django.db import models

class AbstractCar(models.Model):
    manufacturer = models.ForeignKey('Manufacturer', on_delete=models.CASCADE)
```

```
    class Meta:
        abstract = True
```

production/models.py

```
from django.db import models
from products.models import AbstractCar

class Manufacturer(models.Model):
    pass

class Car(AbstractCar):
    pass

# Car.manufacturer will point to `production.Manufacturer` here.
```

To refer to models defined in another application, you can explicitly specify a model with the full application label. For example, if the `Manufacturer` model above is defined in another application called `production`, you'd need to use:

```
class Car(models.Model):
    manufacturer = models.ForeignKey(
        'production.Manufacturer',
        on_delete=models.CASCADE,
    )
```

This sort of reference can be useful when resolving circular import dependencies between two applications.

A database index is automatically created on the `ForeignKey`. You can disable this by setting *db_index* to `False`. You may want to avoid the overhead of an index if you are creating a foreign key for consistency rather than joins, or if you will be creating an alternative index like a partial or multiple column index.

Database Representation Behind the scenes, Django appends `"_id"` to the field name to create its database column name. In the above example, the database table for the `Car` model will have a `manufacturer_id` column. (You can change this explicitly by specifying *db_column*) However, your code should never have to deal with the database column name, unless you write custom SQL. You'll always deal with the field names of your model object.

Arguments *ForeignKey* accepts other arguments that define the details of how the relation works.

`ForeignKey.`**`on_delete`**

When an object referenced by a *ForeignKey* is deleted, Django will emulate the behavior of the SQL constraint specified by the *on_delete* argument. For example, if you have a nullable *ForeignKey* and you want it to be set null when the referenced object is deleted:

```
    user = models.ForeignKey(
        User,
        models.SET_NULL,
        blank=True,
        null=True,
    )
```

1.9 : *on_delete* will become a required argument in Django 2.0. In older versions it defaults to `CASCADE`.

The possible values for *on_delete* are found in *django.db.models*:

- **`CASCADE`**

  Cascade deletes. Django emulates the behavior of the SQL constraint ON DELETE CASCADE and also deletes the object containing the ForeignKey.

- **PROTECT**
  Prevent deletion of the referenced object by raising *ProtectedError*, a subclass of *django.db.IntegrityError*.
- **SET_NULL**
  Set the *ForeignKey* null; this is only possible if *null* is True.
- **SET_DEFAULT**
  Set the *ForeignKey* to its default value; a default for the *ForeignKey* must be set.
- **SET()**
  Set the *ForeignKey* to the value passed to *SET()*, or if a callable is passed in, the result of calling it. In most cases, passing a callable will be necessary to avoid executing queries at the time your models.py is imported:

```
from django.conf import settings
from django.contrib.auth import get_user_model
from django.db import models

def get_sentinel_user():
    return get_user_model().objects.get_or_create(username='deleted')[0]

class MyModel(models.Model):
    user = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.SET(get_sentinel_user),
    )
```

- **DO_NOTHING**
  Take no action. If your database backend enforces referential integrity, this will cause an *IntegrityError* unless you manually add an SQL ON DELETE constraint to the database field.

ForeignKey.**limit_choices_to**
Sets a limit to the available choices for this field when this field is rendered using a ModelForm or the admin (by default, all objects in the queryset are available to choose). Either a dictionary, a *Q* object, or a callable returning a dictionary or *Q* object can be used.

For example:

```
staff_member = models.ForeignKey(
    User,
    on_delete=models.CASCADE,
    limit_choices_to={'is_staff': True},
)
```

causes the corresponding field on the ModelForm to list only Users that have is_staff=True. This may be helpful in the Django admin.

The callable form can be helpful, for instance, when used in conjunction with the Python datetime module to limit selections by date range. For example:

```
def limit_pub_date_choices():
    return {'pub_date__lte': datetime.date.utcnow()}

limit_choices_to = limit_pub_date_choices
```

If limit_choices_to is or returns a *Q object*, which is useful for *complex queries*, then it will only have an effect on the choices available in the admin when the field is not listed in *raw_id_fields* in the ModelAdmin for the model.

: If a callable is used for `limit_choices_to`, it will be invoked every time a new form is instantiated. It may also be invoked when a model is validated, for example by management commands or the admin. The admin constructs querysets to validate its form inputs in various edge cases multiple times, so there is a possibility your callable may be invoked several times.

`ForeignKey.`**`related_name`**

The name to use for the relation from the related object back to this one. It's also the default value for *`related_query_name`* (the name to use for the reverse filter name from the target model). See the *related objects documentation* for a full explanation and example. Note that you must set this value when defining relations on *abstract models*; and when you do so *some special syntax* is available.

If you'd prefer Django not to create a backwards relation, set `related_name` to `'+'` or end it with `'+'`. For example, this will ensure that the `User` model won't have a backwards relation to this model:

```
user = models.ForeignKey(
    User,
    on_delete=models.CASCADE,
    related_name='+',
)
```

`ForeignKey.`**`related_query_name`**

The name to use for the reverse filter name from the target model. It defaults to the value of *`related_name`* or *`default_related_name`* if set, otherwise it defaults to the name of the model:

```
# Declare the ForeignKey with related_query_name
class Tag(models.Model):
    article = models.ForeignKey(
        Article,
        on_delete=models.CASCADE,
        related_name="tags",
        related_query_name="tag",
    )
    name = models.CharField(max_length=255)

# That's now the name of the reverse filter
Article.objects.filter(tag__name="important")
```

Like *`related_name`*, `related_query_name` supports app label and class interpolation via *some special syntax*.

`ForeignKey.`**`to_field`**

The field on the related object that the relation is to. By default, Django uses the primary key of the related object.

`ForeignKey.`**`db_constraint`**

Controls whether or not a constraint should be created in the database for this foreign key. The default is `True`, and that's almost certainly what you want; setting this to `False` can be very bad for data integrity. That said, here are some scenarios where you might want to do this:

- You have legacy data that is not valid.
- You're sharding your database.

If this is set to `False`, accessing a related object that doesn't exist will raise its `DoesNotExist` exception.

`ForeignKey.`**`swappable`**

Controls the migration framework's reaction if this *`ForeignKey`* is pointing at a swappable model. If it is `True` - the default - then if the *`ForeignKey`* is pointing at a model which matches the current value of

`settings.AUTH_USER_MODEL` (or another swappable model setting) the relationship will be stored in the migration using a reference to the setting, not to the model directly.

You only want to override this to be `False` if you are sure your model should always point towards the swapped-in model - for example, if it is a profile model designed specifically for your custom user model.

Setting it to `False` does not mean you can reference a swappable model even if it is swapped out - `False` just means that the migrations made with this ForeignKey will always reference the exact model you specify (so it will fail hard if the user tries to run with a User model you don't support, for example).

If in doubt, leave it to its default of `True`.

ManyToManyField

class ManyToManyField(*othermodel*, ***options*)

A many-to-many relationship. Requires a positional argument: the class to which the model is related, which works exactly the same as it does for *ForeignKey*, including *recursive* and *lazy* relationships.

Related objects can be added, removed, or created with the field's *RelatedManager*.

Database Representation Behind the scenes, Django creates an intermediary join table to represent the many-to-many relationship. By default, this table name is generated using the name of the many-to-many field and the name of the table for the model that contains it. Since some databases don't support table names above a certain length, these table names will be automatically truncated to 64 characters and a uniqueness hash will be used. This means you might see table names like `author_books_9cdf4`; this is perfectly normal. You can manually provide the name of the join table using the *db_table* option.

Arguments *ManyToManyField* accepts an extra set of arguments – all optional – that control how the relationship functions.

`ManyToManyField.`**related_name**
Same as *ForeignKey.related_name*.

`ManyToManyField.`**related_query_name**
Same as *ForeignKey.related_query_name*.

`ManyToManyField.`**limit_choices_to**
Same as *ForeignKey.limit_choices_to*.

`limit_choices_to` has no effect when used on a `ManyToManyField` with a custom intermediate table specified using the *through* parameter.

`ManyToManyField.`**symmetrical**
Only used in the definition of ManyToManyFields on self. Consider the following model:

```
from django.db import models

class Person(models.Model):
    friends = models.ManyToManyField("self")
```

When Django processes this model, it identifies that it has a *ManyToManyField* on itself, and as a result, it doesn't add a `person_set` attribute to the `Person` class. Instead, the *ManyToManyField* is assumed to be symmetrical – that is, if I am your friend, then you are my friend.

If you do not want symmetry in many-to-many relationships with `self`, set *symmetrical* to `False`. This will force Django to add the descriptor for the reverse relationship, allowing *ManyToManyField* relationships to be non-symmetrical.

ManyToManyField.**through**

Django will automatically generate a table to manage many-to-many relationships. However, if you want to manually specify the intermediary table, you can use the *through* option to specify the Django model that represents the intermediate table that you want to use.

The most common use for this option is when you want to associate *extra data with a many-to-many relationship*.

If you don't specify an explicit `through` model, there is still an implicit `through` model class you can use to directly access the table created to hold the association. It has three fields to link the models.

If the source and target models differ, the following fields are generated:

- `id`: the primary key of the relation.
- `<containing_model>_id`: the `id` of the model that declares the `ManyToManyField`.
- `<other_model>_id`: the `id` of the model that the `ManyToManyField` points to.

If the `ManyToManyField` points from and to the same model, the following fields are generated:

- `id`: the primary key of the relation.
- `from_<model>_id`: the `id` of the instance which points at the model (i.e. the source instance).
- `to_<model>_id`: the `id` of the instance to which the relationship points (i.e. the target model instance).

This class can be used to query associated records for a given model instance like a normal model.

ManyToManyField.**through_fields**

Only used when a custom intermediary model is specified. Django will normally determine which fields of the intermediary model to use in order to establish a many-to-many relationship automatically. However, consider the following models:

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=50)

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(
        Person,
        through='Membership',
        through_fields=('group', 'person'),
    )

class Membership(models.Model):
    group = models.ForeignKey(Group, on_delete=models.CASCADE)
    person = models.ForeignKey(Person, on_delete=models.CASCADE)
    inviter = models.ForeignKey(
        Person,
        on_delete=models.CASCADE,
        related_name="membership_invites",
    )
    invite_reason = models.CharField(max_length=64)
```

`Membership` has *two* foreign keys to `Person` (`person` and `inviter`), which makes the relationship ambiguous and Django can't know which one to use. In this case, you must explicitly specify which foreign keys Django should use using `through_fields`, as in the example above.

`through_fields` accepts a 2-tuple `('field1', 'field2')`, where `field1` is the name of the foreign key to the model the *ManyToManyField* is defined on (`group` in this case), and `field2` the name of the foreign key to the target model (`person` in this case).

When you have more than one foreign key on an intermediary model to any (or even both) of the models participating in a many-to-many relationship, you *must* specify `through_fields`. This also applies to *recursive relationships* when an intermediary model is used and there are more than two foreign keys to the model, or you want to explicitly specify which two Django should use.

Recursive relationships using an intermediary model are always defined as non-symmetrical – that is, with *`symmetrical=False`* – therefore, there is the concept of a "source" and a "target". In that case `'field1'` will be treated as the "source" of the relationship and `'field2'` as the "target".

`ManyToManyField.`**`db_table`**
The name of the table to create for storing the many-to-many data. If this is not provided, Django will assume a default name based upon the names of: the table for the model defining the relationship and the name of the field itself.

`ManyToManyField.`**`db_constraint`**
Controls whether or not constraints should be created in the database for the foreign keys in the intermediary table. The default is `True`, and that's almost certainly what you want; setting this to `False` can be very bad for data integrity. That said, here are some scenarios where you might want to do this:

- You have legacy data that is not valid.
- You're sharding your database.

It is an error to pass both `db_constraint` and `through`.

`ManyToManyField.`**`swappable`**
Controls the migration framework's reaction if this *`ManyToManyField`* is pointing at a swappable model. If it is `True` - the default - then if the *`ManyToManyField`* is pointing at a model which matches the current value of `settings.AUTH_USER_MODEL` (or another swappable model setting) the relationship will be stored in the migration using a reference to the setting, not to the model directly.

You only want to override this to be `False` if you are sure your model should always point towards the swapped-in model - for example, if it is a profile model designed specifically for your custom user model.

If in doubt, leave it to its default of `True`.

`ManyToManyField` does not support *`validators`*.

`null` has no effect since there is no way to require a relationship at the database level.

OneToOneField

class `OneToOneField`(*othermodel*, *on_delete*, *parent_link=False*, ***options*)

A one-to-one relationship. Conceptually, this is similar to a *`ForeignKey`* with *`unique=True`*, but the "reverse" side of the relation will directly return a single object.

`on_delete` can now be used as the second positional argument (previously it was typically only passed as a keyword argument). It will be a required argument in Django 2.0.

This is most useful as the primary key of a model which "extends" another model in some way; *Multi-table inheritance* is implemented by adding an implicit one-to-one relation from the child model to the parent model, for example.

One positional argument is required: the class to which the model will be related. This works exactly the same as it does for *`ForeignKey`*, including all the options regarding *recursive* and *lazy* relationships.

If you do not specify the *`related_name`* argument for the `OneToOneField`, Django will use the lower-case name of the current model as default value.

With the following example:

```
from django.conf import settings
from django.db import models

class MySpecialUser(models.Model):
    user = models.OneToOneField(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
    )
    supervisor = models.OneToOneField(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
        related_name='supervisor_of',
    )
```

your resulting `User` model will have the following attributes:

```
>>> user = User.objects.get(pk=1)
>>> hasattr(user, 'myspecialuser')
True
>>> hasattr(user, 'supervisor_of')
True
```

A `DoesNotExist` exception is raised when accessing the reverse relationship if an entry in the related table doesn't exist. For example, if a user doesn't have a supervisor designated by `MySpecialUser`:

```
>>> user.supervisor_of
Traceback (most recent call last):
    ...
DoesNotExist: User matching query does not exist.
```

Additionally, `OneToOneField` accepts all of the extra arguments accepted by *`ForeignKey`*, plus one extra argument:

`OneToOneField.`**`parent_link`**

When `True` and used in a model which inherits from another *concrete model*, indicates that this field should be used as the link back to the parent class, rather than the extra `OneToOneField` which would normally be implicitly created by subclassing.

See One-to-one relationships for usage examples of `OneToOneField`.

Field API reference

class `Field`

`Field` is an abstract class that represents a database table column. Django uses fields to create the database table (*`db_type()`*), to map Python types to database (*`get_prep_value()`*) and vice-versa (*`from_db_value()`*).

A field is thus a fundamental piece in different Django APIs, notably, *`models`* and *`querysets`*.

In models, a field is instantiated as a class attribute and represents a particular table column, see Models. It has attributes such as *`null`* and *`unique`*, and methods that Django uses to map the field value to database-specific values.

A `Field` is a subclass of *`RegisterLookupMixin`* and thus both *`Transform`* and *`Lookup`* can be registered on it to be used in `QuerySets` (e.g. `field_name__exact="foo"`). All *built-in lookups* are registered by default.

All of Django's built-in fields, such as *`CharField`*, are particular implementations of `Field`. If you need a custom field, you can either subclass any of the built-in fields or write a `Field` from scratch. In either case, see

Writing custom model fields.

description
A verbose description of the field, e.g. for the *django.contrib.admindocs* application.

The description can be of the form:

```
description = _("String (up to %(max_length)s)")
```

where the arguments are interpolated from the field's `__dict__`.

To map a `Field` to a database-specific type, Django exposes several methods:

get_internal_type()
Returns a string naming this field for backend specific purposes. By default, it returns the class name.

See *Emulating built-in field types* for usage in custom fields.

db_type(*connection*)
Returns the database column data type for the *Field*, taking into account the `connection`.

See *Custom database types* for usage in custom fields.

rel_db_type(*connection*)
Returns the database column data type for fields such as `ForeignKey` and `OneToOneField` that point to the *Field*, taking into account the `connection`.

See *Custom database types* for usage in custom fields.

There are three main situations where Django needs to interact with the database backend and fields:

- when it queries the database (Python value -> database backend value)
- when it loads data from the database (database backend value -> Python value)
- when it saves to the database (Python value -> database backend value)

When querying, *get_db_prep_value()* and *get_prep_value()* are used:

get_prep_value(*value*)
`value` is the current value of the model's attribute, and the method should return data in a format that has been prepared for use as a parameter in a query.

See *Converting Python objects to query values* for usage.

get_db_prep_value(*value*, *connection*, *prepared=False*)
Converts `value` to a backend-specific value. By default it returns `value` if `prepared=True` and *get_prep_value()* if is `False`.

See *Converting query values to database values* for usage.

When loading data, *from_db_value()* is used:

from_db_value(*value*, *expression*, *connection*, *context*)
Converts a value as returned by the database to a Python object. It is the reverse of *get_prep_value()*.

This method is not used for most built-in fields as the database backend already returns the correct Python type, or the backend itself does the conversion.

See *Converting values to Python objects* for usage.

: For performance reasons, `from_db_value` is not implemented as a no-op on fields which do not require it (all Django fields). Consequently you may not call `super` in your definition.

When saving, *pre_save()* and *get_db_prep_save()* are used:

get_db_prep_save(*value*, *connection*)
Same as the *get_db_prep_value()*, but called when the field value must be *saved* to the database. By default returns *get_db_prep_value()*.

pre_save(*model_instance*, *add*)
Method called prior to *get_db_prep_save()* to prepare the value before being saved (e.g. for *DateField.auto_now*).

`model_instance` is the instance this field belongs to and `add` is whether the instance is being saved to the database for the first time.

It should return the value of the appropriate attribute from `model_instance` for this field. The attribute name is in `self.attname` (this is set up by *Field*).

See *Preprocessing values before saving* for usage.

Fields often receive their values as a different type, either from serialization or from forms.

to_python(*value*)
Converts the value into the correct Python object. It acts as the reverse of *value_to_string()*, and is also called in *clean()*.

See *Converting values to Python objects* for usage.

Besides saving to the database, the field also needs to know how to serialize its value:

value_to_string(*obj*)
Converts `obj` to a string. Used to serialize the value of the field.

See *Converting field data for serialization* for usage.

When using *model forms*, the `Field` needs to know which form field it should be represented by:

formfield(*form_class=None*, *choices_form_class=None*, ***kwargs*)
Returns the default *django.forms.Field* of this field for *ModelForm*.

By default, if both `form_class` and `choices_form_class` are `None`, it uses *CharField*. If the field has *choices* and `choices_form_class` isn't specified, it uses *TypedChoiceField*.

See *Specifying the form field for a model field* for usage.

deconstruct()
Returns a 4-tuple with enough information to recreate the field:

1. The name of the field on the model.
2. The import path of the field (e.g. `"django.db.models.IntegerField"`). This should be the most portable version, so less specific may be better.
3. A list of positional arguments.
4. A dict of keyword arguments.

This method must be added to fields prior to 1.7 to migrate its data using Migrations.

6.15.2 Field attribute reference

Every `Field` instance contains several attributes that allow introspecting its behavior. Use these attributes instead of `isinstance` checks when you need to write code that depends on a field's functionality. These attributes can be used together with the *Model._meta API* to narrow down a search for specific field types. Custom model fields should implement these flags.

Attributes for fields

`Field.`**`auto_created`**
Boolean flag that indicates if the field was automatically created, such as the `OneToOneField` used by model inheritance.

`Field.`**`concrete`**
Boolean flag that indicates if the field has a database column associated with it.

`Field.`**`hidden`**
Boolean flag that indicates if a field is used to back another non-hidden field's functionality (e.g. the `content_type` and `object_id` fields that make up a `GenericForeignKey`). The `hidden` flag is used to distinguish what constitutes the public subset of fields on the model from all the fields on the model.

: *`Options.get_fields()`* excludes hidden fields by default. Pass in `include_hidden=True` to return hidden fields in the results.

`Field.`**`is_relation`**
Boolean flag that indicates if a field contains references to one or more other models for its functionality (e.g. `ForeignKey`, `ManyToManyField`, `OneToOneField`, etc.).

`Field.`**`model`**
Returns the model on which the field is defined. If a field is defined on a superclass of a model, `model` will refer to the superclass, not the class of the instance.

Attributes for fields with relations

These attributes are used to query for the cardinality and other details of a relation. These attribute are present on all fields; however, they will only have boolean values (rather than `None`) if the field is a relation type (*`Field.is_relation=True`*).

`Field.`**`many_to_many`**
Boolean flag that is `True` if the field has a many-to-many relation; `False` otherwise. The only field included with Django where this is `True` is `ManyToManyField`.

`Field.`**`many_to_one`**
Boolean flag that is `True` if the field has a many-to-one relation, such as a `ForeignKey`; `False` otherwise.

`Field.`**`one_to_many`**
Boolean flag that is `True` if the field has a one-to-many relation, such as a `GenericRelation` or the reverse of a `ForeignKey`; `False` otherwise.

`Field.`**`one_to_one`**
Boolean flag that is `True` if the field has a one-to-one relation, such as a `OneToOneField`; `False` otherwise.

`Field.`**`related_model`**
Points to the model the field relates to. For example, `Author` in `ForeignKey(Author, on_delete=models.CASCADE)`. If a field has a generic relation (such as a `GenericForeignKey` or a `GenericRelation`) then `related_model` will be `None`.

6.15.3 Model `_meta` API

class `Options`

The model _meta API is at the core of the Django ORM. It enables other parts of the system such as lookups, queries, forms, and the admin to understand the capabilities of each model. The API is accessible through the _meta attribute of each model class, which is an instance of an django.db.models.options.Options object.

Methods that it provides can be used to:

- Retrieve all field instances of a model
- Retrieve a single field instance of a model by name

Field access API

Retrieving a single field instance of a model by name

Options.**get_field**(*field_name*)

Returns the field instance given a name of a field.

field_name can be the name of a field on the model, a field on an abstract or inherited model, or a field defined on another model that points to the model. In the latter case, the field_name will be the related_name defined by the user or the name automatically generated by Django itself.

Hidden fields cannot be retrieved by name.

If a field with the given name is not found a *FieldDoesNotExist* exception will be raised.

```
>>> from django.contrib.auth.models import User

# A field on the model
>>> User._meta.get_field('username')
<django.db.models.fields.CharField: username>

# A field from another model that has a relation with the current model
>>> User._meta.get_field('logentry')
<ManyToOneRel: admin.logentry>

# A non existent field
>>> User._meta.get_field('does_not_exist')
Traceback (most recent call last):
    ...
FieldDoesNotExist: User has no field named 'does_not_exist'
```

Retrieving all field instances of a model

Options.**get_fields**(*include_parents=True*, *include_hidden=False*)

Returns a tuple of fields associated with a model. get_fields() accepts two parameters that can be used to control which fields are returned:

include_parents True by default. Recursively includes fields defined on parent classes. If set to False, get_fields() will only search for fields declared directly on the current model. Fields from models that directly inherit from abstract models or proxy classes are considered to be local, not on the parent.

include_hidden False by default. If set to True, get_fields() will include fields that are used to back other field's functionality. This will also include any fields that have a related_name (such as *ManyToManyField*, or *ForeignKey*) that start with a "+".

```
>>> from django.contrib.auth.models import User
>>> User._meta.get_fields()
(<ManyToOneRel: admin.logentry>,
```

```
 <django.db.models.fields.AutoField: id>,
 <django.db.models.fields.CharField: password>,
 <django.db.models.fields.DateTimeField: last_login>,
 <django.db.models.fields.BooleanField: is_superuser>,
 <django.db.models.fields.CharField: username>,
 <django.db.models.fields.CharField: first_name>,
 <django.db.models.fields.CharField: last_name>,
 <django.db.models.fields.EmailField: email>,
 <django.db.models.fields.BooleanField: is_staff>,
 <django.db.models.fields.BooleanField: is_active>,
 <django.db.models.fields.DateTimeField: date_joined>,
 <django.db.models.fields.related.ManyToManyField: groups>,
 <django.db.models.fields.related.ManyToManyField: user_permissions>)

# Also include hidden fields.
>>> User._meta.get_fields(include_hidden=True)
(<ManyToOneRel: auth.user_groups>,
 <ManyToOneRel: auth.user_user_permissions>,
 <ManyToOneRel: admin.logentry>,
 <django.db.models.fields.AutoField: id>,
 <django.db.models.fields.CharField: password>,
 <django.db.models.fields.DateTimeField: last_login>,
 <django.db.models.fields.BooleanField: is_superuser>,
 <django.db.models.fields.CharField: username>,
 <django.db.models.fields.CharField: first_name>,
 <django.db.models.fields.CharField: last_name>,
 <django.db.models.fields.EmailField: email>,
 <django.db.models.fields.BooleanField: is_staff>,
 <django.db.models.fields.BooleanField: is_active>,
 <django.db.models.fields.DateTimeField: date_joined>,
 <django.db.models.fields.related.ManyToManyField: groups>,
 <django.db.models.fields.related.ManyToManyField: user_permissions>)
```

Migrating from the old API

As part of the formalization of the `Model._meta` API (from the *`django.db.models.options.Options`* class), a number of methods and properties have been deprecated and will be removed in Django 1.10.

These old APIs can be replicated by either:

- invoking *`Options.get_field()`*, or;
- invoking *`Options.get_fields()`* to retrieve a list of all fields, and then filtering this list using the *field attributes* that describe (or retrieve, in the case of `_with_model` variants) the properties of the desired fields.

Although it's possible to make strictly equivalent replacements of the old methods, that might not be the best approach. Taking the time to refactor any field loops to make better use of the new API - and possibly include fields that were previously excluded - will almost certainly result in better code.

Assuming you have a model named `MyModel`, the following substitutions can be made to convert your code to the new API:

- `MyModel._meta.get_field(name)` becomes:

```
f = MyModel._meta.get_field(name)
```

  then check if:

  - `f.auto_created == False`, because the new `get_field()` API will find "reverse" relations, and:
  - `f.is_relation and f.related_model is None`, because the new `get_field()` API will find *`GenericForeignKey`* relations.

- `MyModel._meta.get_field_by_name(name)` returns a tuple of these four values with the following replacements:
  - `field` can be found by `MyModel._meta.get_field(name)`
  - `model` can be found through the *`model`* attribute on the field.
  - `direct` can be found by: `not field.auto_created or field.concrete`

    The *`auto_created`* check excludes all "forward" and "reverse" relations that are created by Django, but this also includes `AutoField` and `OneToOneField` on proxy models. We avoid filtering out these attributes using the *`concrete`* attribute.
  - `m2m` can be found through the *`many_to_many`* attribute on the field.

- `MyModel._meta.get_fields_with_model()` becomes:

```
[
    (f, f.model if f.model != MyModel else None)
    for f in MyModel._meta.get_fields()
    if not f.is_relation
        or f.one_to_one
        or (f.many_to_one and f.related_model)
]
```

- `MyModel._meta.get_concrete_fields_with_model()` becomes:

```
[
    (f, f.model if f.model != MyModel else None)
    for f in MyModel._meta.get_fields()
    if f.concrete and (
        not f.is_relation
        or f.one_to_one
        or (f.many_to_one and f.related_model)
    )
]
```

- `MyModel._meta.get_m2m_with_model()` becomes:

```
[
    (f, f.model if f.model != MyModel else None)
    for f in MyModel._meta.get_fields()
    if f.many_to_many and not f.auto_created
]
```

- `MyModel._meta.get_all_related_objects()` becomes:

```
[
    f for f in MyModel._meta.get_fields()
    if (f.one_to_many or f.one_to_one)
    and f.auto_created and not f.concrete
]
```

- `MyModel._meta.get_all_related_objects_with_model()` becomes:

```
[
    (f, f.model if f.model != MyModel else None)
    for f in MyModel._meta.get_fields()
    if (f.one_to_many or f.one_to_one)
    and f.auto_created and not f.concrete
]
```

- `MyModel._meta.get_all_related_many_to_many_objects()` becomes:

```
[
    f for f in MyModel._meta.get_fields(include_hidden=True)
    if f.many_to_many and f.auto_created
]
```

- `MyModel._meta.get_all_related_m2m_objects_with_model()` becomes:

```
[
    (f, f.model if f.model != MyModel else None)
    for f in MyModel._meta.get_fields(include_hidden=True)
    if f.many_to_many and f.auto_created
]
```

- `MyModel._meta.get_all_field_names()` becomes:

```
from itertools import chain
list(set(chain.from_iterable(
    (field.name, field.attname) if hasattr(field, 'attname') else (field.name,)
    for field in MyModel._meta.get_fields()
    # For complete backwards compatibility, you may want to exclude
    # GenericForeignKey from the results.
    if not (field.many_to_one and field.related_model is None)
)))
```

This provides a 100% backwards compatible replacement, ensuring that both field names and attribute names `ForeignKeys` are included, but fields associated with `GenericForeignKeys` are not. A simpler version would be:

```
[f.name for f in MyModel._meta.get_fields()]
```

While this isn't 100% backwards compatible, it may be sufficient in many situations.

6.15.4 Related objects reference

class `RelatedManager`

A "related manager" is a manager used in a one-to-many or many-to-many related context. This happens in two cases:

•The "other side" of a *`ForeignKey`* relation. That is:

```
from django.db import models

class Reporter(models.Model):
    # ...
    pass

class Article(models.Model):
    reporter = models.ForeignKey(Reporter, on_delete=models.CASCADE)
```

In the above example, the methods below will be available on the manager `reporter.article_set`.

- Both sides of a *ManyToManyField* relation:

```
class Topping(models.Model):
    # ...
    pass

class Pizza(models.Model):
    toppings = models.ManyToManyField(Topping)
```

In this example, the methods below will be available both on `topping.pizza_set` and on `pizza.toppings`.

add(**objs*, *bulk=True*)

Adds the specified model objects to the related object set.

Example:

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry.objects.get(id=234)
>>> b.entry_set.add(e) # Associates Entry e with Blog b.
```

In the example above, in the case of a *ForeignKey* relationship, *QuerySet.update()* is used to perform the update. This requires the objects to already be saved.

You can use the `bulk=False` argument to instead have the related manager perform the update by calling `e.save()`.

Using `add()` with a many-to-many relationship, however, will not call any `save()` methods, but rather create the relationships using *QuerySet.bulk_create()*. If you need to execute some custom logic when a relationship is created, listen to the *m2m_changed* signal.

The `bulk` parameter was added. In older versions, foreign key updates were always done using `save()`. Use `bulk=False` if you require the old behavior.

create(***kwargs*)

Creates a new object, saves it and puts it in the related object set. Returns the newly created object:

```
>>> b = Blog.objects.get(id=1)
>>> e = b.entry_set.create(
...     headline='Hello',
...     body_text='Hi',
...     pub_date=datetime.date(2005, 1, 1)
... )

# No need to call e.save() at this point -- it's already been saved.
```

This is equivalent to (but much simpler than):

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry(
...     blog=b,
...     headline='Hello',
...     body_text='Hi',
...     pub_date=datetime.date(2005, 1, 1)
... )
>>> e.save(force_insert=True)
```

Note that there's no need to specify the keyword argument of the model that defines the relationship. In the above example, we don't pass the parameter `blog` to `create()`. Django figures out that the new `Entry` object's `blog` field should be set to `b`.

remove(**objs*)

Removes the specified model objects from the related object set:

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry.objects.get(id=234)
>>> b.entry_set.remove(e) # Disassociates Entry e from Blog b.
```

Similar to *add()*, `e.save()` is called in the example above to perform the update. Using `remove()` with a many-to-many relationship, however, will delete the relationships using *QuerySet.delete()* which means no model `save()` methods are called; listen to the *m2m_changed* signal if you wish to execute custom code when a relationship is deleted.

For *ForeignKey* objects, this method only exists if `null=True`. If the related field can't be set to `None` (`NULL`), then an object can't be removed from a relation without being added to another. In the above example, removing `e` from `b.entry_set()` is equivalent to doing `e.blog = None`, and because the `blog` *ForeignKey* doesn't have `null=True`, this is invalid.

For *ForeignKey* objects, this method accepts a `bulk` argument to control how to perform the operation. If `True` (the default), `QuerySet.update()` is used. If `bulk=False`, the `save()` method of each individual model instance is called instead. This triggers the *pre_save* and *post_save* signals and comes at the expense of performance.

clear()

Removes all objects from the related object set:

```
>>> b = Blog.objects.get(id=1)
>>> b.entry_set.clear()
```

Note this doesn't delete the related objects – it just disassociates them.

Just like `remove()`, `clear()` is only available on *ForeignKey*s where `null=True` and it also accepts the `bulk` keyword argument.

set(*objs*, *bulk=True*, *clear=False*)

Replace the set of related objects:

```
>>> new_list = [obj1, obj2, obj3]
>>> e.related_set.set(new_list)
```

This method accepts a `clear` argument to control how to perform the operation. If `False` (the default), the elements missing from the new set are removed using `remove()` and only the new ones are added. If `clear=True`, the `clear()` method is called instead and the whole set is added at once.

The `bulk` argument is passed on to *add()*.

Note that since `set()` is a compound operation, it is subject to race conditions. For instance, new objects may be added to the database in between the call to `clear()` and the call to `add()`.

: Note that `add()`, `create()`, `remove()`, `clear()`, and `set()` all apply database changes immediately for all types of related fields. In other words, there is no need to call `save()` on either end of the relationship.

Also, if you are using *an intermediate model* for a many-to-many relationship, then the `add()`, `create()`, `remove()`, and `set()` methods are disabled.

Direct Assignment

A related object set can be replaced in bulk with one operation by assigning a new iterable of objects to it:

```
>>> new_list = [obj1, obj2, obj3]
>>> e.related_set = new_list
```

If the foreign key relationship has `null=True`, then the related manager will first disassociate any existing objects in the related set before adding the contents of `new_list`. Otherwise the objects in `new_list` will be added to the existing related object set.

In earlier versions, direct assignment used to perform `clear()` followed by `add()`. It now performs a `set()` with the keyword argument `clear=False`.

1.10 : Direct assignment is deprecated in favor of the *`set()`* method:

```
>>> e.related_set.set([obj1, obj2, obj3])
```

This prevents confusion about an assignment resulting in an implicit save.

6.15.5 Model class reference

This document covers features of the *`Model`* class. For more information about models, see the complete list of Model reference guides.

Attributes

objects

`Model.`**`objects`**

Each non-abstract *`Model`* class must have a *`Manager`* instance added to it. Django ensures that in your model class you have at least a default `Manager` specified. If you don't add your own `Manager`, Django will add an attribute `objects` containing default *`Manager`* instance. If you add your own *`Manager`* instance attribute, the default one does not appear. Consider the following example:

```
from django.db import models

class Person(models.Model):
    # Add manager with another name
    people = models.Manager()
```

For more details on model managers see Managers and *Retrieving objects*.

6.15.6 Model `Meta` options

This document explains all the possible *metadata options* that you can give your model in its internal `class Meta`.

Available `Meta` options

abstract

`Options.`**`abstract`**

If `abstract = True`, this model will be an *abstract base class*.

app_label

Options.**app_label**

If a model is defined outside of an application in *INSTALLED_APPS*, it must declare which app it belongs to:

```
app_label = 'myapp'
```

If you want to represent a model with the format `app_label.object_name` or `app_label.model_name` you can use `model._meta.label` or `model._meta.label_lower` respectively.

base_manager_name

Options.**base_manager_name**

The name of the manager to use for the model's *_base_manager*.

db_table

Options.**db_table**

The name of the database table to use for the model:

```
db_table = 'music_album'
```

Table names To save you time, Django automatically derives the name of the database table from the name of your model class and the app that contains it. A model's database table name is constructed by joining the model's "app label" – the name you used in *manage.py startapp* – to the model's class name, with an underscore between them.

For example, if you have an app `bookstore` (as created by `manage.py startapp bookstore`), a model defined as `class Book` will have a database table named `bookstore_book`.

To override the database table name, use the `db_table` parameter in `class Meta`.

If your database table name is an SQL reserved word, or contains characters that aren't allowed in Python variable names – notably, the hyphen – that's OK. Django quotes column and table names behind the scenes.

Use lowercase table names for MySQL

It is strongly advised that you use lowercase table names when you override the table name via `db_table`, particularly if you are using the MySQL backend. See the *MySQL notes* for more details.

Table name quoting for Oracle

In order to meet the 30-char limitation Oracle has on table names, and match the usual conventions for Oracle databases, Django may shorten table names and turn them all-uppercase. To prevent such transformations, use a quoted name as the value for `db_table`:

```
db_table = '"name_left_in_lowercase"'
```

Such quoted names can also be used with Django's other supported database backends; except for Oracle, however, the quotes have no effect. See the *Oracle notes* for more details.

---

db_tablespace

Options.**db_tablespace**
The name of the database tablespace to use for this model. The default is the project's *DEFAULT_TABLESPACE* setting, if set. If the backend doesn't support tablespaces, this option is ignored.

default_manager_name

Options.**default_manager_name**
The name of the manager to use for the model's *_default_manager*.

default_related_name

Options.**default_related_name**
The name that will be used by default for the relation from a related object back to this one. The default is `<model_name>_set`.

This option also sets *related_query_name*.

As the reverse name for a field should be unique, be careful if you intend to subclass your model. To work around name collisions, part of the name should contain `'%(app_label)s'` and `'%(model_name)s'`, which are replaced respectively by the name of the application the model is in, and the name of the model, both lowercased. See the paragraph on *related names for abstract models*.

1.10 : This attribute now affects `related_query_name`. The old query lookup name is deprecated:

```
from django.db import models

class Foo(models.Model):
    pass

class Bar(models.Model):
    foo = models.ForeignKey(Foo)

    class Meta:
        default_related_name = 'bars'
```

```
>>> bar = Bar.objects.get(pk=1)
>>> # Using model name "bar" as lookup string is deprecated.
>>> Foo.objects.get(bar=bar)
>>> # You should use default_related_name "bars".
>>> Foo.objects.get(bars=bar)
```

get_latest_by

Options.**get_latest_by**
The name of an orderable field in the model, typically a *DateField*, *DateTimeField*, or *IntegerField*. This specifies the default field to use in your model *Manager*'s *latest()* and *earliest()* methods.

Example:

```
get_latest_by = "order_date"
```

See the *latest()* docs for more.

managed

Options.**managed**

Defaults to `True`, meaning Django will create the appropriate database tables in *migrate* or as part of migrations and remove them as part of a *flush* management command. That is, Django *manages* the database tables' lifecycles.

If `False`, no database table creation or deletion operations will be performed for this model. This is useful if the model represents an existing table or a database view that has been created by some other means. This is the *only* difference when `managed=False`. All other aspects of model handling are exactly the same as normal. This includes

1. Adding an automatic primary key field to the model if you don't declare it. To avoid confusion for later code readers, it's recommended to specify all the columns from the database table you are modeling when using unmanaged models.
2. If a model with `managed=False` contains a *ManyToManyField* that points to another unmanaged model, then the intermediate table for the many-to-many join will also not be created. However, the intermediary table between one managed and one unmanaged model *will* be created.

   If you need to change this default behavior, create the intermediary table as an explicit model (with `managed` set as needed) and use the *ManyToManyField.through* attribute to make the relation use your custom model.

For tests involving models with `managed=False`, it's up to you to ensure the correct tables are created as part of the test setup.

If you're interested in changing the Python-level behavior of a model class, you *could* use `managed=False` and create a copy of an existing model. However, there's a better approach for that situation: *Proxy models*.

order_with_respect_to

Options.**order_with_respect_to**

Makes this object orderable with respect to the given field, usually a `ForeignKey`. This can be used to make related objects orderable with respect to a parent object. For example, if an `Answer` relates to a `Question` object, and a question has more than one answer, and the order of answers matters, you'd do this:

```
from django.db import models

class Question(models.Model):
    text = models.TextField()
    # ...

class Answer(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    # ...

    class Meta:
        order_with_respect_to = 'question'
```

When `order_with_respect_to` is set, two additional methods are provided to retrieve and to set the order of the related objects: `get_RELATED_order()` and `set_RELATED_order()`, where `RELATED` is the lowercased model name. For example, assuming that a `Question` object has multiple related `Answer` objects, the list returned contains the primary keys of the related `Answer` objects:

```
>>> question = Question.objects.get(id=1)
>>> question.get_answer_order()
[1, 2, 3]
```

The order of a `Question` object's related `Answer` objects can be set by passing in a list of `Answer` primary keys:

```
>>> question.set_answer_order([3, 1, 2])
```

The related objects also get two methods, `get_next_in_order()` and `get_previous_in_order()`, which can be used to access those objects in their proper order. Assuming the `Answer` objects are ordered by `id`:

```
>>> answer = Answer.objects.get(id=2)
>>> answer.get_next_in_order()
<Answer: 3>
>>> answer.get_previous_in_order()
<Answer: 1>
```

`order_with_respect_to` implicitly sets the `ordering` option

Internally, `order_with_respect_to` adds an additional field/database column named `_order` and sets the model's *`ordering`* option to this field. Consequently, `order_with_respect_to` and `ordering` cannot be used together, and the ordering added by `order_with_respect_to` will apply whenever you obtain a list of objects of this model.

Changing `order_with_respect_to`

Because `order_with_respect_to` adds a new database column, be sure to make and apply the appropriate migrations if you add or change `order_with_respect_to` after your initial *`migrate`*.

ordering

Options.**ordering**

The default ordering for the object, for use when obtaining lists of objects:

```
ordering = ['-order_date']
```

This is a tuple or list of strings. Each string is a field name with an optional "-" prefix, which indicates descending order. Fields without a leading "-" will be ordered ascending. Use the string "?" to order randomly.

For example, to order by a `pub_date` field ascending, use this:

```
ordering = ['pub_date']
```

To order by `pub_date` descending, use this:

```
ordering = ['-pub_date']
```

To order by `pub_date` descending, then by `author` ascending, use this:

```
ordering = ['-pub_date', 'author']
```

Default ordering also affects *aggregation queries*.

: Ordering is not a free operation. Each field you add to the ordering incurs a cost to your database. Each foreign key you add will implicitly include all of its default orderings as well.
If a query doesn't have an ordering specified, results are returned from the database in an unspecified order. A particular ordering is guaranteed only when ordering by a set of fields that uniquely identify each object in the results. For example, if a `name` field isn't unique, ordering by it won't guarantee objects with the same name always appear in the same order.

permissions

`Options.`**permissions**

Extra permissions to enter into the permissions table when creating this object. Add, delete and change permissions are automatically created for each model. This example specifies an extra permission, `can_deliver_pizzas`:

```
permissions = (("can_deliver_pizzas", "Can deliver pizzas"),)
```

This is a list or tuple of 2-tuples in the format `(permission_code, human_readable_permission_name)`.

default_permissions

`Options.`**default_permissions**

Defaults to `('add', 'change', 'delete')`. You may customize this list, for example, by setting this to an empty list if your app doesn't require any of the default permissions. It must be specified on the model before the model is created by *`migrate`* in order to prevent any omitted permissions from being created.

proxy

`Options.`**proxy**

If `proxy = True`, a model which subclasses another model will be treated as a *proxy model*.

required_db_features

`Options.`**required_db_features**

List of database features that the current connection should have so that the model is considered during the migration phase. For example, if you set this list to `['gis_enabled']`, the model will only be synchronized on GIS-enabled databases. It's also useful to skip some models when testing with several database backends. Avoid relations between models that may or may not be created as the ORM doesn't handle this.

required_db_vendor

`Options.`**required_db_vendor**

Name of a supported database vendor that this model is specific to. Current built-in vendor names are: `sqlite`, `postgresql`, `mysql`, `oracle`. If this attribute is not empty and the current connection vendor doesn't match it, the model will not be synchronized.

select_on_save

Options.**select_on_save**

Determines if Django will use the pre-1.6 *django.db.models.Model.save()* algorithm. The old algorithm uses SELECT to determine if there is an existing row to be updated. The new algorithm tries an UPDATE directly. In some rare cases the UPDATE of an existing row isn't visible to Django. An example is the PostgreSQL ON UPDATE trigger which returns NULL. In such cases the new algorithm will end up doing an INSERT even when a row exists in the database.

Usually there is no need to set this attribute. The default is False.

See *django.db.models.Model.save()* for more about the old and new saving algorithm.

unique_together

Options.**unique_together**

Sets of field names that, taken together, must be unique:

```
unique_together = (("driver", "restaurant"),)
```

This is a tuple of tuples that must be unique when considered together. It's used in the Django admin and is enforced at the database level (i.e., the appropriate UNIQUE statements are included in the CREATE TABLE statement).

For convenience, unique_together can be a single tuple when dealing with a single set of fields:

```
unique_together = ("driver", "restaurant")
```

A *ManyToManyField* cannot be included in unique_together. (It's not clear what that would even mean!) If you need to validate uniqueness related to a *ManyToManyField*, try using a signal or an explicit *through* model.

The ValidationError raised during model validation when the constraint is violated has the unique_together error code.

index_together

Options.**index_together**

Sets of field names that, taken together, are indexed:

```
index_together = [
    ["pub_date", "deadline"],
]
```

This list of fields will be indexed together (i.e. the appropriate CREATE INDEX statement will be issued.)

For convenience, index_together can be a single list when dealing with a single set of fields:

```
index_together = ["pub_date", "deadline"]
```

verbose_name

Options.**verbose_name**

A human-readable name for the object, singular:

```
    verbose_name = "pizza"
```

If this isn't given, Django will use a munged version of the class name: `CamelCase` becomes `camel case`.

verbose_name_plural

Options.**verbose_name_plural**
The plural name for the object:

```
    verbose_name_plural = "stories"
```

If this isn't given, Django will use *verbose_name* + `"s"`.

Read-only `Meta` attributes

label

Options.**label**
Representation of the object, returns `app_label.object_name`, e.g. `'polls.Question'`.

label_lower

Options.**label_lower**
Representation of the model, returns `app_label.model_name`, e.g. `'polls.question'`.

6.15.7 Model instance reference

This document describes the details of the `Model` API. It builds on the material presented in the model and database query guides, so you'll probably want to read and understand those documents before reading this one.

Throughout this reference we'll use the *example Weblog models* presented in the database query guide.

Creating objects

To create a new instance of a model, just instantiate it like any other Python class:

class Model(***kwargs*)

The keyword arguments are simply the names of the fields you've defined on your model. Note that instantiating a model in no way touches your database; for that, you need to *save()*.

: You may be tempted to customize the model by overriding the `__init__` method. If you do so, however, take care not to change the calling signature as any change may prevent the model instance from being saved. Rather than overriding `__init__`, try using one of these approaches:

1. Add a classmethod on the model class:

```
    from django.db import models

    class Book(models.Model):
        title = models.CharField(max_length=100)
```

```
        @classmethod
        def create(cls, title):
            book = cls(title=title)
            # do something with the book
            return book

    book = Book.create("Pride and Prejudice")
```

2. Add a method on a custom manager (usually preferred):

```
    class BookManager(models.Manager):
        def create_book(self, title):
            book = self.create(title=title)
            # do something with the book
            return book

    class Book(models.Model):
        title = models.CharField(max_length=100)

        objects = BookManager()

    book = Book.objects.create_book("Pride and Prejudice")
```

Customizing model loading

classmethod `Model.`**`from_db`**(*db*, *field_names*, *values*)

The `from_db()` method can be used to customize model instance creation when loading from the database.

The `db` argument contains the database alias for the database the model is loaded from, `field_names` contains the names of all loaded fields, and `values` contains the loaded values for each field in `field_names`. The `field_names` are in the same order as the `values`. If all of the model's fields are present, then `values` are guaranteed to be in the order `__init__()` expects them. That is, the instance can be created by `cls(*values)`. If any fields are deferred, they won't appear in `field_names`. In that case, assign a value of `django.db.models.DEFERRED` to each of the missing fields.

In addition to creating the new model, the `from_db()` method must set the `adding` and `db` flags in the new instance's `_state` attribute.

Below is an example showing how to record the initial values of fields that are loaded from the database:

```
from django.db.models import DEFERRED

@classmethod
def from_db(cls, db, field_names, values):
    # Default implementation of from_db() (subject to change and could
    # be replaced with super()).
    if len(values) != len(cls._meta.concrete_fields):
        values = list(values)
        values.reverse()
        values = [
            values.pop() if f.attname in field_names else DEFERRED
            for f in cls._meta.concrete_fields
        ]
    new = cls(*values)
    instance._state.adding = False
    instance._state.db = db
```

```
    # customization to store the original field values on the instance
    instance._loaded_values = dict(zip(field_names, values))
    return instance

def save(self, *args, **kwargs):
    # Check how the current values differ from ._loaded_values. For example,
    # prevent changing the creator_id of the model. (This example doesn't
    # support cases where 'creator_id' is deferred).
    if not self._state.adding and (
            self.creator_id != self._loaded_values['creator_id']):
        raise ValueError("Updating the value of creator isn't allowed")
    super(...).save(*args, **kwargs)
```

The example above shows a full `from_db()` implementation to clarify how that is done. In this case it would of course be possible to just use `super()` call in the `from_db()` method.

In older versions, you could check if all fields were loaded by consulting `cls._deferred`. This attribute is removed and `django.db.models.DEFERRED` is new.

Refreshing objects from database

If you delete a field from a model instance, accessing it again reloads the value from the database:

```
>>> obj = MyModel.objects.first()
>>> del obj.field
>>> obj.field  # Loads the field from the database
```

In older versions, accessing a deleted field raised `AttributeError` instead of reloading it.

`Model.`**`refresh_from_db`**(*using=None*, *fields=None*)

If you need to reload a model's values from the database, you can use the `refresh_from_db()` method. When this method is called without arguments the following is done:

1. All non-deferred fields of the model are updated to the values currently present in the database.
2. The previously loaded related instances for which the relation's value is no longer valid are removed from the reloaded instance. For example, if you have a foreign key from the reloaded instance to another model with name `Author`, then if `obj.author_id != obj.author.id`, `obj.author` will be thrown away, and when next accessed it will be reloaded with the value of `obj.author_id`.

Only fields of the model are reloaded from the database. Other database-dependent values such as annotations aren't reloaded. Any *`@cached_property`* attributes aren't cleared either.

The reloading happens from the database the instance was loaded from, or from the default database if the instance wasn't loaded from the database. The `using` argument can be used to force the database used for reloading.

It is possible to force the set of fields to be loaded by using the `fields` argument.

For example, to test that an `update()` call resulted in the expected update, you could write a test similar to this:

```
def test_update_result(self):
    obj = MyModel.objects.create(val=1)
    MyModel.objects.filter(pk=obj.pk).update(val=F('val') + 1)
    # At this point obj.val is still 1, but the value in the database
    # was updated to 2. The object's updated value needs to be reloaded
    # from the database.
    obj.refresh_from_db()
    self.assertEqual(obj.val, 2)
```

Note that when deferred fields are accessed, the loading of the deferred field's value happens through this method. Thus it is possible to customize the way deferred loading happens. The example below shows how one can reload all of the instance's fields when a deferred field is reloaded:

```
class ExampleModel(models.Model):
    def refresh_from_db(self, using=None, fields=None, **kwargs):
        # fields contains the name of the deferred field to be
        # loaded.
        if fields is not None:
            fields = set(fields)
            deferred_fields = self.get_deferred_fields()
            # If any deferred field is going to be loaded
            if fields.intersection(deferred_fields):
                # then load all of them
                fields = fields.union(deferred_fields)
        super(ExampleModel, self).refresh_from_db(using, fields, **kwargs)
```

`Model.`**`get_deferred_fields`**`()`

A helper method that returns a set containing the attribute names of all those fields that are currently deferred for this model.

Validating objects

There are three steps involved in validating a model:

1. Validate the model fields - *`Model.clean_fields()`*
2. Validate the model as a whole - *`Model.clean()`*
3. Validate the field uniqueness - *`Model.validate_unique()`*

All three steps are performed when you call a model's *`full_clean()`* method.

When you use a *`ModelForm`*, the call to *`is_valid()`* will perform these validation steps for all the fields that are included on the form. See the ModelForm documentation for more information. You should only need to call a model's *`full_clean()`* method if you plan to handle validation errors yourself, or if you have excluded fields from the *`ModelForm`* that require validation.

`Model.`**`full_clean`**`(`*exclude=None*, *validate_unique=True*`)`

This method calls *`Model.clean_fields()`*, *`Model.clean()`*, and *`Model.validate_unique()`* (if `validate_unique` is `True`), in that order and raises a *`ValidationError`* that has a `message_dict` attribute containing errors from all three stages.

The optional `exclude` argument can be used to provide a list of field names that can be excluded from validation and cleaning. *`ModelForm`* uses this argument to exclude fields that aren't present on your form from being validated since any errors raised could not be corrected by the user.

Note that `full_clean()` will *not* be called automatically when you call your model's *`save()`* method. You'll need to call it manually when you want to run one-step model validation for your own manually created models. For example:

```
from django.core.exceptions import ValidationError
try:
    article.full_clean()
except ValidationError as e:
    # Do something based on the errors contained in e.message_dict.
    # Display them to a user, or handle them programmatically.
    pass
```

The first step `full_clean()` performs is to clean each individual field.

`Model.`**`clean_fields`**`(`*exclude=None*`)`

This method will validate all fields on your model. The optional `exclude` argument lets you provide a list of field names to exclude from validation. It will raise a *`ValidationError`* if any fields fail validation.

The second step `full_clean()` performs is to call *`Model.clean()`*. This method should be overridden to perform custom validation on your model.

`Model.`**`clean`**`()`

This method should be used to provide custom model validation, and to modify attributes on your model if desired. For instance, you could use it to automatically provide a value for a field, or to do validation that requires access to more than a single field:

```
import datetime
from django.core.exceptions import ValidationError
from django.db import models
from django.utils.translation import ugettext_lazy as _

class Article(models.Model):
    ...
    def clean(self):
        # Don't allow draft entries to have a pub_date.
        if self.status == 'draft' and self.pub_date is not None:
            raise ValidationError(_('Draft entries may not have a publication date.'))
        # Set the pub_date for published items if it hasn't been set already.
        if self.status == 'published' and self.pub_date is None:
            self.pub_date = datetime.date.today()
```

Note, however, that like *`Model.full_clean()`*, a model's `clean()` method is not invoked when you call your model's *`save()`* method.

In the above example, the *`ValidationError`* exception raised by `Model.clean()` was instantiated with a string, so it will be stored in a special error dictionary key, *`NON_FIELD_ERRORS`*. This key is used for errors that are tied to the entire model instead of to a specific field:

```
from django.core.exceptions import ValidationError, NON_FIELD_ERRORS
try:
    article.full_clean()
except ValidationError as e:
    non_field_errors = e.message_dict[NON_FIELD_ERRORS]
```

To assign exceptions to a specific field, instantiate the *`ValidationError`* with a dictionary, where the keys are the field names. We could update the previous example to assign the error to the `pub_date` field:

```
class Article(models.Model):
    ...
    def clean(self):
        # Don't allow draft entries to have a pub_date.
        if self.status == 'draft' and self.pub_date is not None:
            raise ValidationError({'pub_date': _('Draft entries may not have a publication date.')})
        ...
```

If you detect errors in multiple fields during `Model.clean()`, you can also pass a dictionary mapping field names to errors:

```
raise ValidationError({
    'title': ValidationError(_('Missing title.'), code='required'),
```

```
    'pub_date': ValidationError(_('Invalid date.'), code='invalid'),
})
```

Finally, `full_clean()` will check any unique constraints on your model.

`Model.`**`validate_unique`**(*exclude=None*)

This method is similar to *`clean_fields()`*, but validates all uniqueness constraints on your model instead of individual field values. The optional `exclude` argument allows you to provide a list of field names to exclude from validation. It will raise a *`ValidationError`* if any fields fail validation.

Note that if you provide an `exclude` argument to `validate_unique()`, any *`unique_together`* constraint involving one of the fields you provided will not be checked.

Saving objects

To save an object back to the database, call `save()`:

`Model.`**`save`**(*force_insert=False*, *force_update=False*, *using=DEFAULT_DB_ALIAS*, *update_fields=None*)

If you want customized saving behavior, you can override this `save()` method. See *Overriding predefined model methods* for more details.

The model save process also has some subtleties; see the sections below.

Auto-incrementing primary keys

If a model has an *`AutoField`* — an auto-incrementing primary key — then that auto-incremented value will be calculated and saved as an attribute on your object the first time you call `save()`:

```
>>> b2 = Blog(name='Cheddar Talk', tagline='Thoughts on cheese.')
>>> b2.id     # Returns None, because b doesn't have an ID yet.
>>> b2.save()
>>> b2.id     # Returns the ID of your new object.
```

There's no way to tell what the value of an ID will be before you call `save()`, because that value is calculated by your database, not by Django.

For convenience, each model has an *`AutoField`* named `id` by default unless you explicitly specify `primary_key=True` on a field in your model. See the documentation for *`AutoField`* for more details.

The `pk` property

`Model.`**`pk`**

Regardless of whether you define a primary key field yourself, or let Django supply one for you, each model will have a property called `pk`. It behaves like a normal attribute on the model, but is actually an alias for whichever attribute is the primary key field for the model. You can read and set this value, just as you would for any other attribute, and it will update the correct field in the model.

Explicitly specifying auto-primary-key values If a model has an *`AutoField`* but you want to define a new object's ID explicitly when saving, just define it explicitly before saving, rather than relying on the auto-assignment of the ID:

```
>>> b3 = Blog(id=3, name='Cheddar Talk', tagline='Thoughts on cheese.')
>>> b3.id     # Returns 3.
>>> b3.save()
>>> b3.id     # Returns 3.
```

If you assign auto-primary-key values manually, make sure not to use an already-existing primary-key value! If you create a new object with an explicit primary-key value that already exists in the database, Django will assume you're changing the existing record rather than creating a new one.

Given the above `'Cheddar Talk'` blog example, this example would override the previous record in the database:

```
b4 = Blog(id=3, name='Not Cheddar', tagline='Anything but cheese.')
b4.save()  # Overrides the previous blog with ID=3!
```

See *How Django knows to UPDATE vs. INSERT*, below, for the reason this happens.

Explicitly specifying auto-primary-key values is mostly useful for bulk-saving objects, when you're confident you won't have primary-key collision.

What happens when you save?

When you save an object, Django performs the following steps:

1. **Emit a pre-save signal.** The signal *django.db.models.signals.pre_save* is sent, allowing any functions listening for that signal to take some customized action.
2. **Pre-process the data.** Each field on the object is asked to perform any automated data modification that the field may need to perform.

   Most fields do *no* pre-processing — the field data is kept as-is. Pre-processing is only used on fields that have special behavior. For example, if your model has a *DateField* with `auto_now=True`, the pre-save phase will alter the data in the object to ensure that the date field contains the current date stamp. (Our documentation doesn't yet include a list of all the fields with this "special behavior.")
3. **Prepare the data for the database.** Each field is asked to provide its current value in a data type that can be written to the database.

   Most fields require *no* data preparation. Simple data types, such as integers and strings, are 'ready to write' as a Python object. However, more complex data types often require some modification.

   For example, *DateField* fields use a Python `datetime` object to store data. Databases don't store `datetime` objects, so the field value must be converted into an ISO-compliant date string for insertion into the database.
4. **Insert the data into the database.** The pre-processed, prepared data is then composed into an SQL statement for insertion into the database.
5. **Emit a post-save signal.** The signal *django.db.models.signals.post_save* is sent, allowing any functions listening for that signal to take some customized action.

How Django knows to UPDATE vs. INSERT

You may have noticed Django database objects use the same `save()` method for creating and changing objects. Django abstracts the need to use `INSERT` or `UPDATE` SQL statements. Specifically, when you call `save()`, Django follows this algorithm:

- If the object's primary key attribute is set to a value that evaluates to `True` (i.e., a value other than `None` or the empty string), Django executes an `UPDATE`.
- If the object's primary key attribute is *not* set or if the `UPDATE` didn't update anything, Django executes an `INSERT`.

The one gotcha here is that you should be careful not to specify a primary-key value explicitly when saving new objects, if you cannot guarantee the primary-key value is unused. For more on this nuance, see *Explicitly specifying auto-primary-key values* above and *Forcing an INSERT or UPDATE* below.

In Django 1.5 and earlier, Django did a `SELECT` when the primary key attribute was set. If the `SELECT` found a row, then Django did an `UPDATE`, otherwise it did an `INSERT`. The old algorithm results in one more query in the `UPDATE` case. There are some rare cases where the database doesn't report that a row was updated even if the database contains a row for the object's primary key value. An example is the PostgreSQL `ON UPDATE` trigger which returns `NULL`. In such cases it is possible to revert to the old algorithm by setting the *`select_on_save`* option to `True`.

Forcing an INSERT or UPDATE In some rare circumstances, it's necessary to be able to force the *`save()`* method to perform an SQL `INSERT` and not fall back to doing an `UPDATE`. Or vice-versa: update, if possible, but not insert a new row. In these cases you can pass the `force_insert=True` or `force_update=True` parameters to the *`save()`* method. Obviously, passing both parameters is an error: you cannot both insert *and* update at the same time!

It should be very rare that you'll need to use these parameters. Django will almost always do the right thing and trying to override that will lead to errors that are difficult to track down. This feature is for advanced use only.

Using `update_fields` will force an update similarly to `force_update`.

Updating attributes based on existing fields

Sometimes you'll need to perform a simple arithmetic task on a field, such as incrementing or decrementing the current value. The obvious way to achieve this is to do something like:

```
>>> product = Product.objects.get(name='Venezuelan Beaver Cheese')
>>> product.number_sold += 1
>>> product.save()
```

If the old `number_sold` value retrieved from the database was 10, then the value of 11 will be written back to the database.

The process can be made robust, *avoiding a race condition*, as well as slightly faster by expressing the update relative to the original field value, rather than as an explicit assignment of a new value. Django provides *`F expressions`* for performing this kind of relative update. Using *`F expressions`*, the previous example is expressed as:

```
>>> from django.db.models import F
>>> product = Product.objects.get(name='Venezuelan Beaver Cheese')
>>> product.number_sold = F('number_sold') + 1
>>> product.save()
```

For more details, see the documentation on *`F expressions`* and their *use in update queries*.

Specifying which fields to save

If `save()` is passed a list of field names in keyword argument `update_fields`, only the fields named in that list will be updated. This may be desirable if you want to update just one or a few fields on an object. There will be a slight performance benefit from preventing all of the model fields from being updated in the database. For example:

```
product.name = 'Name changed again'
product.save(update_fields=['name'])
```

The `update_fields` argument can be any iterable containing strings. An empty `update_fields` iterable will skip the save. A value of None will perform an update on all fields.

Specifying `update_fields` will force an update.

When saving a model fetched through deferred model loading (*only()* or *defer()*) only the fields loaded from the DB will get updated. In effect there is an automatic update_fields in this case. If you assign or change any deferred field value, the field will be added to the updated fields.

Deleting objects

Model.**delete**(*using=DEFAULT_DB_ALIAS*, *keep_parents=False*)

Issues an SQL DELETE for the object. This only deletes the object in the database; the Python instance will still exist and will still have data in its fields. This method returns the number of objects deleted and a dictionary with the number of deletions per object type.

For more details, including how to delete objects in bulk, see *Deleting objects*.

If you want customized deletion behavior, you can override the delete() method. See *Overriding predefined model methods* for more details.

Sometimes with *multi-table inheritance* you may want to delete only a child model's data. Specifying keep_parents=True will keep the parent model's data.

The keep_parents parameter was added.

The return value describing the number of objects deleted was added.

Pickling objects

When you pickle a model, its current state is pickled. When you unpickle it, it'll contain the model instance at the moment it was pickled, rather than the data that's currently in the database.

You can't share pickles between versions

Pickles of models are only valid for the version of Django that was used to generate them. If you generate a pickle using Django version N, there is no guarantee that pickle will be readable with Django version N+1. Pickles should not be used as part of a long-term archival strategy.

Since pickle compatibility errors can be difficult to diagnose, such as silently corrupted objects, a RuntimeWarning is raised when you try to unpickle a model in a Django version that is different than the one in which it was pickled.

Other model instance methods

A few object methods have special purposes.

__str__()

Model.**__str__**()

The __str__() method is called whenever you call str() on an object. Django uses str(obj) in a number of places. Most notably, to display an object in the Django admin site and as the value inserted into a template when it displays an object. Thus, you should always return a nice, human-readable representation of the model from the __str__() method.

For example:

```
from django.db import models
from django.utils.encoding import python_2_unicode_compatible

@python_2_unicode_compatible  # only if you need to support Python 2
class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)

    def __str__(self):
        return '%s %s' % (self.first_name, self.last_name)
```

If you'd like compatibility with Python 2, you can decorate your model class with *python_2_unicode_compatible()* as shown above.

__eq__()

Model.__eq__()

The equality method is defined such that instances with the same primary key value and the same concrete class are considered equal, except that instances with a primary key value of None aren't equal to anything except themselves. For proxy models, concrete class is defined as the model's first non-proxy parent; for all other models it's simply the model's class.

For example:

```
from django.db import models

class MyModel(models.Model):
    id = models.AutoField(primary_key=True)

class MyProxyModel(MyModel):
    class Meta:
        proxy = True

class MultitableInherited(MyModel):
    pass

# Primary keys compared
MyModel(id=1) == MyModel(id=1)
MyModel(id=1) != MyModel(id=2)
# Primay keys are None
MyModel(id=None) != MyModel(id=None)
# Same instance
instance = MyModel(id=None)
instance == instance
# Proxy model
MyModel(id=1) == MyProxyModel(id=1)
# Multi-table inheritance
MyModel(id=1) != MultitableInherited(id=1)
```

__hash__()

Model.__hash__()

The __hash__() method is based on the instance's primary key value. It is effectively hash(obj.pk). If the instance doesn't have a primary key value then a TypeError will be raised (otherwise the __hash__() method

would return different values before and after the instance is saved, but changing the `__hash__()` value of an instance is forbidden in Python.

get_absolute_url()

`Model.get_absolute_url()`

Define a `get_absolute_url()` method to tell Django how to calculate the canonical URL for an object. To callers, this method should appear to return a string that can be used to refer to the object over HTTP.

For example:

```
def get_absolute_url(self):
    return "/people/%i/" % self.id
```

While this code is correct and simple, it may not be the most portable way to to write this kind of method. The *reverse()* function is usually the best approach.

For example:

```
def get_absolute_url(self):
    from django.urls import reverse
    return reverse('people.views.details', args=[str(self.id)])
```

One place Django uses `get_absolute_url()` is in the admin app. If an object defines this method, the object-editing page will have a "View on site" link that will jump you directly to the object's public view, as given by `get_absolute_url()`.

Similarly, a couple of other bits of Django, such as the syndication feed framework, use `get_absolute_url()` when it is defined. If it makes sense for your model's instances to each have a unique URL, you should define `get_absolute_url()`.

: You should avoid building the URL from unvalidated user input, in order to reduce possibilities of link or redirect poisoning:

```
def get_absolute_url(self):
    return '/%s/' % self.name
```

If `self.name` is `'/example.com'` this returns `'//example.com/'` which, in turn, is a valid schema relative URL but not the expected `'/%2Fexample.com/'`.

It's good practice to use `get_absolute_url()` in templates, instead of hard-coding your objects' URLs. For example, this template code is bad:

```
<!-- BAD template code. Avoid! -->
<a href="/people/{{ object.id }}/">{{ object.name }}</a>
```

This template code is much better:

```
<a href="{{ object.get_absolute_url }}">{{ object.name }}</a>
```

The logic here is that if you change the URL structure of your objects, even for something simple such as correcting a spelling error, you don't want to have to track down every place that the URL might be created. Specify it once, in `get_absolute_url()` and have all your other code call that one place.

: The string you return from `get_absolute_url()` **must** contain only ASCII characters (required by the URI specification, RFC 2396) and be URL-encoded, if necessary.

Code and templates calling `get_absolute_url()` should be able to use the result directly without any further processing. You may wish to use the `django.utils.encoding.iri_to_uri()` function to help with this if you are using unicode strings containing characters outside the ASCII range at all.

Extra instance methods

In addition to *`save()`*, *`delete()`*, a model object might have some of the following methods:

`Model.`**`get_FOO_display`**`()`

For every field that has *`choices`* set, the object will have a `get_FOO_display()` method, where `FOO` is the name of the field. This method returns the "human-readable" value of the field.

For example:

```
from django.db import models

class Person(models.Model):
    SHIRT_SIZES = (
        ('S', 'Small'),
        ('M', 'Medium'),
        ('L', 'Large'),
    )
    name = models.CharField(max_length=60)
    shirt_size = models.CharField(max_length=2, choices=SHIRT_SIZES)
```

```
>>> p = Person(name="Fred Flintstone", shirt_size="L")
>>> p.save()
>>> p.shirt_size
'L'
>>> p.get_shirt_size_display()
'Large'
```

`Model.`**`get_next_by_FOO`**`(`*`**kwargs`*`)`

`Model.`**`get_previous_by_FOO`**`(`*`**kwargs`*`)`

For every *`DateField`* and *`DateTimeField`* that does not have *`null=True`*, the object will have `get_next_by_FOO()` and `get_previous_by_FOO()` methods, where `FOO` is the name of the field. This returns the next and previous object with respect to the date field, raising a *`DoesNotExist`* exception when appropriate.

Both of these methods will perform their queries using the default manager for the model. If you need to emulate filtering used by a custom manager, or want to perform one-off custom filtering, both methods also accept optional keyword arguments, which should be in the format described in *Field lookups*.

Note that in the case of identical date values, these methods will use the primary key as a tie-breaker. This guarantees that no records are skipped or duplicated. That also means you cannot use those methods on unsaved objects.

Other attributes

`DoesNotExist`

exception `Model.`**`DoesNotExist`**

This exception is raised by the ORM in a couple places, for example by *`QuerySet.get()`* when an object is not found for the given query parameters.

Django provides a `DoesNotExist` exception as an attribute of each model class to identify the class of object that could not be found and to allow you to catch a particular model class with `try/except`. The exception is a subclass of *`django.core.exceptions.ObjectDoesNotExist`*.

6.15.8 `QuerySet` API reference

This document describes the details of the `QuerySet` API. It builds on the material presented in the model and database query guides, so you'll probably want to read and understand those documents before reading this one.

Throughout this reference we'll use the *example Weblog models* presented in the database query guide.

When `QuerySet`s are evaluated

Internally, a `QuerySet` can be constructed, filtered, sliced, and generally passed around without actually hitting the database. No database activity actually occurs until you do something to evaluate the queryset.

You can evaluate a `QuerySet` in the following ways:

- **Iteration.** A `QuerySet` is iterable, and it executes its database query the first time you iterate over it. For example, this will print the headline of all entries in the database:

```
for e in Entry.objects.all():
    print(e.headline)
```

  Note: Don't use this if all you want to do is determine if at least one result exists. It's more efficient to use *`exists()`*.

- **Slicing.** As explained in *Limiting QuerySets*, a `QuerySet` can be sliced, using Python's array-slicing syntax. Slicing an unevaluated `QuerySet` usually returns another unevaluated `QuerySet`, but Django will execute the database query if you use the "step" parameter of slice syntax, and will return a list. Slicing a `QuerySet` that has been evaluated also returns a list.

  Also note that even though slicing an unevaluated `QuerySet` returns another unevaluated `QuerySet`, modifying it further (e.g., adding more filters, or modifying ordering) is not allowed, since that does not translate well into SQL and it would not have a clear meaning either.

- **Pickling/Caching.** See the following section for details of what is involved when *pickling QuerySets*. The important thing for the purposes of this section is that the results are read from the database.

- **repr().** A `QuerySet` is evaluated when you call `repr()` on it. This is for convenience in the Python interactive interpreter, so you can immediately see your results when using the API interactively.

- **len().** A `QuerySet` is evaluated when you call `len()` on it. This, as you might expect, returns the length of the result list.

  Note: If you only need to determine the number of records in the set (and don't need the actual objects), it's much more efficient to handle a count at the database level using SQL's `SELECT COUNT(*)`. Django provides a *`count()`* method for precisely this reason.

- **list().** Force evaluation of a `QuerySet` by calling `list()` on it. For example:

```
entry_list = list(Entry.objects.all())
```

- **bool().** Testing a `QuerySet` in a boolean context, such as using `bool()`, `or`, `and` or an `if` statement, will cause the query to be executed. If there is at least one result, the `QuerySet` is `True`, otherwise `False`. For example:

```
if Entry.objects.filter(headline="Test"):
   print("There is at least one Entry with the headline Test")
```

Note: If you only want to determine if at least one result exists (and don't need the actual objects), it's more efficient to use *exists()*.

Pickling QuerySets

If you pickle a QuerySet, this will force all the results to be loaded into memory prior to pickling. Pickling is usually used as a precursor to caching and when the cached queryset is reloaded, you want the results to already be present and ready for use (reading from the database can take some time, defeating the purpose of caching). This means that when you unpickle a QuerySet, it contains the results at the moment it was pickled, rather than the results that are currently in the database.

If you only want to pickle the necessary information to recreate the QuerySet from the database at a later time, pickle the query attribute of the QuerySet. You can then recreate the original QuerySet (without any results loaded) using some code like this:

```
>>> import pickle
>>> query = pickle.loads(s)      # Assuming 's' is the pickled string.
>>> qs = MyModel.objects.all()
>>> qs.query = query             # Restore the original 'query'.
```

The query attribute is an opaque object. It represents the internals of the query construction and is not part of the public API. However, it is safe (and fully supported) to pickle and unpickle the attribute's contents as described here.

You can't share pickles between versions

Pickles of QuerySets are only valid for the version of Django that was used to generate them. If you generate a pickle using Django version N, there is no guarantee that pickle will be readable with Django version N+1. Pickles should not be used as part of a long-term archival strategy.

Since pickle compatibility errors can be difficult to diagnose, such as silently corrupted objects, a RuntimeWarning is raised when you try to unpickle a queryset in a Django version that is different than the one in which it was pickled.

QuerySet API

Here's the formal declaration of a QuerySet:

class QuerySet(*model=None*, *query=None*, *using=None*)

Usually when you'll interact with a QuerySet you'll use it by *chaining filters*. To make this work, most QuerySet methods return new querysets. These methods are covered in detail later in this section.

The QuerySet class has two public attributes you can use for introspection:

ordered

True if the QuerySet is ordered — i.e. has an *order_by()* clause or a default ordering on the model. False otherwise.

db

The database that will be used if this query is executed now.

: The query parameter to *QuerySet* exists so that specialized query subclasses such as *GeoQuerySet* can reconstruct internal query state. The value of the parameter is an opaque representation of that query state and is not part of a public API. To put it simply: if you need to ask, you don't need to use it.

Methods that return new QuerySets

Django provides a range of QuerySet refinement methods that modify either the types of results returned by the QuerySet or the way its SQL query is executed.

filter()
filter(***kwargs*)
Returns a new QuerySet containing objects that match the given lookup parameters.

The lookup parameters (**kwargs) should be in the format described in *Field lookups* below. Multiple parameters are joined via AND in the underlying SQL statement.

If you need to execute more complex queries (for example, queries with OR statements), you can use *Q objects*.

exclude()
exclude(***kwargs*)
Returns a new QuerySet containing objects that do *not* match the given lookup parameters.

The lookup parameters (**kwargs) should be in the format described in *Field lookups* below. Multiple parameters are joined via AND in the underlying SQL statement, and the whole thing is enclosed in a NOT().

This example excludes all entries whose pub_date is later than 2005-1-3 AND whose headline is "Hello":

```
Entry.objects.exclude(pub_date__gt=datetime.date(2005, 1, 3), headline='Hello')
```

In SQL terms, that evaluates to:

```
SELECT ...
WHERE NOT (pub_date > '2005-1-3' AND headline = 'Hello')
```

This example excludes all entries whose pub_date is later than 2005-1-3 OR whose headline is "Hello":

```
Entry.objects.exclude(pub_date__gt=datetime.date(2005, 1, 3)).exclude(headline='Hello')
```

In SQL terms, that evaluates to:

```
SELECT ...
WHERE NOT pub_date > '2005-1-3'
AND NOT headline = 'Hello'
```

Note the second example is more restrictive.

If you need to execute more complex queries (for example, queries with OR statements), you can use *Q objects*.

annotate()
annotate(**args*, ***kwargs*)
Annotates each object in the QuerySet with the provided list of query expressions. An expression may be a simple value, a reference to a field on the model (or any related models), or an aggregate expression (averages, sums, etc.) that has been computed over the objects that are related to the objects in the QuerySet.

Each argument to annotate() is an annotation that will be added to each object in the QuerySet that is returned.

The aggregation functions that are provided by Django are described in *Aggregation Functions* below.

Annotations specified using keyword arguments will use the keyword as the alias for the annotation. Anonymous arguments will have an alias generated for them based upon the name of the aggregate function and the model field that is being aggregated. Only aggregate expressions that reference a single field can be anonymous arguments. Everything else must be a keyword argument.

For example, if you were manipulating a list of blogs, you may want to determine how many entries have been made in each blog:

```
>>> from django.db.models import Count
>>> q = Blog.objects.annotate(Count('entry'))
# The name of the first blog
>>> q[0].name
'Blogasaurus'
# The number of entries on the first blog
>>> q[0].entry__count
42
```

The `Blog` model doesn't define an `entry__count` attribute by itself, but by using a keyword argument to specify the aggregate function, you can control the name of the annotation:

```
>>> q = Blog.objects.annotate(number_of_entries=Count('entry'))
# The number of entries on the first blog, using the name provided
>>> q[0].number_of_entries
42
```

For an in-depth discussion of aggregation, see the topic guide on Aggregation.

order_by()
order_by(**fields*)
By default, results returned by a `QuerySet` are ordered by the ordering tuple given by the `ordering` option in the model's `Meta`. You can override this on a per-`QuerySet` basis by using the `order_by` method.

Example:

```
Entry.objects.filter(pub_date__year=2005).order_by('-pub_date', 'headline')
```

The result above will be ordered by `pub_date` descending, then by `headline` ascending. The negative sign in front of `"-pub_date"` indicates *descending* order. Ascending order is implied. To order randomly, use `"?"`, like so:

```
Entry.objects.order_by('?')
```

Note: `order_by('?')` queries may be expensive and slow, depending on the database backend you're using.

To order by a field in a different model, use the same syntax as when you are querying across model relations. That is, the name of the field, followed by a double underscore (__), followed by the name of the field in the new model, and so on for as many models as you want to join. For example:

```
Entry.objects.order_by('blog__name', 'headline')
```

If you try to order by a field that is a relation to another model, Django will use the default ordering on the related model, or order by the related model's primary key if there is no *`Meta.ordering`* specified. For example, since the `Blog` model has no default ordering specified:

```
Entry.objects.order_by('blog')
```

...is identical to:

```
Entry.objects.order_by('blog__id')
```

If `Blog` had `ordering = ['name']`, then the first queryset would be identical to:

```
Entry.objects.order_by('blog__name')
```

It is also possible to order a queryset by a related field, without incurring the cost of a JOIN, by referring to the `_id` of the related field:

```
# No Join
Entry.objects.order_by('blog_id')

# Join
Entry.objects.order_by('blog__id')
```

You can also order by query expressions by calling `asc()` or `desc()` on the expression:

```
Entry.objects.order_by(Coalesce('summary', 'headline').desc())
```

Be cautious when ordering by fields in related models if you are also using *`distinct()`*. See the note in *`distinct()`* for an explanation of how related model ordering can change the expected results.

: It is permissible to specify a multi-valued field to order the results by (for example, a *`ManyToManyField`* field, or the reverse relation of a *`ForeignKey`* field).

Consider this case:

```
class Event(Model):
   parent = models.ForeignKey(
        'self',
        on_delete=models.CASCADE,
        related_name='children',
   )
   date = models.DateField()

Event.objects.order_by('children__date')
```

Here, there could potentially be multiple ordering data for each `Event`; each `Event` with multiple `children` will be returned multiple times into the new `QuerySet` that `order_by()` creates. In other words, using `order_by()` on the `QuerySet` could return more items than you were working on to begin with - which is probably neither expected nor useful.

Thus, take care when using multi-valued field to order the results. **If** you can be sure that there will only be one ordering piece of data for each of the items you're ordering, this approach should not present problems. If not, make sure the results are what you expect.

There's no way to specify whether ordering should be case sensitive. With respect to case-sensitivity, Django will order results however your database backend normally orders them.

You can order by a field converted to lowercase with *`Lower`* which will achieve case-consistent ordering:

```
Entry.objects.order_by(Lower('headline').desc())
```

If you don't want any ordering to be applied to a query, not even the default ordering, call *`order_by()`* with no parameters.

You can tell if a query is ordered or not by checking the *`QuerySet.ordered`* attribute, which will be `True` if the `QuerySet` has been ordered in any way.

Each `order_by()` call will clear any previous ordering. For example, this query will be ordered by `pub_date` and not `headline`:

```
Entry.objects.order_by('headline').order_by('pub_date')
```

: Ordering is not a free operation. Each field you add to the ordering incurs a cost to your database. Each foreign key you add will implicitly include all of its default orderings as well.
If a query doesn't have an ordering specified, results are returned from the database in an unspecified order. A particular ordering is guaranteed only when ordering by a set of fields that uniquely identify each object in the results. For example, if a `name` field isn't unique, ordering by it won't guarantee objects with the same name always appear in the same order.

`reverse()`
`reverse``()`
Use the `reverse()` method to reverse the order in which a queryset's elements are returned. Calling `reverse()` a second time restores the ordering back to the normal direction.

To retrieve the "last" five items in a queryset, you could do this:

```
my_queryset.reverse()[:5]
```

Note that this is not quite the same as slicing from the end of a sequence in Python. The above example will return the last item first, then the penultimate item and so on. If we had a Python sequence and looked at `seq[-5:]`, we would see the fifth-last item first. Django doesn't support that mode of access (slicing from the end), because it's not possible to do it efficiently in SQL.

Also, note that `reverse()` should generally only be called on a `QuerySet` which has a defined ordering (e.g., when querying against a model which defines a default ordering, or when using *`order_by()`*). If no such ordering is defined for a given `QuerySet`, calling `reverse()` on it has no real effect (the ordering was undefined prior to calling `reverse()`, and will remain undefined afterward).

`distinct()`
`distinct``(`*fields*`)`
Returns a new `QuerySet` that uses `SELECT DISTINCT` in its SQL query. This eliminates duplicate rows from the query results.

By default, a `QuerySet` will not eliminate duplicate rows. In practice, this is rarely a problem, because simple queries such as `Blog.objects.all()` don't introduce the possibility of duplicate result rows. However, if your query spans multiple tables, it's possible to get duplicate results when a `QuerySet` is evaluated. That's when you'd use `distinct()`.

: Any fields used in an *`order_by()`* call are included in the SQL `SELECT` columns. This can sometimes lead to unexpected results when used in conjunction with `distinct()`. If you order by fields from a related model, those fields will be added to the selected columns and they may make otherwise duplicate rows appear to be distinct. Since the extra columns don't appear in the returned results (they are only there to support ordering), it sometimes looks like non-distinct results are being returned.

Similarly, if you use a *`values()`* query to restrict the columns selected, the columns used in any *`order_by()`* (or default model ordering) will still be involved and may affect uniqueness of the results.

The moral here is that if you are using `distinct()` be careful about ordering by related models. Similarly, when using `distinct()` and *`values()`* together, be careful when ordering by fields not in the *`values()`* call.

On PostgreSQL only, you can pass positional arguments (`*fields`) in order to specify the names of fields to which the `DISTINCT` should apply. This translates to a `SELECT DISTINCT ON` SQL query. Here's the difference. For a normal `distinct()` call, the database compares *each* field in each row when determining which rows are distinct. For a `distinct()` call with specified field names, the database will only compare the specified field names.

: When you specify field names, you *must* provide an `order_by()` in the `QuerySet`, and the fields in `order_by()` must start with the fields in `distinct()`, in the same order.

For example, `SELECT DISTINCT ON (a)` gives you the first row for each value in column `a`. If you don't specify an order, you'll get some arbitrary row.

Examples (those after the first will only work on PostgreSQL):

```
>>> Author.objects.distinct()
[...]

>>> Entry.objects.order_by('pub_date').distinct('pub_date')
[...]

>>> Entry.objects.order_by('blog').distinct('blog')
[...]

>>> Entry.objects.order_by('author', 'pub_date').distinct('author', 'pub_date')
[...]

>>> Entry.objects.order_by('blog__name', 'mod_date').distinct('blog__name', 'mod_date')
[...]

>>> Entry.objects.order_by('author', 'pub_date').distinct('author')
[...]
```

: Keep in mind that *`order_by()`* uses any default related model ordering that has been defined. You might have to explicitly order by the relation `_id` or referenced field to make sure the `DISTINCT ON` expressions match those at the beginning of the `ORDER BY` clause. For example, if the `Blog` model defined an *`ordering`* by `name`:

```
Entry.objects.order_by('blog').distinct('blog')
```

...wouldn't work because the query would be ordered by `blog__name` thus mismatching the `DISTINCT ON` expression. You'd have to explicitly order by the relation *_id* field (`blog_id` in this case) or the referenced one (`blog__pk`) to make sure both expressions match.

`values()`
`values`(**fields*)
Returns a `QuerySet` that returns dictionaries, rather than model instances, when used as an iterable.

Each of those dictionaries represents an object, with the keys corresponding to the attribute names of model objects.

This example compares the dictionaries of `values()` with the normal model objects:

```
# This list contains a Blog object.
>>> Blog.objects.filter(name__startswith='Beatles')
<QuerySet [<Blog: Beatles Blog>]>

# This list contains a dictionary.
>>> Blog.objects.filter(name__startswith='Beatles').values()
<QuerySet [{'id': 1, 'name': 'Beatles Blog', 'tagline': 'All the latest Beatles news.'}]>
```

The `values()` method takes optional positional arguments, `*fields`, which specify field names to which the `SELECT` should be limited. If you specify the fields, each dictionary will contain only the field keys/values for the

fields you specify. If you don't specify the fields, each dictionary will contain a key and value for every field in the database table.

Example:

```
>>> Blog.objects.values()
<QuerySet [{'id': 1, 'name': 'Beatles Blog', 'tagline': 'All the latest Beatles news.'}]>
>>> Blog.objects.values('id', 'name')
<QuerySet [{'id': 1, 'name': 'Beatles Blog'}]>
```

A few subtleties that are worth mentioning:

- If you have a field called `foo` that is a *`ForeignKey`*, the default `values()` call will return a dictionary key called `foo_id`, since this is the name of the hidden model attribute that stores the actual value (the `foo` attribute refers to the related model). When you are calling `values()` and passing in field names, you can pass in either `foo` or `foo_id` and you will get back the same thing (the dictionary key will match the field name you passed in).

  For example:

```
>>> Entry.objects.values()
<QuerySet [{'blog_id': 1, 'headline': 'First Entry', ...}, ...]>

>>> Entry.objects.values('blog')
<QuerySet [{'blog': 1}, ...]>

>>> Entry.objects.values('blog_id')
<QuerySet [{'blog_id': 1}, ...]>
```

- When using `values()` together with *`distinct()`*, be aware that ordering can affect the results. See the note in *`distinct()`* for details.
- If you use a `values()` clause after an *`extra()`* call, any fields defined by a `select` argument in the *`extra()`* must be explicitly included in the `values()` call. Any *`extra()`* call made after a `values()` call will have its extra selected fields ignored.
- Calling *`only()`* and *`defer()`* after `values()` doesn't make sense, so doing so will raise a `NotImplementedError`.

It is useful when you know you're only going to need values from a small number of the available fields and you won't need the functionality of a model instance object. It's more efficient to select only the fields you need to use.

Finally, note that you can call `filter()`, `order_by()`, etc. after the `values()` call, that means that these two calls are identical:

```
Blog.objects.values().order_by('id')
Blog.objects.order_by('id').values()
```

The people who made Django prefer to put all the SQL-affecting methods first, followed (optionally) by any output-affecting methods (such as `values()`), but it doesn't really matter. This is your chance to really flaunt your individualism.

You can also refer to fields on related models with reverse relations through `OneToOneField`, `ForeignKey` and `ManyToManyField` attributes:

```
>>> Blog.objects.values('name', 'entry__headline')
<QuerySet [{'name': 'My blog', 'entry__headline': 'An entry'},
     {'name': 'My blog', 'entry__headline': 'Another entry'}, ...]>
```

: Because *ManyToManyField* attributes and reverse relations can have multiple related rows, including these can have a multiplier effect on the size of your result set. This will be especially pronounced if you include multiple such fields in your `values()` query, in which case all possible combinations will be returned.

values_list()
values_list(**fields*, *flat=False*)
This is similar to `values()` except that instead of returning dictionaries, it returns tuples when iterated over. Each tuple contains the value from the respective field passed into the `values_list()` call — so the first item is the first field, etc. For example:

```
>>> Entry.objects.values_list('id', 'headline')
[(1, 'First entry'), ...]
```

If you only pass in a single field, you can also pass in the `flat` parameter. If `True`, this will mean the returned results are single values, rather than one-tuples. An example should make the difference clearer:

```
>>> Entry.objects.values_list('id').order_by('id')
[(1,), (2,), (3,), ...]

>>> Entry.objects.values_list('id', flat=True).order_by('id')
[1, 2, 3, ...]
```

It is an error to pass in `flat` when there is more than one field.

If you don't pass any values to `values_list()`, it will return all the fields in the model, in the order they were declared.

A common need is to get a specific field value of a certain model instance. To achieve that, use `values_list()` followed by a `get()` call:

```
>>> Entry.objects.values_list('headline', flat=True).get(pk=1)
'First entry'
```

`values()` and `values_list()` are both intended as optimizations for a specific use case: retrieving a subset of data without the overhead of creating a model instance. This metaphor falls apart when dealing with many-to-many and other multivalued relations (such as the one-to-many relation of a reverse foreign key) because the "one row, one object" assumption doesn't hold.

For example, notice the behavior when querying across a *ManyToManyField*:

```
>>> Author.objects.values_list('name', 'entry__headline')
[('Noam Chomsky', 'Impressions of Gaza'),
 ('George Orwell', 'Why Socialists Do Not Believe in Fun'),
 ('George Orwell', 'In Defence of English Cooking'),
 ('Don Quixote', None)]
```

Authors with multiple entries appear multiple times and authors without any entries have `None` for the entry headline.

Similarly, when querying a reverse foreign key, `None` appears for entries not having any author:

```
>>> Entry.objects.values_list('authors')
[('Noam Chomsky',), ('George Orwell',), (None,)]
```

dates()
dates(*field*, *kind*, *order='ASC'*)
Returns a `QuerySet` that evaluates to a list of `datetime.date` objects representing all available dates of a particular kind within the contents of the `QuerySet`.

`field` should be the name of a `DateField` of your model. `kind` should be either `"year"`, `"month"` or `"day"`. Each `datetime.date` object in the result list is "truncated" to the given `type`.

- `"year"` returns a list of all distinct year values for the field.
- `"month"` returns a list of all distinct year/month values for the field.
- `"day"` returns a list of all distinct year/month/day values for the field.

`order`, which defaults to `'ASC'`, should be either `'ASC'` or `'DESC'`. This specifies how to order the results.

Examples:

```
>>> Entry.objects.dates('pub_date', 'year')
[datetime.date(2005, 1, 1)]
>>> Entry.objects.dates('pub_date', 'month')
[datetime.date(2005, 2, 1), datetime.date(2005, 3, 1)]
>>> Entry.objects.dates('pub_date', 'day')
[datetime.date(2005, 2, 20), datetime.date(2005, 3, 20)]
>>> Entry.objects.dates('pub_date', 'day', order='DESC')
[datetime.date(2005, 3, 20), datetime.date(2005, 2, 20)]
>>> Entry.objects.filter(headline__contains='Lennon').dates('pub_date', 'day')
[datetime.date(2005, 3, 20)]
```

datetimes()
datetimes(*field_name*, *kind*, *order='ASC'*, *tzinfo=None*)
Returns a `QuerySet` that evaluates to a list of `datetime.datetime` objects representing all available dates of a particular kind within the contents of the `QuerySet`.

`field_name` should be the name of a `DateTimeField` of your model.

`kind` should be either `"year"`, `"month"`, `"day"`, `"hour"`, `"minute"` or `"second"`. Each `datetime.datetime` object in the result list is "truncated" to the given `type`.

`order`, which defaults to `'ASC'`, should be either `'ASC'` or `'DESC'`. This specifies how to order the results.

`tzinfo` defines the time zone to which datetimes are converted prior to truncation. Indeed, a given datetime has different representations depending on the time zone in use. This parameter must be a `datetime.tzinfo` object. If it's `None`, Django uses the *current time zone*. It has no effect when *`USE_TZ`* is `False`.

: This function performs time zone conversions directly in the database. As a consequence, your database must be able to interpret the value of `tzinfo.tzname(None)`. This translates into the following requirements:

- SQLite: install pytz — conversions are actually performed in Python.
- PostgreSQL: no requirements (see Time Zones).
- Oracle: no requirements (see Choosing a Time Zone File).
- MySQL: install pytz and load the time zone tables with mysql_tzinfo_to_sql.

---

none()
none()
Calling none() will create a queryset that never returns any objects and no query will be executed when accessing the results. A qs.none() queryset is an instance of `EmptyQuerySet`.

Examples:

```
>>> Entry.objects.none()
<QuerySet []>
>>> from django.db.models.query import EmptyQuerySet
>>> isinstance(Entry.objects.none(), EmptyQuerySet)
True
```

all()
all()
Returns a *copy* of the current QuerySet (or QuerySet subclass). This can be useful in situations where you might want to pass in either a model manager or a QuerySet and do further filtering on the result. After calling all() on either object, you'll definitely have a QuerySet to work with.

When a QuerySet is *evaluated*, it typically caches its results. If the data in the database might have changed since a QuerySet was evaluated, you can get updated results for the same query by calling all() on a previously evaluated QuerySet.

select_related()
select_related(**fields*)
Returns a QuerySet that will "follow" foreign-key relationships, selecting additional related-object data when it executes its query. This is a performance booster which results in a single more complex query but means later use of foreign-key relationships won't require database queries.

The following examples illustrate the difference between plain lookups and select_related() lookups. Here's standard lookup:

```
# Hits the database.
e = Entry.objects.get(id=5)

# Hits the database again to get the related Blog object.
b = e.blog
```

And here's select_related lookup:

```
# Hits the database.
e = Entry.objects.select_related('blog').get(id=5)

# Doesn't hit the database, because e.blog has been prepopulated
# in the previous query.
b = e.blog
```

You can use select_related() with any queryset of objects:

```
from django.utils import timezone

# Find all the blogs with entries scheduled to be published in the future.
blogs = set()

for e in Entry.objects.filter(pub_date__gt=timezone.now()).select_related('blog'):
    # Without select_related(), this would make a database query for each
    # loop iteration in order to fetch the related blog for each entry.
    blogs.add(e.blog)
```

The order of filter() and select_related() chaining isn't important. These querysets are equivalent:

```
Entry.objects.filter(pub_date__gt=timezone.now()).select_related('blog')
Entry.objects.select_related('blog').filter(pub_date__gt=timezone.now())
```

You can follow foreign keys in a similar way to querying them. If you have the following models:

```
from django.db import models

class City(models.Model):
    # ...
    pass

class Person(models.Model):
    # ...
    hometown = models.ForeignKey(
        City,
        on_delete=models.SET_NULL,
        blank=True,
        null=True,
    )

class Book(models.Model):
    # ...
    author = models.ForeignKey(Person, on_delete=models.CASCADE)
```

... then a call to `Book.objects.select_related('author__hometown').get(id=4)` will cache the related `Person` *and* the related `City`:

```
b = Book.objects.select_related('author__hometown').get(id=4)
p = b.author         # Doesn't hit the database.
c = p.hometown       # Doesn't hit the database.

b = Book.objects.get(id=4) # No select_related() in this example.
p = b.author         # Hits the database.
c = p.hometown       # Hits the database.
```

You can refer to any *ForeignKey* or *OneToOneField* relation in the list of fields passed to `select_related()`.

You can also refer to the reverse direction of a *OneToOneField* in the list of fields passed to `select_related` — that is, you can traverse a *OneToOneField* back to the object on which the field is defined. Instead of specifying the field name, use the *related_name* for the field on the related object.

There may be some situations where you wish to call `select_related()` with a lot of related objects, or where you don't know all of the relations. In these cases it is possible to call `select_related()` with no arguments. This will follow all non-null foreign keys it can find - nullable foreign keys must be specified. This is not recommended in most cases as it is likely to make the underlying query more complex, and return more data, than is actually needed.

If you need to clear the list of related fields added by past calls of `select_related` on a `QuerySet`, you can pass `None` as a parameter:

```
>>> without_relations = queryset.select_related(None)
```

Chaining `select_related` calls works in a similar way to other methods - that is that `select_related('foo', 'bar')` is equivalent to `select_related('foo').select_related('bar')`.

prefetch_related()

prefetch_related(**lookups*)

Returns a `QuerySet` that will automatically retrieve, in a single batch, related objects for each of the specified lookups.

This has a similar purpose to `select_related`, in that both are designed to stop the deluge of database queries that is caused by accessing related objects, but the strategy is quite different.

`select_related` works by creating an SQL join and including the fields of the related object in the `SELECT` statement. For this reason, `select_related` gets the related objects in the same database query. However, to avoid the much larger result set that would result from joining across a 'many' relationship, `select_related` is limited to single-valued relationships - foreign key and one-to-one.

`prefetch_related`, on the other hand, does a separate lookup for each relationship, and does the 'joining' in Python. This allows it to prefetch many-to-many and many-to-one objects, which cannot be done using `select_related`, in addition to the foreign key and one-to-one relationships that are supported by `select_related`. It also supports prefetching of *`GenericRelation`* and *`GenericForeignKey`*, however, it must be restricted to a homogeneous set of results. For example, prefetching objects referenced by a `GenericForeignKey` is only supported if the query is restricted to one `ContentType`.

For example, suppose you have these models:

```
from django.db import models

class Topping(models.Model):
    name = models.CharField(max_length=30)

class Pizza(models.Model):
    name = models.CharField(max_length=50)
    toppings = models.ManyToManyField(Topping)

    def __str__(self):              # __unicode__ on Python 2
        return "%s (%s)" % (
            self.name,
            ", ".join(topping.name for topping in self.toppings.all()),
        )
```

and run:

```
>>> Pizza.objects.all()
["Hawaiian (ham, pineapple)", "Seafood (prawns, smoked salmon)"...
```

The problem with this is that every time `Pizza.__str__()` asks for `self.toppings.all()` it has to query the database, so `Pizza.objects.all()` will run a query on the Toppings table for **every** item in the Pizza `QuerySet`.

We can reduce to just two queries using `prefetch_related`:

```
>>> Pizza.objects.all().prefetch_related('toppings')
```

This implies a `self.toppings.all()` for each `Pizza`; now each time `self.toppings.all()` is called, instead of having to go to the database for the items, it will find them in a prefetched `QuerySet` cache that was populated in a single query.

That is, all the relevant toppings will have been fetched in a single query, and used to make `QuerySets` that have a pre-filled cache of the relevant results; these `QuerySets` are then used in the `self.toppings.all()` calls.

The additional queries in `prefetch_related()` are executed after the `QuerySet` has begun to be evaluated and the primary query has been executed.

If you have an iterable of model instances, you can prefetch related attributes on those instances using the *`prefetch_related_objects()`* function.

Note that the result cache of the primary `QuerySet` and all specified related objects will then be fully loaded into memory. This changes the typical behavior of `QuerySets`, which normally try to avoid loading all objects into memory before they are needed, even after a query has been executed in the database.

: Remember that, as always with QuerySets, any subsequent chained methods which imply a different database query will ignore previously cached results, and retrieve data using a fresh database query. So, if you write the following:

```
>>> pizzas = Pizza.objects.prefetch_related('toppings')
>>> [list(pizza.toppings.filter(spicy=True)) for pizza in pizzas]
```

...then the fact that `pizza.toppings.all()` has been prefetched will not help you. The `prefetch_related('toppings')` implied `pizza.toppings.all()`, but `pizza.toppings.filter()` is a new and different query. The prefetched cache can't help here; in fact it hurts performance, since you have done a database query that you haven't used. So use this feature with caution!

You can also use the normal join syntax to do related fields of related fields. Suppose we have an additional model to the example above:

```
class Restaurant(models.Model):
    pizzas = models.ManyToManyField(Pizza, related_name='restaurants')
    best_pizza = models.ForeignKey(Pizza, related_name='championed_by')
```

The following are all legal:

```
>>> Restaurant.objects.prefetch_related('pizzas__toppings')
```

This will prefetch all pizzas belonging to restaurants, and all toppings belonging to those pizzas. This will result in a total of 3 database queries - one for the restaurants, one for the pizzas, and one for the toppings.

```
>>> Restaurant.objects.prefetch_related('best_pizza__toppings')
```

This will fetch the best pizza and all the toppings for the best pizza for each restaurant. This will be done in 3 database queries - one for the restaurants, one for the 'best pizzas', and one for one for the toppings.

Of course, the `best_pizza` relationship could also be fetched using `select_related` to reduce the query count to 2:

```
>>> Restaurant.objects.select_related('best_pizza').prefetch_related('best_pizza__toppings')
```

Since the prefetch is executed after the main query (which includes the joins needed by `select_related`), it is able to detect that the `best_pizza` objects have already been fetched, and it will skip fetching them again.

Chaining `prefetch_related` calls will accumulate the lookups that are prefetched. To clear any `prefetch_related` behavior, pass `None` as a parameter:

```
>>> non_prefetched = qs.prefetch_related(None)
```

One difference to note when using `prefetch_related` is that objects created by a query can be shared between the different objects that they are related to i.e. a single Python model instance can appear at more than one point in the tree of objects that are returned. This will normally happen with foreign key relationships. Typically this behavior will not be a problem, and will in fact save both memory and CPU time.

While `prefetch_related` supports prefetching `GenericForeignKey` relationships, the number of queries will depend on the data. Since a `GenericForeignKey` can reference data in multiple tables, one query per table referenced is needed, rather than one query for all the items. There could be additional queries on the `ContentType` table if the relevant rows have not already been fetched.

`prefetch_related` in most cases will be implemented using an SQL query that uses the 'IN' operator. This means that for a large `QuerySet` a large 'IN' clause could be generated, which, depending on the database, might have performance problems of its own when it comes to parsing or executing the SQL query. Always profile for your use case!

Note that if you use `iterator()` to run the query, `prefetch_related()` calls will be ignored since these two optimizations do not make sense together.

You can use the *`Prefetch`* object to further control the prefetch operation.

In its simplest form `Prefetch` is equivalent to the traditional string based lookups:

```
>>> Restaurant.objects.prefetch_related(Prefetch('pizzas__toppings'))
```

You can provide a custom queryset with the optional `queryset` argument. This can be used to change the default ordering of the queryset:

```
>>> Restaurant.objects.prefetch_related(
...     Prefetch('pizzas__toppings', queryset=Toppings.objects.order_by('name')))
```

Or to call *`select_related()`* when applicable to reduce the number of queries even further:

```
>>> Pizza.objects.prefetch_related(
...     Prefetch('restaurants', queryset=Restaurant.objects.select_related('best_pizza')))
```

You can also assign the prefetched result to a custom attribute with the optional `to_attr` argument. The result will be stored directly in a list.

This allows prefetching the same relation multiple times with a different `QuerySet`; for instance:

```
>>> vegetarian_pizzas = Pizza.objects.filter(vegetarian=True)
>>> Restaurant.objects.prefetch_related(
...     Prefetch('pizzas', to_attr='menu'),
...     Prefetch('pizzas', queryset=vegetarian_pizzas, to_attr='vegetarian_menu'))
```

Lookups created with custom `to_attr` can still be traversed as usual by other lookups:

```
>>> vegetarian_pizzas = Pizza.objects.filter(vegetarian=True)
>>> Restaurant.objects.prefetch_related(
...     Prefetch('pizzas', queryset=vegetarian_pizzas, to_attr='vegetarian_menu'),
...     'vegetarian_menu__toppings')
```

Using `to_attr` is recommended when filtering down the prefetch result as it is less ambiguous than storing a filtered result in the related manager's cache:

```
>>> queryset = Pizza.objects.filter(vegetarian=True)
>>>
>>> # Recommended:
>>> restaurants = Restaurant.objects.prefetch_related(
...     Prefetch('pizzas', queryset=queryset, to_attr='vegetarian_pizzas'))
>>> vegetarian_pizzas = restaurants[0].vegetarian_pizzas
>>>
>>> # Not recommended:
>>> restaurants = Restaurant.objects.prefetch_related(
...     Prefetch('pizzas', queryset=queryset))
>>> vegetarian_pizzas = restaurants[0].pizzas.all()
```

Custom prefetching also works with single related relations like forward `ForeignKey` or `OneToOneField`. Generally you'll want to use *`select_related()`* for these relations, but there are a number of cases where prefetching with a custom `QuerySet` is useful:

- You want to use a `QuerySet` that performs further prefetching on related models.
- You want to prefetch only a subset of the related objects.
- You want to use performance optimization techniques like *`deferred fields`*:

```
    >>> queryset = Pizza.objects.only('name')
    >>>
    >>> restaurants = Restaurant.objects.prefetch_related(
    ...     Prefetch('best_pizza', queryset=queryset))
```

: The ordering of lookups matters.

Take the following examples:

```
>>> prefetch_related('pizzas__toppings', 'pizzas')
```

This works even though it's unordered because `'pizzas__toppings'` already contains all the needed information, therefore the second argument `'pizzas'` is actually redundant.

```
>>> prefetch_related('pizzas__toppings', Prefetch('pizzas', queryset=Pizza.objects.all()))
```

This will raise a `ValueError` because of the attempt to redefine the queryset of a previously seen lookup. Note that an implicit queryset was created to traverse `'pizzas'` as part of the `'pizzas__toppings'` lookup.

```
>>> prefetch_related('pizza_list__toppings', Prefetch('pizzas', to_attr='pizza_list'))
```

This will trigger an `AttributeError` because `'pizza_list'` doesn't exist yet when `'pizza_list__toppings'` is being processed.

This consideration is not limited to the use of `Prefetch` objects. Some advanced techniques may require that the lookups be performed in a specific order to avoid creating extra queries; therefore it's recommended to always carefully order `prefetch_related` arguments.

extra()

extra(*select=None*, *where=None*, *params=None*, *tables=None*, *order_by=None*, *select_params=None*)

Sometimes, the Django query syntax by itself can't easily express a complex `WHERE` clause. For these edge cases, Django provides the `extra() QuerySet` modifier — a hook for injecting specific clauses into the SQL generated by a `QuerySet`.

Use this method as a last resort

This is an old API that we aim to deprecate at some point in the future. Use it only if you cannot express your query using other queryset methods. If you do need to use it, please file a ticket using the QuerySet.extra keyword with your use case (please check the list of existing tickets first) so that we can enhance the QuerySet API to allow removing `extra()`. We are no longer improving or fixing bugs for this method.

For example, this use of `extra()`:

```
>>> qs.extra(
...     select={'val': "select col from sometable where othercol = %s"},
...     select_params=(someparam,),
... )
```

is equivalent to:

```
>>> qs.annotate(val=RawSQL("select col from sometable where othercol = %s", (someparam,)))
```

The main benefit of using *RawSQL* is that you can set `output_field` if needed. The main downside is that if you refer to some table alias of the queryset in the raw SQL, then it is possible that Django might change that alias (for example, when the queryset is used as a subquery in yet another query).

: You should be very careful whenever you use `extra()`. Every time you use it, you should escape any parameters that the user can control by using `params` in order to protect against SQL injection attacks . Please read more about *SQL injection protection*.

By definition, these extra lookups may not be portable to different database engines (because you're explicitly writing SQL code) and violate the DRY principle, so you should avoid them if possible.

Specify one or more of `params`, `select`, `where` or `tables`. None of the arguments is required, but you should use at least one of them.

- `select`

  The `select` argument lets you put extra fields in the `SELECT` clause. It should be a dictionary mapping attribute names to SQL clauses to use to calculate that attribute.

  Example:

```
Entry.objects.extra(select={'is_recent': "pub_date > '2006-01-01'"})
```

  As a result, each `Entry` object will have an extra attribute, `is_recent`, a boolean representing whether the entry's `pub_date` is greater than Jan. 1, 2006.

  Django inserts the given SQL snippet directly into the `SELECT` statement, so the resulting SQL of the above example would be something like:

```
SELECT blog_entry.*, (pub_date > '2006-01-01') AS is_recent
FROM blog_entry;
```

  The next example is more advanced; it does a subquery to give each resulting `Blog` object an `entry_count` attribute, an integer count of associated `Entry` objects:

```
Blog.objects.extra(
    select={
        'entry_count': 'SELECT COUNT(*) FROM blog_entry WHERE blog_entry.blog_id = blog_blog.id'
    },
)
```

  In this particular case, we're exploiting the fact that the query will already contain the `blog_blog` table in its `FROM` clause.

  The resulting SQL of the above example would be:

```
SELECT blog_blog.*, (SELECT COUNT(*) FROM blog_entry WHERE blog_entry.blog_id = blog_blog.id) AS
FROM blog_blog;
```

  Note that the parentheses required by most database engines around subqueries are not required in Django's `select` clauses. Also note that some database backends, such as some MySQL versions, don't support subqueries.

  In some rare cases, you might wish to pass parameters to the SQL fragments in `extra(select=...)`. For this purpose, use the `select_params` parameter. Since `select_params` is a sequence and the `select` attribute is a dictionary, some care is required so that the parameters are matched up correctly with the extra select pieces. In this situation, you should use a `collections.OrderedDict` for the `select` value, not just a normal Python dictionary.

  This will work, for example:

```
Blog.objects.extra(
    select=OrderedDict([('a', '%s'), ('b', '%s')]),
    select_params=('one', 'two'))
```

If you need to use a literal `%s` inside your select string, use the sequence `%%s`.

- `where` / `tables`

  You can define explicit SQL `WHERE` clauses — perhaps to perform non-explicit joins — by using `where`. You can manually add tables to the SQL `FROM` clause by using `tables`.

  `where` and `tables` both take a list of strings. All `where` parameters are "AND"ed to any other search criteria.

  Example:

```
Entry.objects.extra(where=["foo='a' OR bar = 'a'", "baz = 'a'"])
```

  ...translates (roughly) into the following SQL:

```
SELECT * FROM blog_entry WHERE (foo='a' OR bar='a') AND (baz='a')
```

  Be careful when using the `tables` parameter if you're specifying tables that are already used in the query. When you add extra tables via the `tables` parameter, Django assumes you want that table included an extra time, if it is already included. That creates a problem, since the table name will then be given an alias. If a table appears multiple times in an SQL statement, the second and subsequent occurrences must use aliases so the database can tell them apart. If you're referring to the extra table you added in the extra `where` parameter this is going to cause errors.

  Normally you'll only be adding extra tables that don't already appear in the query. However, if the case outlined above does occur, there are a few solutions. First, see if you can get by without including the extra table and use the one already in the query. If that isn't possible, put your `extra()` call at the front of the queryset construction so that your table is the first use of that table. Finally, if all else fails, look at the query produced and rewrite your `where` addition to use the alias given to your extra table. The alias will be the same each time you construct the queryset in the same way, so you can rely upon the alias name to not change.

- `order_by`

  If you need to order the resulting queryset using some of the new fields or tables you have included via `extra()` use the `order_by` parameter to `extra()` and pass in a sequence of strings. These strings should either be model fields (as in the normal *`order_by()`* method on querysets), of the form `table_name.column_name` or an alias for a column that you specified in the `select` parameter to `extra()`.

  For example:

```
q = Entry.objects.extra(select={'is_recent': "pub_date > '2006-01-01'"})
q = q.extra(order_by = ['-is_recent'])
```

  This would sort all the items for which `is_recent` is true to the front of the result set (`True` sorts before `False` in a descending ordering).

  This shows, by the way, that you can make multiple calls to `extra()` and it will behave as you expect (adding new constraints each time).

- `params`

  The `where` parameter described above may use standard Python database string placeholders — `'%s'` to indicate parameters the database engine should automatically quote. The `params` argument is a list of any extra parameters to be substituted.

  Example:

```
Entry.objects.extra(where=['headline=%s'], params=['Lennon'])
```

  Always use `params` instead of embedding values directly into `where` because `params` will ensure values are quoted correctly according to your particular backend. For example, quotes will be escaped correctly.

Bad:

```
Entry.objects.extra(where=["headline='Lennon'"])
```

Good:

```
Entry.objects.extra(where=['headline=%s'], params=['Lennon'])
```

: If you are performing queries on MySQL, note that MySQL's silent type coercion may cause unexpected results when mixing types. If you query on a string type column, but with an integer value, MySQL will coerce the types of all values in the table to an integer before performing the comparison. For example, if your table contains the values `'abc'`, `'def'` and you query for `WHERE mycolumn=0`, both rows will match. To prevent this, perform the correct typecasting before using the value in a query.

defer()
defer(**fields*)
In some complex data-modeling situations, your models might contain a lot of fields, some of which could contain a lot of data (for example, text fields), or require expensive processing to convert them to Python objects. If you are using the results of a queryset in some situation where you don't know if you need those particular fields when you initially fetch the data, you can tell Django not to retrieve them from the database.

This is done by passing the names of the fields to not load to `defer()`:

```
Entry.objects.defer("headline", "body")
```

A queryset that has deferred fields will still return model instances. Each deferred field will be retrieved from the database if you access that field (one at a time, not all the deferred fields at once).

You can make multiple calls to `defer()`. Each call adds new fields to the deferred set:

```
# Defers both the body and headline fields.
Entry.objects.defer("body").filter(rating=5).defer("headline")
```

The order in which fields are added to the deferred set does not matter. Calling `defer()` with a field name that has already been deferred is harmless (the field will still be deferred).

You can defer loading of fields in related models (if the related models are loading via *`select_related()`*) by using the standard double-underscore notation to separate related fields:

```
Blog.objects.select_related().defer("entry__headline", "entry__body")
```

If you want to clear the set of deferred fields, pass `None` as a parameter to `defer()`:

```
# Load all fields immediately.
my_queryset.defer(None)
```

Some fields in a model won't be deferred, even if you ask for them. You can never defer the loading of the primary key. If you are using *`select_related()`* to retrieve related models, you shouldn't defer the loading of the field that connects from the primary model to the related one, doing so will result in an error.

: The `defer()` method (and its cousin, *`only()`*, below) are only for advanced use-cases. They provide an optimization for when you have analyzed your queries closely and understand *exactly* what information you need and have measured that the difference between returning the fields you need and the full set of fields for the model will be significant.

Even if you think you are in the advanced use-case situation, **only use defer() when you cannot, at queryset load time, determine if you will need the extra fields or not**. If you are frequently loading and using a particular subset of

your data, the best choice you can make is to normalize your models and put the non-loaded data into a separate model (and database table). If the columns *must* stay in the one table for some reason, create a model with `Meta.managed = False` (see the *managed attribute* documentation) containing just the fields you normally need to load and use that where you might otherwise call `defer()`. This makes your code more explicit to the reader, is slightly faster and consumes a little less memory in the Python process.

For example, both of these models use the same underlying database table:

```
class CommonlyUsedModel(models.Model):
    f1 = models.CharField(max_length=10)

    class Meta:
        managed = False
        db_table = 'app_largetable'

class ManagedModel(models.Model):
    f1 = models.CharField(max_length=10)
    f2 = models.CharField(max_length=10)

    class Meta:
        db_table = 'app_largetable'

# Two equivalent QuerySets:
CommonlyUsedModel.objects.all()
ManagedModel.objects.all().defer('f2')
```

If many fields need to be duplicated in the unmanaged model, it may be best to create an abstract model with the shared fields and then have the unmanaged and managed models inherit from the abstract model.

: When calling *save()* for instances with deferred fields, only the loaded fields will be saved. See *save()* for more details.

---

only()
only(**fields*)
The `only()` method is more or less the opposite of *defer()*. You call it with the fields that should *not* be deferred when retrieving a model. If you have a model where almost all the fields need to be deferred, using `only()` to specify the complementary set of fields can result in simpler code.

Suppose you have a model with fields `name`, `age` and `biography`. The following two querysets are the same, in terms of deferred fields:

```
Person.objects.defer("age", "biography")
Person.objects.only("name")
```

Whenever you call `only()` it *replaces* the set of fields to load immediately. The method's name is mnemonic: **only** those fields are loaded immediately; the remainder are deferred. Thus, successive calls to `only()` result in only the final fields being considered:

```
# This will defer all fields except the headline.
Entry.objects.only("body", "rating").only("headline")
```

Since `defer()` acts incrementally (adding fields to the deferred list), you can combine calls to `only()` and `defer()` and things will behave logically:

```
# Final result is that everything except "headline" is deferred.
Entry.objects.only("headline", "body").defer("body")

# Final result loads headline and body immediately (only() replaces any
# existing set of fields).
Entry.objects.defer("body").only("headline", "body")
```

All of the cautions in the note for the *defer()* documentation apply to `only()` as well. Use it cautiously and only after exhausting your other options.

Using *only()* and omitting a field requested using *select_related()* is an error as well.

: When calling *save()* for instances with deferred fields, only the loaded fields will be saved. See *save()* for more details.

---

using()
using(*alias*)
This method is for controlling which database the `QuerySet` will be evaluated against if you are using more than one database. The only argument this method takes is the alias of a database, as defined in *DATABASES*.

For example:

```
# queries the database with the 'default' alias.
>>> Entry.objects.all()

# queries the database with the 'backup' alias
>>> Entry.objects.using('backup')
```

select_for_update()
select_for_update(*nowait=False*)
Returns a queryset that will lock rows until the end of the transaction, generating a `SELECT ... FOR UPDATE` SQL statement on supported databases.

For example:

```
entries = Entry.objects.select_for_update().filter(author=request.user)
```

All matched entries will be locked until the end of the transaction block, meaning that other transactions will be prevented from changing or acquiring locks on them.

Usually, if another transaction has already acquired a lock on one of the selected rows, the query will block until the lock is released. If this is not the behavior you want, call `select_for_update(nowait=True)`. This will make the call non-blocking. If a conflicting lock is already acquired by another transaction, *DatabaseError* will be raised when the queryset is evaluated.

Currently, the `postgresql`, `oracle`, and `mysql` database backends support `select_for_update()`. However, MySQL has no support for the `nowait` argument. Obviously, users of external third-party backends should check with their backend's documentation for specifics in those cases.

Passing `nowait=True` to `select_for_update()` using database backends that do not support `nowait`, such as MySQL, will cause a *DatabaseError* to be raised. This is in order to prevent code unexpectedly blocking.

Evaluating a queryset with `select_for_update()` in autocommit mode on backends which support `SELECT ... FOR UPDATE` is a *TransactionManagementError* error because the rows are not locked in that case. If allowed, this would facilitate data corruption and could easily be caused by calling code that expects to be run in a transaction outside of one.

Using `select_for_update()` on backends which do not support `SELECT ... FOR UPDATE` (such as SQLite) will have no effect. `SELECT ... FOR UPDATE` will not be added to the query, and an error isn't raised if `select_for_update()` is used in autocommit mode.

: Although `select_for_update()` normally fails in autocommit mode, since *`TestCase`* automatically wraps each test in a transaction, calling `select_for_update()` in a `TestCase` even outside an *`atomic()`* block will (perhaps unexpectedly) pass without raising a `TransactionManagementError`. To properly test `select_for_update()` you should use *`TransactionTestCase`*.

raw()
raw(*raw_query*, *params=None*, *translations=None*)
Takes a raw SQL query, executes it, and returns a `django.db.models.query.RawQuerySet` instance. This `RawQuerySet` instance can be iterated over just like an normal `QuerySet` to provide object instances.

See the Performing raw SQL queries for more information.

: `raw()` always triggers a new query and doesn't account for previous filtering. As such, it should generally be called from the `Manager` or from a fresh `QuerySet` instance.

Methods that do not return `QuerySets`

The following `QuerySet` methods evaluate the `QuerySet` and return something *other than* a `QuerySet`.

These methods do not use a cache (see *Caching and QuerySets*). Rather, they query the database each time they're called.

get()
get(***kwargs*)
Returns the object matching the given lookup parameters, which should be in the format described in *Field lookups*.

`get()` raises *`MultipleObjectsReturned`* if more than one object was found. The *`MultipleObjectsReturned`* exception is an attribute of the model class.

`get()` raises a *`DoesNotExist`* exception if an object wasn't found for the given parameters. This exception is an attribute of the model class. Example:

```
Entry.objects.get(id='foo') # raises Entry.DoesNotExist
```

The *`DoesNotExist`* exception inherits from *`django.core.exceptions.ObjectDoesNotExist`*, so you can target multiple *`DoesNotExist`* exceptions. Example:

```
from django.core.exceptions import ObjectDoesNotExist
try:
    e = Entry.objects.get(id=3)
    b = Blog.objects.get(id=1)
except ObjectDoesNotExist:
    print("Either the entry or blog doesn't exist.")
```

If you expect a queryset to return one row, you can use `get()` without any arguments to return the object for that row:

```
entry = Entry.objects.filter(...).exclude(...).get()
```

create()
create(***kwargs*)
A convenience method for creating an object and saving it all in one step. Thus:

```
p = Person.objects.create(first_name="Bruce", last_name="Springsteen")
```

and:

```
p = Person(first_name="Bruce", last_name="Springsteen")
p.save(force_insert=True)
```

are equivalent.

The *force_insert* parameter is documented elsewhere, but all it means is that a new object will always be created. Normally you won't need to worry about this. However, if your model contains a manual primary key value that you set and if that value already exists in the database, a call to `create()` will fail with an *IntegrityError* since primary keys must be unique. Be prepared to handle the exception if you are using manual primary keys.

get_or_create()
get_or_create(*defaults=None*, ***kwargs*)
A convenience method for looking up an object with the given `kwargs` (may be empty if your model has defaults for all fields), creating one if necessary.

Returns a tuple of `(object, created)`, where `object` is the retrieved or created object and `created` is a boolean specifying whether a new object was created.

This is meant as a shortcut to boilerplatish code. For example:

```
try:
    obj = Person.objects.get(first_name='John', last_name='Lennon')
except Person.DoesNotExist:
    obj = Person(first_name='John', last_name='Lennon', birthday=date(1940, 10, 9))
    obj.save()
```

This pattern gets quite unwieldy as the number of fields in a model goes up. The above example can be rewritten using `get_or_create()` like so:

```
obj, created = Person.objects.get_or_create(
    first_name='John',
    last_name='Lennon',
    defaults={'birthday': date(1940, 10, 9)},
)
```

Any keyword arguments passed to `get_or_create()` — *except* an optional one called `defaults` — will be used in a *get()* call. If an object is found, `get_or_create()` returns a tuple of that object and `False`. If multiple objects are found, `get_or_create` raises *MultipleObjectsReturned*. If an object is *not* found, `get_or_create()` will instantiate and save a new object, returning a tuple of the new object and `True`. The new object will be created roughly according to this algorithm:

```
params = {k: v for k, v in kwargs.items() if '__' not in k}
params.update(defaults)
obj = self.model(**params)
obj.save()
```

In English, that means start with any non-`'defaults'` keyword argument that doesn't contain a double underscore (which would indicate a non-exact lookup). Then add the contents of `defaults`, overriding any keys if necessary, and use the result as the keyword arguments to the model class. As hinted at above, this is a simplification of the algorithm that is used, but it contains all the pertinent details. The internal implementation has some more error-checking than this and handles some extra edge-conditions; if you're interested, read the code.

If you have a field named `defaults` and want to use it as an exact lookup in `get_or_create()`, just use `'defaults__exact'`, like so:

```
Foo.objects.get_or_create(defaults__exact='bar', defaults={'defaults': 'baz'})
```

The `get_or_create()` method has similar error behavior to *create()* when you're using manually specified primary keys. If an object needs to be created and the key already exists in the database, an *IntegrityError* will be raised.

This method is atomic assuming correct usage, correct database configuration, and correct behavior of the underlying database. However, if uniqueness is not enforced at the database level for the `kwargs` used in a `get_or_create` call (see *unique* or *unique_together*), this method is prone to a race-condition which can result in multiple rows with the same parameters being inserted simultaneously.

If you are using MySQL, be sure to use the `READ COMMITTED` isolation level rather than `REPEATABLE READ` (the default), otherwise you may see cases where `get_or_create` will raise an *IntegrityError* but the object won't appear in a subsequent *get()* call.

Finally, a word on using `get_or_create()` in Django views. Please make sure to use it only in `POST` requests unless you have a good reason not to. `GET` requests shouldn't have any effect on data. Instead, use `POST` whenever a request to a page has a side effect on your data. For more, see **Safe methods** in the HTTP spec.

: You can use `get_or_create()` through *ManyToManyField* attributes and reverse relations. In that case you will restrict the queries inside the context of that relation. That could lead you to some integrity problems if you don't use it consistently.
Being the following models:

```
class Chapter(models.Model):
    title = models.CharField(max_length=255, unique=True)

class Book(models.Model):
    title = models.CharField(max_length=256)
    chapters = models.ManyToManyField(Chapter)
```

You can use `get_or_create()` through Book's chapters field, but it only fetches inside the context of that book:

```
>>> book = Book.objects.create(title="Ulysses")
>>> book.chapters.get_or_create(title="Telemachus")
(<Chapter: Telemachus>, True)
>>> book.chapters.get_or_create(title="Telemachus")
(<Chapter: Telemachus>, False)
>>> Chapter.objects.create(title="Chapter 1")
<Chapter: Chapter 1>
>>> book.chapters.get_or_create(title="Chapter 1")
# Raises IntegrityError
```

This is happening because it's trying to get or create "Chapter 1" through the book "Ulysses", but it can't do any of them: the relation can't fetch that chapter because it isn't related to that book, but it can't create it either because `title` field should be unique.

update_or_create()

update_or_create(*defaults=None*, ***kwargs*)

A convenience method for updating an object with the given `kwargs`, creating a new one if necessary. The `defaults` is a dictionary of (field, value) pairs used to update the object.

Returns a tuple of `(object, created)`, where `object` is the created or updated object and `created` is a boolean specifying whether a new object was created.

The `update_or_create` method tries to fetch an object from database based on the given `kwargs`. If a match is found, it updates the fields passed in the `defaults` dictionary.

This is meant as a shortcut to boilerplatish code. For example:

```
defaults = {'first_name': 'Bob'}
try:
    obj = Person.objects.get(first_name='John', last_name='Lennon')
    for key, value in defaults.items():
        setattr(obj, key, value)
    obj.save()
except Person.DoesNotExist:
    new_values = {'first_name': 'John', 'last_name': 'Lennon'}
    new_values.update(defaults)
    obj = Person(**new_values)
    obj.save()
```

This pattern gets quite unwieldy as the number of fields in a model goes up. The above example can be rewritten using `update_or_create()` like so:

```
obj, created = Person.objects.update_or_create(
    first_name='John', last_name='Lennon',
    defaults={'first_name': 'Bob'},
)
```

For detailed description how names passed in `kwargs` are resolved see *`get_or_create()`*.

As described above in *`get_or_create()`*, this method is prone to a race-condition which can result in multiple rows being inserted simultaneously if uniqueness is not enforced at the database level.

`bulk_create()`
`bulk_create`(*objs*, *batch_size=None*)
This method inserts the provided list of objects into the database in an efficient manner (generally only 1 query, no matter how many objects there are):

```
>>> Entry.objects.bulk_create([
...     Entry(headline="Django 1.0 Released"),
...     Entry(headline="Django 1.1 Announced"),
...     Entry(headline="Breaking: Django is awesome")
... ])
```

This has a number of caveats though:

- The model's `save()` method will not be called, and the `pre_save` and `post_save` signals will not be sent.
- It does not work with child models in a multi-table inheritance scenario.
- If the model's primary key is an *`AutoField`* it does not retrieve and set the primary key attribute, as `save()` does, unless the database backend supports it (currently PostgreSQL).
- It does not work with many-to-many relationships.

Support for using `bulk_create()` with proxy models was added.

Support for setting primary keys on objects created using `bulk_create()` when using PostgreSQL was added.

The `batch_size` parameter controls how many objects are created in single query. The default is to create all objects in one batch, except for SQLite where the default is such that at most 999 variables per query are used.

count()
count()
Returns an integer representing the number of objects in the database matching the QuerySet. The count() method never raises exceptions.

Example:

```
# Returns the total number of entries in the database.
Entry.objects.count()

# Returns the number of entries whose headline contains 'Lennon'
Entry.objects.filter(headline__contains='Lennon').count()
```

A count() call performs a SELECT COUNT(*) behind the scenes, so you should always use count() rather than loading all of the record into Python objects and calling len() on the result (unless you need to load the objects into memory anyway, in which case len() will be faster).

Depending on which database you're using (e.g. PostgreSQL vs. MySQL), count() may return a long integer instead of a normal Python integer. This is an underlying implementation quirk that shouldn't pose any real-world problems.

Note that if you want the number of items in a QuerySet and are also retrieving model instances from it (for example, by iterating over it), it's probably more efficient to use len(queryset) which won't cause an extra database query like count() would.

in_bulk()
in_bulk(*id_list=None*)
Takes a list of primary-key values and returns a dictionary mapping each primary-key value to an instance of the object with the given ID. If a list isn't provided, all objects in the queryset are returned.

Example:

```
>>> Blog.objects.in_bulk([1])
{1: <Blog: Beatles Blog>}
>>> Blog.objects.in_bulk([1, 2])
{1: <Blog: Beatles Blog>, 2: <Blog: Cheddar Talk>}
>>> Blog.objects.in_bulk([])
{}
>>> Blog.objects.in_bulk()
{1: <Blog: Beatles Blog>, 2: <Blog: Cheddar Talk>, 3: <Blog: Django Weblog>}
```

If you pass in_bulk() an empty list, you'll get an empty dictionary.

In older versions, id_list was a required argument.

iterator()
iterator()
Evaluates the QuerySet (by performing the query) and returns an iterator (see **PEP 234**) over the results. A QuerySet typically caches its results internally so that repeated evaluations do not result in additional queries. In contrast, iterator() will read results directly, without doing any caching at the QuerySet level (internally, the default iterator calls iterator() and caches the return value). For a QuerySet which returns a large number of objects that you only need to access once, this can result in better performance and a significant reduction in memory.

Note that using iterator() on a QuerySet which has already been evaluated will force it to evaluate again, repeating the query.

Also, use of iterator() causes previous prefetch_related() calls to be ignored since these two optimizations do not make sense together.

: Some Python database drivers like `psycopg2` perform caching if using client side cursors (instantiated with `connection.cursor()` and what Django's ORM uses). Using `iterator()` does not affect caching at the database driver level. To disable this caching, look at server side cursors.

`latest()`
`latest`(*field_name=None*)
Returns the latest object in the table, by date, using the `field_name` provided as the date field.

This example returns the latest `Entry` in the table, according to the `pub_date` field:

```
Entry.objects.latest('pub_date')
```

If your model's *Meta* specifies *`get_latest_by`*, you can leave off the `field_name` argument to `earliest()` or `latest()`. Django will use the field specified in *`get_latest_by`* by default.

Like *`get()`*, `earliest()` and `latest()` raise *`DoesNotExist`* if there is no object with the given parameters.

Note that `earliest()` and `latest()` exist purely for convenience and readability.

`earliest()` and `latest()` may return instances with null dates.

Since ordering is delegated to the database, results on fields that allow null values may be ordered differently if you use different databases. For example, PostgreSQL and MySQL sort null values as if they are higher than non-null values, while SQLite does the opposite.

You may want to filter out null values:

```
Entry.objects.filter(pub_date__isnull=False).latest('pub_date')
```

`earliest()`
`earliest`(*field_name=None*)
Works otherwise like *`latest()`* except the direction is changed.

`first()`
`first`()
Returns the first object matched by the queryset, or `None` if there is no matching object. If the `QuerySet` has no ordering defined, then the queryset is automatically ordered by the primary key.

Example:

```
p = Article.objects.order_by('title', 'pub_date').first()
```

Note that `first()` is a convenience method, the following code sample is equivalent to the above example:

```
try:
    p = Article.objects.order_by('title', 'pub_date')[0]
except IndexError:
    p = None
```

`last()`
`last`()
Works like *`first()`*, but returns the last object in the queryset.

aggregate()
aggregate(**args*, ***kwargs*)
Returns a dictionary of aggregate values (averages, sums, etc.) calculated over the `QuerySet`. Each argument to `aggregate()` specifies a value that will be included in the dictionary that is returned.

The aggregation functions that are provided by Django are described in *Aggregation Functions* below. Since aggregates are also query expressions, you may combine aggregates with other aggregates or values to create complex aggregates.

Aggregates specified using keyword arguments will use the keyword as the name for the annotation. Anonymous arguments will have a name generated for them based upon the name of the aggregate function and the model field that is being aggregated. Complex aggregates cannot use anonymous arguments and must specify a keyword argument as an alias.

For example, when you are working with blog entries, you may want to know the number of authors that have contributed blog entries:

```
>>> from django.db.models import Count
>>> q = Blog.objects.aggregate(Count('entry'))
{'entry__count': 16}
```

By using a keyword argument to specify the aggregate function, you can control the name of the aggregation value that is returned:

```
>>> q = Blog.objects.aggregate(number_of_entries=Count('entry'))
{'number_of_entries': 16}
```

For an in-depth discussion of aggregation, see the topic guide on Aggregation.

exists()
exists()
Returns `True` if the *QuerySet* contains any results, and `False` if not. This tries to perform the query in the simplest and fastest way possible, but it *does* execute nearly the same query as a normal *QuerySet* query.

exists() is useful for searches relating to both object membership in a *QuerySet* and to the existence of any objects in a *QuerySet*, particularly in the context of a large *QuerySet*.

The most efficient method of finding whether a model with a unique field (e.g. `primary_key`) is a member of a *QuerySet* is:

```
entry = Entry.objects.get(pk=123)
if some_queryset.filter(pk=entry.pk).exists():
    print("Entry contained in queryset")
```

Which will be faster than the following which requires evaluating and iterating through the entire queryset:

```
if entry in some_queryset:
   print("Entry contained in QuerySet")
```

And to find whether a queryset contains any items:

```
if some_queryset.exists():
    print("There is at least one object in some_queryset")
```

Which will be faster than:

```
if some_queryset:
    print("There is at least one object in some_queryset")
```

... but not by a large degree (hence needing a large queryset for efficiency gains).

Additionally, if a `some_queryset` has not yet been evaluated, but you know that it will be at some point, then using `some_queryset.exists()` will do more overall work (one query for the existence check plus an extra one to later retrieve the results) than simply using `bool(some_queryset)`, which retrieves the results and then checks if any were returned.

update()
update(***kwargs*)
Performs an SQL update query for the specified fields, and returns the number of rows matched (which may not be equal to the number of rows updated if some rows already have the new value).

For example, to turn comments off for all blog entries published in 2010, you could do this:

```
>>> Entry.objects.filter(pub_date__year=2010).update(comments_on=False)
```

(This assumes your `Entry` model has fields `pub_date` and `comments_on`.)

You can update multiple fields — there's no limit on how many. For example, here we update the `comments_on` and `headline` fields:

```
>>> Entry.objects.filter(pub_date__year=2010).update(comments_on=False, headline='This is old')
```

The `update()` method is applied instantly, and the only restriction on the *QuerySet* that is updated is that it can only update columns in the model's main table, not on related models. You can't do this, for example:

```
>>> Entry.objects.update(blog__name='foo') # Won't work!
```

Filtering based on related fields is still possible, though:

```
>>> Entry.objects.filter(blog__id=1).update(comments_on=True)
```

You cannot call `update()` on a *QuerySet* that has had a slice taken or can otherwise no longer be filtered.

The `update()` method returns the number of affected rows:

```
>>> Entry.objects.filter(id=64).update(comments_on=True)
1

>>> Entry.objects.filter(slug='nonexistent-slug').update(comments_on=True)
0

>>> Entry.objects.filter(pub_date__year=2010).update(comments_on=False)
132
```

If you're just updating a record and don't need to do anything with the model object, the most efficient approach is to call `update()`, rather than loading the model object into memory. For example, instead of doing this:

```
e = Entry.objects.get(id=10)
e.comments_on = False
e.save()
```

...do this:

```
Entry.objects.filter(id=10).update(comments_on=False)
```

Using `update()` also prevents a race condition wherein something might change in your database in the short period of time between loading the object and calling `save()`.

Finally, realize that `update()` does an update at the SQL level and, thus, does not call any `save()` methods on your models, nor does it emit the *pre_save* or *post_save* signals (which are a consequence of calling *Model.save()*). If you want to update a bunch of records for a model that has a custom *save()* method, loop over them and call *save()*, like this:

```
for e in Entry.objects.filter(pub_date__year=2010):
    e.comments_on = False
    e.save()
```

delete()
delete()
Performs an SQL delete query on all rows in the *QuerySet* and returns the number of objects deleted and a dictionary with the number of deletions per object type.

The delete() is applied instantly. You cannot call delete() on a *QuerySet* that has had a slice taken or can otherwise no longer be filtered.

For example, to delete all the entries in a particular blog:

```
>>> b = Blog.objects.get(pk=1)

# Delete all the entries belonging to this Blog.
>>> Entry.objects.filter(blog=b).delete()
(4, {'weblog.Entry': 2, 'weblog.Entry_authors': 2})
```

The return value describing the number of objects deleted was added.

By default, Django's *ForeignKey* emulates the SQL constraint ON DELETE CASCADE — in other words, any objects with foreign keys pointing at the objects to be deleted will be deleted along with them. For example:

```
>>> blogs = Blog.objects.all()

# This will delete all Blogs and all of their Entry objects.
>>> blogs.delete()
(5, {'weblog.Blog': 1, 'weblog.Entry': 2, 'weblog.Entry_authors': 2})
```

This cascade behavior is customizable via the *on_delete* argument to the *ForeignKey*.

The delete() method does a bulk delete and does not call any delete() methods on your models. It does, however, emit the *pre_delete* and *post_delete* signals for all deleted objects (including cascaded deletions).

Django needs to fetch objects into memory to send signals and handle cascades. However, if there are no cascades and no signals, then Django may take a fast-path and delete objects without fetching into memory. For large deletes this can result in significantly reduced memory usage. The amount of executed queries can be reduced, too.

ForeignKeys which are set to *on_delete* DO_NOTHING do not prevent taking the fast-path in deletion.

Note that the queries generated in object deletion is an implementation detail subject to change.

as_manager()
classmethod as_manager()
Class method that returns an instance of *Manager* with a copy of the QuerySet's methods. See *Creating a manager with QuerySet methods* for more details.

Field lookups

Field lookups are how you specify the meat of an SQL WHERE clause. They're specified as keyword arguments to the QuerySet methods *filter()*, *exclude()* and *get()*.

For an introduction, see *models and database queries documentation*.

Django's built-in lookups are listed below. It is also possible to write custom lookups for model fields.

As a convenience when no lookup type is provided (like in `Entry.objects.get(id=14)`) the lookup type is assumed to be *exact*.

exact Exact match. If the value provided for comparison is `None`, it will be interpreted as an SQL `NULL` (see *isnull* for more details).

Examples:

```
Entry.objects.get(id__exact=14)
Entry.objects.get(id__exact=None)
```

SQL equivalents:

```
SELECT ... WHERE id = 14;
SELECT ... WHERE id IS NULL;
```

MySQL comparisons

In MySQL, a database table's "collation" setting determines whether `exact` comparisons are case-sensitive. This is a database setting, *not* a Django setting. It's possible to configure your MySQL tables to use case-sensitive comparisons, but some trade-offs are involved. For more information about this, see the *collation section* in the databases documentation.

iexact Case-insensitive exact match. If the value provided for comparison is `None`, it will be interpreted as an SQL `NULL` (see *isnull* for more details).

Example:

```
Blog.objects.get(name__iexact='beatles blog')
Blog.objects.get(name__iexact=None)
```

SQL equivalents:

```
SELECT ... WHERE name ILIKE 'beatles blog';
SELECT ... WHERE name IS NULL;
```

Note the first query will match `'Beatles Blog'`, `'beatles blog'`, `'BeAtLes BLoG'`, etc.

SQLite users

When using the SQLite backend and Unicode (non-ASCII) strings, bear in mind the *database note* about string comparisons. SQLite does not do case-insensitive matching for Unicode strings.

contains Case-sensitive containment test.

Example:

```
Entry.objects.get(headline__contains='Lennon')
```

SQL equivalent:

```
SELECT ... WHERE headline LIKE '%Lennon%';
```

Note this will match the headline `'Lennon honored today'` but not `'lennon honored today'`.

SQLite users

SQLite doesn't support case-sensitive `LIKE` statements; `contains` acts like `icontains` for SQLite. See the *database note* for more information.

`icontains` Case-insensitive containment test.

Example:

```
Entry.objects.get(headline__icontains='Lennon')
```

SQL equivalent:

```
SELECT ... WHERE headline ILIKE '%Lennon%';
```

SQLite users

When using the SQLite backend and Unicode (non-ASCII) strings, bear in mind the *database note* about string comparisons.

`in` In a given list.

Example:

```
Entry.objects.filter(id__in=[1, 3, 4])
```

SQL equivalent:

```
SELECT ... WHERE id IN (1, 3, 4);
```

You can also use a queryset to dynamically evaluate the list of values instead of providing a list of literal values:

```
inner_qs = Blog.objects.filter(name__contains='Cheddar')
entries = Entry.objects.filter(blog__in=inner_qs)
```

This queryset will be evaluated as subselect statement:

```
SELECT ... WHERE blog.id IN (SELECT id FROM ... WHERE NAME LIKE '%Cheddar%')
```

If you pass in a `QuerySet` resulting from `values()` or `values_list()` as the value to an `__in` lookup, you need to ensure you are only extracting one field in the result. For example, this will work (filtering on the blog names):

```
inner_qs = Blog.objects.filter(name__contains='Ch').values('name')
entries = Entry.objects.filter(blog__name__in=inner_qs)
```

This example will raise an exception, since the inner query is trying to extract two field values, where only one is expected:

```
# Bad code! Will raise a TypeError.
inner_qs = Blog.objects.filter(name__contains='Ch').values('name', 'id')
entries = Entry.objects.filter(blog__name__in=inner_qs)
```

Performance considerations

Be cautious about using nested queries and understand your database server's performance characteristics (if in doubt, benchmark!). Some database backends, most notably MySQL, don't optimize nested queries very well. It is more efficient, in those cases, to extract a list of values and then pass that into the second query. That is, execute two queries instead of one:

```
values = Blog.objects.filter(
        name__contains='Cheddar').values_list('pk', flat=True)
entries = Entry.objects.filter(blog__in=list(values))
```

Note the `list()` call around the Blog `QuerySet` to force execution of the first query. Without it, a nested query would be executed, because *QuerySets are lazy*.

gt Greater than.

Example:

```
Entry.objects.filter(id__gt=4)
```

SQL equivalent:

```
SELECT ... WHERE id > 4;
```

gte Greater than or equal to.

lt Less than.

lte Less than or equal to.

startswith Case-sensitive starts-with.

Example:

```
Entry.objects.filter(headline__startswith='Will')
```

SQL equivalent:

```
SELECT ... WHERE headline LIKE 'Will%';
```

SQLite doesn't support case-sensitive `LIKE` statements; `startswith` acts like `istartswith` for SQLite.

istartswith Case-insensitive starts-with.

Example:

```
Entry.objects.filter(headline__istartswith='will')
```

SQL equivalent:

```
SELECT ... WHERE headline ILIKE 'Will%';
```

SQLite users

When using the SQLite backend and Unicode (non-ASCII) strings, bear in mind the *database note* about string comparisons.

endswith Case-sensitive ends-with.

Example:

```
Entry.objects.filter(headline__endswith='cats')
```

SQL equivalent:

```
SELECT ... WHERE headline LIKE '%cats';
```

SQLite users

SQLite doesn't support case-sensitive `LIKE` statements; `endswith` acts like `iendswith` for SQLite. Refer to the *database note* documentation for more.

iendswith Case-insensitive ends-with.

Example:

```
Entry.objects.filter(headline__iendswith='will')
```

SQL equivalent:

```
SELECT ... WHERE headline ILIKE '%will'
```

SQLite users

When using the SQLite backend and Unicode (non-ASCII) strings, bear in mind the *database note* about string comparisons.

range Range test (inclusive).

Example:

```
import datetime
start_date = datetime.date(2005, 1, 1)
end_date = datetime.date(2005, 3, 31)
Entry.objects.filter(pub_date__range=(start_date, end_date))
```

SQL equivalent:

```
SELECT ... WHERE pub_date BETWEEN '2005-01-01' and '2005-03-31';
```

You can use `range` anywhere you can use `BETWEEN` in SQL — for dates, numbers and even characters.

: Filtering a `DateTimeField` with dates won't include items on the last day, because the bounds are interpreted as "0am on the given date". If `pub_date` was a `DateTimeField`, the above expression would be turned into this SQL:

```
SELECT ... WHERE pub_date BETWEEN '2005-01-01 00:00:00' and '2005-03-31 00:00:00';
```

Generally speaking, you can't mix dates and datetimes.

date For datetime fields, casts the value as date. Allows chaining additional field lookups. Takes a date value.

Example:

```
Entry.objects.filter(pub_date__date=datetime.date(2005, 1, 1))
Entry.objects.filter(pub_date__date__gt=datetime.date(2005, 1, 1))
```

(No equivalent SQL code fragment is included for this lookup because implementation of the relevant query varies among different database engines.)

When *USE_TZ* is `True`, fields are converted to the current time zone before filtering.

year For date and datetime fields, an exact year match. Allows chaining additional field lookups. Takes an integer year.

Example:

```
Entry.objects.filter(pub_date__year=2005)
Entry.objects.filter(pub_date__year__gte=2005)
```

SQL equivalent:

```
SELECT ... WHERE pub_date BETWEEN '2005-01-01' AND '2005-12-31';
SELECT ... WHERE pub_date >= '2005-01-01';
```

(The exact SQL syntax varies for each database engine.)

When *USE_TZ* is `True`, datetime fields are converted to the current time zone before filtering.

Allowed chaining additional field lookups.

month For date and datetime fields, an exact month match. Allows chaining additional field lookups. Takes an integer 1 (January) through 12 (December).

Example:

```
Entry.objects.filter(pub_date__month=12)
Entry.objects.filter(pub_date__month__gte=6)
```

SQL equivalent:

```
SELECT ... WHERE EXTRACT('month' FROM pub_date) = '12';
SELECT ... WHERE EXTRACT('month' FROM pub_date) >= '6';
```

(The exact SQL syntax varies for each database engine.)

When *USE_TZ* is `True`, datetime fields are converted to the current time zone before filtering. This requires *time zone definitions in the database*.

Allowed chaining additional field lookups.

day For date and datetime fields, an exact day match. Allows chaining additional field lookups. Takes an integer day.

Example:

```
Entry.objects.filter(pub_date__day=3)
Entry.objects.filter(pub_date__day__gte=3)
```

SQL equivalent:

```
SELECT ... WHERE EXTRACT('day' FROM pub_date) = '3';
SELECT ... WHERE EXTRACT('day' FROM pub_date) >= '3';
```

(The exact SQL syntax varies for each database engine.)

Note this will match any record with a pub_date on the third day of the month, such as January 3, July 3, etc.

When *USE_TZ* is `True`, datetime fields are converted to the current time zone before filtering. This requires *time zone definitions in the database*.

Allowed chaining additional field lookups.

week_day For date and datetime fields, a 'day of the week' match. Allows chaining additional field lookups.

Takes an integer value representing the day of week from 1 (Sunday) to 7 (Saturday).

Example:

```
Entry.objects.filter(pub_date__week_day=2)
Entry.objects.filter(pub_date__week_day__gte=2)
```

(No equivalent SQL code fragment is included for this lookup because implementation of the relevant query varies among different database engines.)

Note this will match any record with a `pub_date` that falls on a Monday (day 2 of the week), regardless of the month or year in which it occurs. Week days are indexed with day 1 being Sunday and day 7 being Saturday.

When *USE_TZ* is `True`, datetime fields are converted to the current time zone before filtering. This requires *time zone definitions in the database*.

Allowed chaining additional field lookups.

hour For datetime and time fields, an exact hour match. Allows chaining additional field lookups. Takes an integer between 0 and 23.

Example:

```
Event.objects.filter(timestamp__hour=23)
Event.objects.filter(time__hour=5)
Event.objects.filter(timestamp__hour__gte=12)
```

SQL equivalent:

```
SELECT ... WHERE EXTRACT('hour' FROM timestamp) = '23';
SELECT ... WHERE EXTRACT('hour' FROM time) = '5';
SELECT ... WHERE EXTRACT('hour' FROM timestamp) >= '12';
```

(The exact SQL syntax varies for each database engine.)

For datetime fields, when *USE_TZ* is `True`, values are converted to the current time zone before filtering.

Added support for *TimeField* on SQLite (other databases supported it as of 1.7).

Allowed chaining additional field lookups.

minute For datetime and time fields, an exact minute match. Allows chaining additional field lookups. Takes an integer between 0 and 59.

Example:

```
Event.objects.filter(timestamp__minute=29)
Event.objects.filter(time__minute=46)
Event.objects.filter(timestamp__minute__gte=29)
```

SQL equivalent:

```
SELECT ... WHERE EXTRACT('minute' FROM timestamp) = '29';
SELECT ... WHERE EXTRACT('minute' FROM time) = '46';
SELECT ... WHERE EXTRACT('minute' FROM timestamp) >= '29';
```

(The exact SQL syntax varies for each database engine.)

For datetime fields, When *USE_TZ* is `True`, values are converted to the current time zone before filtering.

Added support for *TimeField* on SQLite (other databases supported it as of 1.7).

Allowed chaining additional field lookups.

second For datetime and time fields, an exact second match. Allows chaining additional field lookups. Takes an integer between 0 and 59.

Example:

```
Event.objects.filter(timestamp__second=31)
Event.objects.filter(time__second=2)
Event.objects.filter(timestamp__second__gte=31)
```

SQL equivalent:

```
SELECT ... WHERE EXTRACT('second' FROM timestamp) = '31';
SELECT ... WHERE EXTRACT('second' FROM time) = '2';
SELECT ... WHERE EXTRACT('second' FROM timestamp) >= '31';
```

(The exact SQL syntax varies for each database engine.)

For datetime fields, when *USE_TZ* is `True`, values are converted to the current time zone before filtering.

Added support for *TimeField* on SQLite (other databases supported it as of 1.7).

Allowed chaining additional field lookups.

isnull Takes either `True` or `False`, which correspond to SQL queries of `IS NULL` and `IS NOT NULL`, respectively.

Example:

```
Entry.objects.filter(pub_date__isnull=True)
```

SQL equivalent:

```
SELECT ... WHERE pub_date IS NULL;
```

search 1.10 : See *the 1.10 release notes* for how to replace it.

A boolean full-text search, taking advantage of full-text indexing. This is like *contains* but is significantly faster due to full-text indexing.

Example:

```
Entry.objects.filter(headline__search="+Django -jazz Python")
```

SQL equivalent:

```
SELECT ... WHERE MATCH(tablename, headline) AGAINST (+Django -jazz Python IN BOOLEAN MODE);
```

Note this is only available in MySQL and requires direct manipulation of the database to add the full-text index. By default Django uses BOOLEAN MODE for full text searches. See the MySQL documentation for additional details.

regex Case-sensitive regular expression match.

The regular expression syntax is that of the database backend in use. In the case of SQLite, which has no built in regular expression support, this feature is provided by a (Python) user-defined REGEXP function, and the regular expression syntax is therefore that of Python's `re` module.

Example:

```
Entry.objects.get(title__regex=r'^(An?|The) +')
```

SQL equivalents:

```
SELECT ... WHERE title REGEXP BINARY '^(An?|The) +'; -- MySQL

SELECT ... WHERE REGEXP_LIKE(title, '^(An?|The) +', 'c'); -- Oracle

SELECT ... WHERE title ~ '^(An?|The) +'; -- PostgreSQL

SELECT ... WHERE title REGEXP '^(An?|The) +'; -- SQLite
```

Using raw strings (e.g., `r'foo'` instead of `'foo'`) for passing in the regular expression syntax is recommended.

iregex Case-insensitive regular expression match.

Example:

```
Entry.objects.get(title__iregex=r'^(an?|the) +')
```

SQL equivalents:

```
SELECT ... WHERE title REGEXP '^(an?|the) +'; -- MySQL

SELECT ... WHERE REGEXP_LIKE(title, '^(an?|the) +', 'i'); -- Oracle

SELECT ... WHERE title ~* '^(an?|the) +'; -- PostgreSQL

SELECT ... WHERE title REGEXP '(?i)^(an?|the) +'; -- SQLite
```

Aggregation functions

Django provides the following aggregation functions in the `django.db.models` module. For details on how to use these aggregate functions, see the topic guide on aggregation. See the *`Aggregate`* documentation to learn how to create your aggregates.

: SQLite can't handle aggregation on date/time fields out of the box. This is because there are no native date/time fields in SQLite and Django currently emulates these features using a text field. Attempts to use aggregation on date/time fields in SQLite will raise `NotImplementedError`.

Note

Aggregation functions return `None` when used with an empty `QuerySet`. For example, the `Sum` aggregation function returns `None` instead of `0` if the `QuerySet` contains no entries. An exception is `Count`, which does return `0` if the `QuerySet` is empty.

All aggregates have the following parameters in common:

expression A string that references a field on the model, or a query expression.

output_field An optional argument that represents the model field of the return value

: When combining multiple field types, Django can only determine the `output_field` if all fields are of the same type. Otherwise, you must provide the `output_field` yourself.

****extra** Keyword arguments that can provide extra context for the SQL generated by the aggregate.

Avg

class Avg(*expression*, *output_field=FloatField()*, ***extra*)

Returns the mean value of the given expression, which must be numeric unless you specify a different `output_field`.

- Default alias: `<field>__avg`
- Return type: `float` (or the type of whatever `output_field` is specified)

The `output_field` parameter was added to allow aggregating over non-numeric columns, such as `DurationField`.

Count

class Count(*expression*, *distinct=False*, ***extra*)

Returns the number of objects that are related through the provided expression.

- Default alias: `<field>__count`
- Return type: `int`

Has one optional argument:

distinct

If `distinct=True`, the count will only include unique instances. This is the SQL equivalent of `COUNT(DISTINCT <field>)`. The default value is `False`.

Max

class Max(*expression*, *output_field=None*, ***extra*)

Returns the maximum value of the given expression.

- Default alias: `<field>__max`
- Return type: same as input field, or `output_field` if supplied

Min

class Min(*expression*, *output_field=None*, ***extra*)

Returns the minimum value of the given expression.

- Default alias: `<field>__min`
- Return type: same as input field, or `output_field` if supplied

StdDev

class StdDev(*expression*, *sample=False*, ***extra*)

Returns the standard deviation of the data in the provided expression.

- Default alias: `<field>__stddev`
- Return type: `float`

Has one optional argument:

sample

By default, `StdDev` returns the population standard deviation. However, if `sample=True`, the return value will be the sample standard deviation.

SQLite

SQLite doesn't provide `StdDev` out of the box. An implementation is available as an extension module for SQLite. Consult the SQlite documentation for instructions on obtaining and installing this extension.

Sum

class Sum(*expression*, *output_field=None*, ***extra*)

Computes the sum of all values of the given expression.

- Default alias: `<field>__sum`
- Return type: same as input field, or `output_field` if supplied

Variance

class Variance(*expression*, *sample=False*, ***extra*)

Returns the variance of the data in the provided expression.

- Default alias: `<field>__variance`
- Return type: `float`

Has one optional argument:

sample

By default, `Variance` returns the population variance. However, if `sample=True`, the return value will be the sample variance.

SQLite

SQLite doesn't provide `Variance` out of the box. An implementation is available as an extension module for SQLite. Consult the SQlite documentation for instructions on obtaining and installing this extension.

Query-related tools

This section provides reference material for query-related tools not documented elsewhere.

`Q()` objects

class `Q`

A `Q()` object, like an *`F`* object, encapsulates a SQL expression in a Python object that can be used in database-related operations.

In general, `Q() objects` make it possible to define and reuse conditions. This permits the *construction of complex database queries* using `|` (`OR`) and `&` (`AND`) operators; in particular, it is not otherwise possible to use `OR` in `QuerySets`.

`Prefetch()` objects

class `Prefetch`(*lookup*, *queryset=None*, *to_attr=None*)

The `Prefetch()` object can be used to control the operation of *`prefetch_related()`*.

The `lookup` argument describes the relations to follow and works the same as the string based lookups passed to *`prefetch_related()`*. For example:

```
>>> Question.objects.prefetch_related(Prefetch('choice_set')).get().choice_set.all()
<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]>
# This will only execute two queries regardless of the number of Question
# and Choice objects.
>>> Question.objects.prefetch_related(Prefetch('choice_set')).all()
<QuerySet [<Question: Question object>]>
```

The `queryset` argument supplies a base `QuerySet` for the given lookup. This is useful to further filter down the prefetch operation, or to call *`select_related()`* from the prefetched relation, hence reducing the number of queries even further:

```
>>> voted_choices = Choice.objects.filter(votes__gt=0)
>>> voted_choices
<QuerySet [<Choice: The sky>]>
>>> prefetch = Prefetch('choice_set', queryset=voted_choices)
>>> Question.objects.prefetch_related(prefetch).get().choice_set.all()
<QuerySet [<Choice: The sky>]>
```

The `to_attr` argument sets the result of the prefetch operation to a custom attribute:

```
>>> prefetch = Prefetch('choice_set', queryset=voted_choices, to_attr='voted_choices')
>>> Question.objects.prefetch_related(prefetch).get().voted_choices
<QuerySet [<Choice: The sky>]>
>>> Question.objects.prefetch_related(prefetch).get().choice_set.all()
<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]>
```

: When using `to_attr` the prefetched result is stored in a list. This can provide a significant speed improvement over traditional `prefetch_related` calls which store the cached result within a `QuerySet` instance.

prefetch_related_objects()

prefetch_related_objects(*model_instances*, **related_lookups*)

Prefetches the given lookups on an iterable of model instances. This is useful in code that receives a list of model instances as opposed to a `QuerySet`; for example, when fetching models from a cache or instantiating them manually.

Pass an iterable of model instances (must all be of the same class) and the lookups or *`Prefetch`* objects you want to prefetch for. For example:

```
>>> from django.db.models import prefetch_related_objects
>>> restaurants = fetch_top_restaurants_from_cache()  # A list of Restaurants
>>> prefetch_related_objects(restaurants, 'pizzas__toppings')
```

6.15.9 Lookup API reference

This document has the API references of lookups, the Django API for building the `WHERE` clause of a database query. To learn how to *use* lookups, see Making queries; to learn how to *create* new lookups, see Custom Lookups.

The lookup API has two components: a *`RegisterLookupMixin`* class that registers lookups, and the *Query Expression API*, a set of methods that a class has to implement to be registrable as a lookup.

Django has two base classes that follow the query expression API and from where all Django builtin lookups are derived:

- *`Lookup`*: to lookup a field (e.g. the `exact` of `field_name__exact`)
- *`Transform`*: to transform a field

A lookup expression consists of three parts:

- Fields part (e.g. `Book.objects.filter(author__best_friends__first_name...`);
- Transforms part (may be omitted) (e.g. `__lower__first3chars__reversed`);
- A lookup (e.g. `__icontains`) that, if omitted, defaults to `__exact`.

Registration API

Django uses *`RegisterLookupMixin`* to give a class the interface to register lookups on itself. The two prominent examples are *`Field`*, the base class of all model fields, and `Aggregate`, the base class of all Django aggregates.

class `lookups`.**RegisterLookupMixin**

A mixin that implements the lookup API on a class.

classmethod register_lookup(*lookup*, *lookup_name=None*)

Registers a new lookup in the class. For example `DateField.register_lookup(YearExact)` will register `YearExact` lookup on `DateField`. It overrides a lookup that already exists with the same name. `lookup_name` will be used for this lookup if provided, otherwise `lookup.lookup_name` will be used.

The `lookup_name` parameter was added.

get_lookup(*lookup_name*)

Returns the *`Lookup`* named `lookup_name` registered in the class. The default implementation looks recursively on all parent classes and checks if any has a registered lookup named `lookup_name`, returning the first match.

get_transform(*transform_name*)
Returns a *Transform* named `transform_name`. The default implementation looks recursively on all parent classes to check if any has the registered transform named `transform_name`, returning the first match.

For a class to be a lookup, it must follow the *Query Expression API*. *Lookup* and *Transform* naturally follow this API.

The Query Expression API

The query expression API is a common set of methods that classes define to be usable in query expressions to translate themselves into SQL expressions. Direct field references, aggregates, and `Transform` are examples that follow this API. A class is said to follow the query expression API when it implements the following methods:

as_sql(*self*, *compiler*, *connection*)
Responsible for producing the query string and parameters for the expression. The `compiler` is an `SQLCompiler` object, which has a `compile()` method that can be used to compile other expressions. The `connection` is the connection used to execute the query.

Calling `expression.as_sql()` is usually incorrect - instead `compiler.compile(expression)` should be used. The `compiler.compile()` method will take care of calling vendor-specific methods of the expression.

Custom keyword arguments may be defined on this method if it's likely that `as_vendorname()` methods or subclasses will need to supply data to override the generation of the SQL string. See *Func.as_sql()* for example usage.

as_vendorname(*self*, *compiler*, *connection*)
Works like `as_sql()` method. When an expression is compiled by `compiler.compile()`, Django will first try to call `as_vendorname()`, where `vendorname` is the vendor name of the backend used for executing the query. The `vendorname` is one of `postgresql`, `oracle`, `sqlite`, or `mysql` for Django's built-in backends.

get_lookup(*lookup_name*)
Must return the lookup named `lookup_name`. For instance, by returning `self.output_field.get_lookup(lookup_name)`.

get_transform(*transform_name*)
Must return the lookup named `transform_name`. For instance, by returning `self.output_field.get_transform(transform_name)`.

output_field
Defines the type of class returned by the `get_lookup()` method. It must be a *Field* instance.

`Transform` reference

class Transform
A `Transform` is a generic class to implement field transformations. A prominent example is `__year` that transforms a `DateField` into a `IntegerField`.

The notation to use a `Transform` in an lookup expression is `<expression>__<transformation>` (e.g. `date__year`).

This class follows the *Query Expression API*, which implies that you can use `<expression>__<transform1>__<transform2>`. It's a specialized *Func() expression* that only accepts one argument. It can also be used on the right hand side of a filter or directly as an annotation.

`Transform` is now a subclass of `Func`.

bilateral
A boolean indicating whether this transformation should apply to both `lhs` and `rhs`. Bilateral transformations will be applied to `rhs` in the same order as they appear in the lookup expression. By default it is set to `False`. For example usage, see Custom Lookups.

lhs
The left-hand side - what is being transformed. It must follow the *Query Expression API*.

lookup_name
The name of the lookup, used for identifying it on parsing query expressions. It cannot contain the string `"__"`.

output_field
Defines the class this transformation outputs. It must be a *`Field`* instance. By default is the same as its `lhs.output_field`.

`Lookup` reference

class `Lookup`
A `Lookup` is a generic class to implement lookups. A lookup is a query expression with a left-hand side, *`lhs`*; a right-hand side, *`rhs`*; and a `lookup_name` that is used to produce a boolean comparison between `lhs` and `rhs` such as `lhs in rhs` or `lhs > rhs`.

The notation to use a lookup in an expression is `<lhs>__<lookup_name>=<rhs>`.

This class doesn't follow the *Query Expression API* since it has `=<rhs>` on its construction: lookups are always the end of a lookup expression.

lhs
The left-hand side - what is being looked up. The object must follow the *Query Expression API*.

rhs
The right-hand side - what `lhs` is being compared against. It can be a plain value, or something that compiles into SQL, typically an `F()` object or a `QuerySet`.

lookup_name
The name of this lookup, used to identify it on parsing query expressions. It cannot contain the string `"__"`.

process_lhs(*compiler*, *connection*, *lhs=None*)
Returns a tuple `(lhs_string, lhs_params)`, as returned by `compiler.compile(lhs)`. This method can be overridden to tune how the `lhs` is processed.

`compiler` is an `SQLCompiler` object, to be used like `compiler.compile(lhs)` for compiling `lhs`. The `connection` can be used for compiling vendor specific SQL. If `lhs` is not `None`, use it as the processed `lhs` instead of `self.lhs`.

process_rhs(*compiler*, *connection*)
Behaves the same way as *`process_lhs()`*, for the right-hand side.

6.15.10 Query Expressions

Query expressions describe a value or a computation that can be used as part of an update, create, filter, order by, annotation, or aggregate. There are a number of built-in expressions (documented below) that can be used to help you write queries. Expressions can be combined, or in some cases nested, to form more complex computations.

Support for using expressions when creating new model instances was added.

Supported arithmetic

Django supports addition, subtraction, multiplication, division, modulo arithmetic, and the power operator on query expressions, using Python constants, variables, and even other expressions.

Some examples

```
from django.db.models import F, Count
from django.db.models.functions import Length, Upper, Value

# Find companies that have more employees than chairs.
Company.objects.filter(num_employees__gt=F('num_chairs'))

# Find companies that have at least twice as many employees
# as chairs. Both the querysets below are equivalent.
Company.objects.filter(num_employees__gt=F('num_chairs') * 2)
Company.objects.filter(
    num_employees__gt=F('num_chairs') + F('num_chairs'))

# How many chairs are needed for each company to seat all employees?
>>> company = Company.objects.filter(
...     num_employees__gt=F('num_chairs')).annotate(
...     chairs_needed=F('num_employees') - F('num_chairs')).first()
>>> company.num_employees
120
>>> company.num_chairs
50
>>> company.chairs_needed
70

# Create a new company using expressions.
>>> company = Company.objects.create(name='Google', ticker=Upper(Value('goog')))
# Be sure to refresh it if you need to access the field.
>>> company.refresh_from_db()
>>> company.ticker
'GOOG'

# Annotate models with an aggregated value. Both forms
# below are equivalent.
Company.objects.annotate(num_products=Count('products'))
Company.objects.annotate(num_products=Count(F('products')))

# Aggregates can contain complex computations also
Company.objects.annotate(num_offerings=Count(F('products') + F('services')))

# Expressions can also be used in order_by()
Company.objects.order_by(Length('name').asc())
Company.objects.order_by(Length('name').desc())
```

Built-in Expressions

: These expressions are defined in `django.db.models.expressions` and `django.db.models.aggregates`, but for convenience they're available and usually imported from *`django.db.models`*.

F() expressions

class F

An F() object represents the value of a model field or annotated column. It makes it possible to refer to model field values and perform database operations using them without actually having to pull them out of the database into Python memory.

Instead, Django uses the F() object to generate an SQL expression that describes the required operation at the database level.

This is easiest to understand through an example. Normally, one might do something like this:

```
# Tintin filed a news story!
reporter = Reporters.objects.get(name='Tintin')
reporter.stories_filed += 1
reporter.save()
```

Here, we have pulled the value of reporter.stories_filed from the database into memory and manipulated it using familiar Python operators, and then saved the object back to the database. But instead we could also have done:

```
from django.db.models import F

reporter = Reporters.objects.get(name='Tintin')
reporter.stories_filed = F('stories_filed') + 1
reporter.save()
```

Although reporter.stories_filed = F('stories_filed') + 1 looks like a normal Python assignment of value to an instance attribute, in fact it's an SQL construct describing an operation on the database.

When Django encounters an instance of F(), it overrides the standard Python operators to create an encapsulated SQL expression; in this case, one which instructs the database to increment the database field represented by reporter.stories_filed.

Whatever value is or was on reporter.stories_filed, Python never gets to know about it - it is dealt with entirely by the database. All Python does, through Django's F() class, is create the SQL syntax to refer to the field and describe the operation.

: In order to access the new value that has been saved in this way, the object will need to be reloaded:

```
reporter = Reporters.objects.get(pk=reporter.pk)
# Or, more succinctly:
reporter.refresh_from_db()
```

As well as being used in operations on single instances as above, F() can be used on QuerySets of object instances, with update(). This reduces the two queries we were using above - the get() and the *save()* - to just one:

```
reporter = Reporters.objects.filter(name='Tintin')
reporter.update(stories_filed=F('stories_filed') + 1)
```

We can also use *update()* to increment the field value on multiple objects - which could be very much faster than pulling them all into Python from the database, looping over them, incrementing the field value of each one, and saving each one back to the database:

```
Reporter.objects.all().update(stories_filed=F('stories_filed') + 1)
```

F() therefore can offer performance advantages by:

- getting the database, rather than Python, to do work

- reducing the number of queries some operations require

Avoiding race conditions using `F()` Another useful benefit of `F()` is that having the database - rather than Python - update a field's value avoids a *race condition*.

If two Python threads execute the code in the first example above, one thread could retrieve, increment, and save a field's value after the other has retrieved it from the database. The value that the second thread saves will be based on the original value; the work of the first thread will simply be lost.

If the database is responsible for updating the field, the process is more robust: it will only ever update the field based on the value of the field in the database when the *`save()`* or `update()` is executed, rather than based on its value when the instance was retrieved.

`F()` assignments persist after `Model.save()` `F()` objects assigned to model fields persist after saving the model instance and will be applied on each *`save()`*. For example:

```
reporter = Reporters.objects.get(name='Tintin')
reporter.stories_filed = F('stories_filed') + 1
reporter.save()

reporter.name = 'Tintin Jr.'
reporter.save()
```

`stories_filed` will be updated twice in this case. If it's initially `1`, the final value will be `3`.

Using `F()` in filters `F()` is also very useful in `QuerySet` filters, where they make it possible to filter a set of objects against criteria based on their field values, rather than on Python values.

This is documented in *using F() expressions in queries*.

Using `F()` with annotations `F()` can be used to create dynamic fields on your models by combining different fields with arithmetic:

```
company = Company.objects.annotate(
    chairs_needed=F('num_employees') - F('num_chairs'))
```

If the fields that you're combining are of different types you'll need to tell Django what kind of field will be returned. Since `F()` does not directly support `output_field` you will need to wrap the expression with *`ExpressionWrapper`*:

```
from django.db.models import DateTimeField, ExpressionWrapper, F

Ticket.objects.annotate(
    expires=ExpressionWrapper(
        F('active_at') + F('duration'), output_field=DateTimeField()))
```

`Func()` expressions

`Func()` expressions are the base type of all expressions that involve database functions like `COALESCE` and `LOWER`, or aggregates like `SUM`. They can be used directly:

```
from django.db.models import Func, F

queryset.annotate(field_lower=Func(F('field'), function='LOWER'))
```

or they can be used to build a library of database functions:

```
class Lower(Func):
    function = 'LOWER'

queryset.annotate(field_lower=Lower('field'))
```

But both cases will result in a queryset where each model is annotated with an extra attribute `field_lower` produced, roughly, from the following SQL:

```
SELECT
    ...
    LOWER("db_table"."field") as "field_lower"
```

See Database Functions for a list of built-in database functions.

The `Func` API is as follows:

class Func(**expressions*, ***extra*)

function
A class attribute describing the function that will be generated. Specifically, the `function` will be interpolated as the `function` placeholder within *`template`*. Defaults to `None`.

template
A class attribute, as a format string, that describes the SQL that is generated for this function. Defaults to `'%(function)s(%(expressions)s)'`.

If you're constructing SQL like `strftime('%W', 'date')` and need a literal `%` character in the query, quadruple it (`%%%%`) in the `template` attribute because the string is interpolated twice: once during the template interpolation in `as_sql()` and once in the SQL interpolation with the query parameters in the database cursor.

arg_joiner
A class attribute that denotes the character used to join the list of `expressions` together. Defaults to `', '`.

arity
A class attribute that denotes the number of arguments the function accepts. If this attribute is set and the function is called with a different number of expressions, `TypeError` will be raised. Defaults to `None`.

as_sql(*compiler*, *connection*, *function=None*, *template=None*, *arg_joiner=None*, ***extra_context*)
Generates the SQL for the database function.

The `as_vendor()` methods should use the `function`, `template`, `arg_joiner`, and any other `**extra_context` parameters to customize the SQL as needed. For example:

django/db/models/functions.py

```
class ConcatPair(Func):
    ...
    function = 'CONCAT'
    ...

    def as_mysql(self, compiler, connection):
        return super(ConcatPair, self).as_sql(
            compiler, connection,
            function='CONCAT_WS',
            template="%(function)s('', %(expressions)s)",
        )
```

Support for the `arg_joiner` and `**extra_context` parameters was added.

The `*expressions` argument is a list of positional expressions that the function will be applied to. The expressions will be converted to strings, joined together with `arg_joiner`, and then interpolated into the `template` as the `expressions` placeholder.

Positional arguments can be expressions or Python values. Strings are assumed to be column references and will be wrapped in `F()` expressions while other values will be wrapped in `Value()` expressions.

The `**extra` kwargs are `key=value` pairs that can be interpolated into the `template` attribute. The `function`, `template`, and `arg_joiner` keywords can be used to replace the attributes of the same name without having to define your own class. `output_field` can be used to define the expected return type.

`Aggregate()` expressions

An aggregate expression is a special case of a *Func() expression* that informs the query that a `GROUP BY` clause is required. All of the *aggregate functions*, like `Sum()` and `Count()`, inherit from `Aggregate()`.

Since `Aggregate`s are expressions and wrap expressions, you can represent some complex computations:

```
from django.db.models import Count

Company.objects.annotate(
    managers_required=(Count('num_employees') / 4) + Count('num_managers'))
```

The `Aggregate` API is as follows:

class **Aggregate**(*expression*, *output_field=None*, ***extra*)

template
A class attribute, as a format string, that describes the SQL that is generated for this aggregate. Defaults to `'%(function)s( %(expressions)s )'`.

function
A class attribute describing the aggregate function that will be generated. Specifically, the `function` will be interpolated as the `function` placeholder within *`template`*. Defaults to `None`.

The `expression` argument can be the name of a field on the model, or another expression. It will be converted to a string and used as the `expressions` placeholder within the `template`.

The `output_field` argument requires a model field instance, like `IntegerField()` or `BooleanField()`, into which Django will load the value after it's retrieved from the database. Usually no arguments are needed when instantiating the model field as any arguments relating to data validation (`max_length`, `max_digits`, etc.) will not be enforced on the expression's output value.

Note that `output_field` is only required when Django is unable to determine what field type the result should be. Complex expressions that mix field types should define the desired `output_field`. For example, adding an `IntegerField()` and a `FloatField()` together should probably have `output_field=FloatField()` defined.

The `**extra` kwargs are `key=value` pairs that can be interpolated into the `template` attribute.

Creating your own Aggregate Functions

Creating your own aggregate is extremely easy. At a minimum, you need to define `function`, but you can also completely customize the SQL that is generated. Here's a brief example:

```
from django.db.models import Aggregate

class Count(Aggregate):
    # supports COUNT(distinct field)
    function = 'COUNT'
    template = '%(function)s(%(distinct)s%(expressions)s)'

    def __init__(self, expression, distinct=False, **extra):
        super(Count, self).__init__(
            expression,
            distinct='DISTINCT ' if distinct else '',
            output_field=IntegerField(),
            **extra
        )
```

`Value()` expressions

class `Value`(*value*, *output_field=None*)

A `Value()` object represents the smallest possible component of an expression: a simple value. When you need to represent the value of an integer, boolean, or string within an expression, you can wrap that value within a `Value()`.

You will rarely need to use `Value()` directly. When you write the expression `F('field') + 1`, Django implicitly wraps the `1` in a `Value()`, allowing simple values to be used in more complex expressions. You will need to use `Value()` when you want to pass a string to an expression. Most expressions interpret a string argument as the name of a field, like `Lower('name')`.

The `value` argument describes the value to be included in the expression, such as `1`, `True`, or `None`. Django knows how to convert these Python values into their corresponding database type.

The `output_field` argument should be a model field instance, like `IntegerField()` or `BooleanField()`, into which Django will load the value after it's retrieved from the database. Usually no arguments are needed when instantiating the model field as any arguments relating to data validation (`max_length`, `max_digits`, etc.) will not be enforced on the expression's output value.

`ExpressionWrapper()` expressions

class `ExpressionWrapper`(*expression*, *output_field*)

`ExpressionWrapper` simply surrounds another expression and provides access to properties, such as `output_field`, that may not be available on other expressions. `ExpressionWrapper` is necessary when using arithmetic on `F()` expressions with different types as described in *Using F() with annotations*.

Conditional expressions

Conditional expressions allow you to use `if` ... `elif` ... `else` logic in queries. Django natively supports SQL `CASE` expressions. For more details see Conditional Expressions.

Raw SQL expressions

class `RawSQL`(*sql*, *params*, *output_field=None*)

Sometimes database expressions can't easily express a complex `WHERE` clause. In these edge cases, use the `RawSQL` expression. For example:

```
>>> from django.db.models.expressions import RawSQL
>>> queryset.annotate(val=RawSQL("select col from sometable where othercol = %s", (someparam,)))
```

These extra lookups may not be portable to different database engines (because you're explicitly writing SQL code) and violate the DRY principle, so you should avoid them if possible.

: You should be very careful to escape any parameters that the user can control by using `params` in order to protect against *SQL injection attacks*. `params` is a required argument to force you to acknowledge that you're not interpolating your SQL with user provided data.

Technical Information

Below you'll find technical implementation details that may be useful to library authors. The technical API and examples below will help with creating generic query expressions that can extend the built-in functionality that Django provides.

Expression API

Query expressions implement the *query expression API*, but also expose a number of extra methods and attributes listed below. All query expressions must inherit from `Expression()` or a relevant subclass.

When a query expression wraps another expression, it is responsible for calling the appropriate methods on the wrapped expression.

class `Expression`

`contains_aggregate`
Tells Django that this expression contains an aggregate and that a `GROUP BY` clause needs to be added to the query.

`resolve_expression`(*query=None*, *allow_joins=True*, *reuse=None*, *summarize=False*, *for_save=False*)
Provides the chance to do any pre-processing or validation of the expression before it's added to the query. `resolve_expression()` must also be called on any nested expressions. A `copy()` of `self` should be returned with any necessary transformations.

`query` is the backend query implementation.

`allow_joins` is a boolean that allows or denies the use of joins in the query.

`reuse` is a set of reusable joins for multi-join scenarios.

`summarize` is a boolean that, when `True`, signals that the query being computed is a terminal aggregate query.

`get_source_expressions`()
Returns an ordered list of inner expressions. For example:

```
>>> Sum(F('foo')).get_source_expressions()
[F('foo')]
```

`set_source_expressions`(*expressions*)
Takes a list of expressions and stores them such that `get_source_expressions()` can return them.

relabeled_clone(*change_map*)
Returns a clone (copy) of `self`, with any column aliases relabeled. Column aliases are renamed when subqueries are created. `relabeled_clone()` should also be called on any nested expressions and assigned to the clone.

`change_map` is a dictionary mapping old aliases to new aliases.

Example:

```
def relabeled_clone(self, change_map):
    clone = copy.copy(self)
    clone.expression = self.expression.relabeled_clone(change_map)
    return clone
```

convert_value(*self*, *value*, *expression*, *connection*, *context*)
A hook allowing the expression to coerce `value` into a more appropriate type.

get_group_by_cols()
Responsible for returning the list of columns references by this expression. `get_group_by_cols()` should be called on any nested expressions. `F()` objects, in particular, hold a reference to a column.

asc()
Returns the expression ready to be sorted in ascending order.

desc()
Returns the expression ready to be sorted in descending order.

reverse_ordering()
Returns `self` with any modifications required to reverse the sort order within an `order_by` call. As an example, an expression implementing `NULLS LAST` would change its value to be `NULLS FIRST`. Modifications are only required for expressions that implement sort order like `OrderBy`. This method is called when *`reverse()`* is called on a queryset.

Writing your own Query Expressions

You can write your own query expression classes that use, and can integrate with, other query expressions. Let's step through an example by writing an implementation of the `COALESCE` SQL function, without using the built-in *Func() expressions*.

The `COALESCE` SQL function is defined as taking a list of columns or values. It will return the first column or value that isn't `NULL`.

We'll start by defining the template to be used for SQL generation and an `__init__()` method to set some attributes:

```
import copy
from django.db.models import Expression

class Coalesce(Expression):
    template = 'COALESCE( %(expressions)s )'

    def __init__(self, expressions, output_field):
      super(Coalesce, self).__init__(output_field=output_field)
      if len(expressions) < 2:
          raise ValueError('expressions must have at least 2 elements')
      for expression in expressions:
          if not hasattr(expression, 'resolve_expression'):
              raise TypeError('%r is not an Expression' % expression)
      self.expressions = expressions
```

We do some basic validation on the parameters, including requiring at least 2 columns or values, and ensuring they are expressions. We are requiring `output_field` here so that Django knows what kind of model field to assign the eventual result to.

Now we implement the pre-processing and validation. Since we do not have any of our own validation at this point, we just delegate to the nested expressions:

```
def resolve_expression(self, query=None, allow_joins=True, reuse=None, summarize=False, for_save=Fals
    c = self.copy()
    c.is_summary = summarize
    for pos, expression in enumerate(self.expressions):
        c.expressions[pos] = expression.resolve_expression(query, allow_joins, reuse, summarize, for_
    return c
```

Next, we write the method responsible for generating the SQL:

```
def as_sql(self, compiler, connection, template=None):
    sql_expressions, sql_params = [], []
    for expression in self.expressions:
        sql, params = compiler.compile(expression)
        sql_expressions.append(sql)
        sql_params.extend(params)
    template = template or self.template
    data = {'expressions': ','.join(sql_expressions)}
    return template % data, params

def as_oracle(self, compiler, connection):
    """
    Example of vendor specific handling (Oracle in this case).
    Let's make the function name lowercase.
    """
    return self.as_sql(compiler, connection, template='coalesce( %(expressions)s )')
```

`as_sql()` methods can support custom keyword arguments, allowing `as_vendorname()` methods to override data used to generate the SQL string. Using `as_sql()` keyword arguments for customization is preferable to mutating `self` within `as_vendorname()` methods as the latter can lead to errors when running on different database backends. If your class relies on class attributes to define data, consider allowing overrides in your `as_sql()` method.

We generate the SQL for each of the `expressions` by using the `compiler.compile()` method, and join the result together with commas. Then the template is filled out with our data and the SQL and parameters are returned.

We've also defined a custom implementation that is specific to the Oracle backend. The `as_oracle()` function will be called instead of `as_sql()` if the Oracle backend is in use.

Finally, we implement the rest of the methods that allow our query expression to play nice with other query expressions:

```
def get_source_expressions(self):
    return self.expressions

def set_source_expressions(self, expressions):
    self.expressions = expressions
```

Let's see how it works:

```
>>> from django.db.models import F, Value, CharField
>>> qs = Company.objects.annotate(
...    tagline=Coalesce([
...        F('motto'),
...        F('ticker_name'),
...        F('description'),
...        Value('No Tagline')
```

```
...         ], output_field=CharField()))
>>> for c in qs:
...     print("%s: %s" % (c.name, c.tagline))
...
Google: Do No Evil
Apple: AAPL
Yahoo: Internet Company
Django Software Foundation: No Tagline
```

Adding support in third-party database backends

If you're using a database backend that uses a different SQL syntax for a certain function, you can add support for it by monkey patching a new method onto the function's class.

Let's say we're writing a backend for Microsoft's SQL Server which uses the SQL `LEN` instead of `LENGTH` for the *`Length`* function. We'll monkey patch a new method called `as_sqlserver()` onto the `Length` class:

```
from django.db.models.functions import Length

def sqlserver_length(self, compiler, connection):
    return self.as_sql(compiler, connection, function='LEN')

Length.as_sqlserver = sqlserver_length
```

You can also customize the SQL using the `template` parameter of `as_sql()`.

We use `as_sqlserver()` because `django.db.connection.vendor` returns `sqlserver` for the backend.

Third-party backends can register their functions in the top level `__init__.py` file of the backend package or in a top level `expressions.py` file (or package) that is imported from the top level `__init__.py`.

For user projects wishing to patch the backend that they're using, this code should live in an *`AppConfig.ready()`* method.

6.15.11 Conditional Expressions

Conditional expressions let you use `if` ... `elif` ... `else` logic within filters, annotations, aggregations, and updates. A conditional expression evaluates a series of conditions for each row of a table and returns the matching result expression. Conditional expressions can also be combined and nested like other expressions.

The conditional expression classes

We'll be using the following model in the subsequent examples:

```
from django.db import models

class Client(models.Model):
    REGULAR = 'R'
    GOLD = 'G'
    PLATINUM = 'P'
    ACCOUNT_TYPE_CHOICES = (
        (REGULAR, 'Regular'),
        (GOLD, 'Gold'),
        (PLATINUM, 'Platinum'),
    )
```

```
    name = models.CharField(max_length=50)
    registered_on = models.DateField()
    account_type = models.CharField(
        max_length=1,
        choices=ACCOUNT_TYPE_CHOICES,
        default=REGULAR,
    )
```

When

class When (*condition=None*, *then=None*, ***lookups*)

A `When()` object is used to encapsulate a condition and its result for use in the conditional expression. Using a `When()` object is similar to using the *`filter()`* method. The condition can be specified using *field lookups* or *`Q`* objects. The result is provided using the `then` keyword.

Some examples:

```
>>> from django.db.models import When, F, Q
>>> # String arguments refer to fields; the following two examples are equivalent:
>>> When(account_type=Client.GOLD, then='name')
>>> When(account_type=Client.GOLD, then=F('name'))
>>> # You can use field lookups in the condition
>>> from datetime import date
>>> When(registered_on__gt=date(2014, 1, 1),
...      registered_on__lt=date(2015, 1, 1),
...      then='account_type')
>>> # Complex conditions can be created using Q objects
>>> When(Q(name__startswith="John") | Q(name__startswith="Paul"),
...      then='name')
```

Keep in mind that each of these values can be an expression.

: Since the `then` keyword argument is reserved for the result of the `When()`, there is a potential conflict if a *`Model`* has a field named `then`. This can be resolved in two ways:

```
>>> When(then__exact=0, then=1)
>>> When(Q(then=0), then=1)
```

Case

class Case (**cases*, ***extra*)

A `Case()` expression is like the `if ... elif ... else` statement in `Python`. Each `condition` in the provided `When()` objects is evaluated in order, until one evaluates to a truthful value. The `result` expression from the matching `When()` object is returned.

A simple example:

```
>>>
>>> from datetime import date, timedelta
>>> from django.db.models import CharField, Case, Value, When
>>> Client.objects.create(
...     name='Jane Doe',
...     account_type=Client.REGULAR,
```

```
...     registered_on=date.today() - timedelta(days=36))
>>> Client.objects.create(
...     name='James Smith',
...     account_type=Client.GOLD,
...     registered_on=date.today() - timedelta(days=5))
>>> Client.objects.create(
...     name='Jack Black',
...     account_type=Client.PLATINUM,
...     registered_on=date.today() - timedelta(days=10 * 365))
>>> # Get the discount for each Client based on the account type
>>> Client.objects.annotate(
...     discount=Case(
...         When(account_type=Client.GOLD, then=Value('5%')),
...         When(account_type=Client.PLATINUM, then=Value('10%')),
...         default=Value('0%'),
...         output_field=CharField(),
...     ),
... ).values_list('name', 'discount')
[('Jane Doe', '0%'), ('James Smith', '5%'), ('Jack Black', '10%')]
```

`Case()` accepts any number of `When()` objects as individual arguments. Other options are provided using keyword arguments. If none of the conditions evaluate to `TRUE`, then the expression given with the `default` keyword argument is returned. If a `default` argument isn't provided, `None` is used.

If we wanted to change our previous query to get the discount based on how long the `Client` has been with us, we could do so using lookups:

```
>>> a_month_ago = date.today() - timedelta(days=30)
>>> a_year_ago = date.today() - timedelta(days=365)
>>> # Get the discount for each Client based on the registration date
>>> Client.objects.annotate(
...     discount=Case(
...         When(registered_on__lte=a_year_ago, then=Value('10%')),
...         When(registered_on__lte=a_month_ago, then=Value('5%')),
...         default=Value('0%'),
...         output_field=CharField(),
...     )
... ).values_list('name', 'discount')
[('Jane Doe', '5%'), ('James Smith', '0%'), ('Jack Black', '10%')]
```

: Remember that the conditions are evaluated in order, so in the above example we get the correct result even though the second condition matches both Jane Doe and Jack Black. This works just like an `if` ... `elif` ... `else` statement in `Python`.

Advanced queries

Conditional expressions can be used in annotations, aggregations, lookups, and updates. They can also be combined and nested with other expressions. This allows you to make powerful conditional queries.

Conditional update

Let's say we want to change the `account_type` for our clients to match their registration dates. We can do this using a conditional expression and the *`update()`* method:

```
>>> a_month_ago = date.today() - timedelta(days=30)
>>> a_year_ago = date.today() - timedelta(days=365)
>>> # Update the account_type for each Client from the registration date
>>> Client.objects.update(
...     account_type=Case(
...         When(registered_on__lte=a_year_ago,
...              then=Value(Client.PLATINUM)),
...         When(registered_on__lte=a_month_ago,
...              then=Value(Client.GOLD)),
...         default=Value(Client.REGULAR)
...     ),
... )
>>> Client.objects.values_list('name', 'account_type')
[('Jane Doe', 'G'), ('James Smith', 'R'), ('Jack Black', 'P')]
```

Conditional aggregation

What if we want to find out how many clients there are for each `account_type`? We can nest conditional expression within *aggregate functions* to achieve this:

```
>>> # Create some more Clients first so we can have something to count
>>> Client.objects.create(
...     name='Jean Grey',
...     account_type=Client.REGULAR,
...     registered_on=date.today())
>>> Client.objects.create(
...     name='James Bond',
...     account_type=Client.PLATINUM,
...     registered_on=date.today())
>>> Client.objects.create(
...     name='Jane Porter',
...     account_type=Client.PLATINUM,
...     registered_on=date.today())
>>> # Get counts for each value of account_type
>>> from django.db.models import IntegerField, Sum
>>> Client.objects.aggregate(
...     regular=Sum(
...         Case(When(account_type=Client.REGULAR, then=1),
...              output_field=IntegerField())
...     ),
...     gold=Sum(
...         Case(When(account_type=Client.GOLD, then=1),
...              output_field=IntegerField())
...     ),
...     platinum=Sum(
...         Case(When(account_type=Client.PLATINUM, then=1),
...              output_field=IntegerField())
...     )
... )
{'regular': 2, 'gold': 1, 'platinum': 3}
```

6.15.12 Database Functions

The classes documented below provide a way for users to use functions provided by the underlying database as annotations, aggregations, or filters in Django. Functions are also expressions, so they can be used and combined with

other expressions like *aggregate functions*.

We'll be using the following model in examples of each function:

```
class Author(models.Model):
    name = models.CharField(max_length=50)
    age = models.PositiveIntegerField(null=True, blank=True)
    alias = models.CharField(max_length=50, null=True, blank=True)
    goes_by = models.CharField(max_length=50, null=True, blank=True)
```

We don't usually recommend allowing `null=True` for `CharField` since this allows the field to have two "empty values", but it's important for the `Coalesce` example below.

Cast

class Cast(*expression*, *output_field*)

Forces the result type of `expression` to be the one from `output_field`.

Usage example:

```
>>> from django.db.models import FloatField
>>> from django.db.models.functions import Cast
>>> Value.objects.create(integer=4)
>>> value = Value.objects.annotate(as_float=Cast('integer', FloatField())).get()
>>> print(value.as_float)
4.0
```

Coalesce

class Coalesce(**expressions*, ***extra*)

Accepts a list of at least two field names or expressions and returns the first non-null value (note that an empty string is not considered a null value). Each argument must be of a similar type, so mixing text and numbers will result in a database error.

Usage examples:

```
>>> # Get a screen name from least to most public
>>> from django.db.models import Sum, Value as V
>>> from django.db.models.functions import Coalesce
>>> Author.objects.create(name='Margaret Smith', goes_by='Maggie')
>>> author = Author.objects.annotate(
...     screen_name=Coalesce('alias', 'goes_by', 'name')).get()
>>> print(author.screen_name)
Maggie

>>> # Prevent an aggregate Sum() from returning None
>>> aggregated = Author.objects.aggregate(
...     combined_age=Coalesce(Sum('age'), V(0)),
...     combined_age_default=Sum('age'))
>>> print(aggregated['combined_age'])
0
>>> print(aggregated['combined_age_default'])
None
```

: A Python value passed to `Coalesce` on MySQL may be converted to an incorrect type unless explicitly cast to the correct database type:

```
>>> from django.db.models.expressions import RawSQL
>>> from django.utils import timezone
>>> now = timezone.now()
>>> now_sql = RawSQL("cast(%s as datetime)", (now,))
>>> Coalesce('updated', now_sql)
```

Concat

class Concat (**expressions*, ***extra*)

Accepts a list of at least two text fields or expressions and returns the concatenated text. Each argument must be of a text or char type. If you want to concatenate a `TextField()` with a `CharField()`, then be sure to tell Django that the `output_field` should be a `TextField()`. This is also required when concatenating a `Value` as in the example below.

This function will never have a null result. On backends where a null argument results in the entire expression being null, Django will ensure that each null part is converted to an empty string first.

Usage example:

```
>>> # Get the display name as "name (goes_by)"
>>> from django.db.models import CharField, Value as V
>>> from django.db.models.functions import Concat
>>> Author.objects.create(name='Margaret Smith', goes_by='Maggie')
>>> author = Author.objects.annotate(
...     screen_name=Concat('name', V(' ('), 'goes_by', V(')'),
...     output_field=CharField())).get()
>>> print(author.screen_name)
Margaret Smith (Maggie)
```

Greatest

class Greatest (**expressions*, ***extra*)

Accepts a list of at least two field names or expressions and returns the greatest value. Each argument must be of a similar type, so mixing text and numbers will result in a database error.

Usage example:

```
class Blog(models.Model):
    body = models.TextField()
    modified = models.DateTimeField(auto_now=True)

class Comment(models.Model):
    body = models.TextField()
    modified = models.DateTimeField(auto_now=True)
    blog = models.ForeignKey(Blog, on_delete=models.CASCADE)

>>> from django.db.models.functions import Greatest
>>> blog = Blog.objects.create(body='Greatest is the best.')
>>> comment = Comment.objects.create(body='No, Least is better.', blog=blog)
>>> comments = Comment.objects.annotate(last_updated=Greatest('modified', 'blog__modified'))
>>> annotated_comment = comments.get()
```

`annotated_comment.last_updated` will be the most recent of `blog.modified` and `comment.modified`.

: The behavior of `Greatest` when one or more expression may be `null` varies between databases:
- PostgreSQL: `Greatest` will return the largest non-null expression, or `null` if all expressions are `null`.
- SQLite, Oracle, and MySQL: If any expression is `null`, `Greatest` will return `null`.

The PostgreSQL behavior can be emulated using `Coalesce` if you know a sensible minimum value to provide as a default.

Least

class Least(**expressions*, ***extra*)

Accepts a list of at least two field names or expressions and returns the least value. Each argument must be of a similar type, so mixing text and numbers will result in a database error.

: The behavior of `Least` when one or more expression may be `null` varies between databases:
- PostgreSQL: `Least` will return the smallest non-null expression, or `null` if all expressions are `null`.
- SQLite, Oracle, and MySQL: If any expression is `null`, `Least` will return `null`.

The PostgreSQL behavior can be emulated using `Coalesce` if you know a sensible maximum value to provide as a default.

Length

class Length(*expression*, ***extra*)

Accepts a single text field or expression and returns the number of characters the value has. If the expression is null, then the length will also be null.

Usage example:

```
>>> # Get the length of the name and goes_by fields
>>> from django.db.models.functions import Length
>>> Author.objects.create(name='Margaret Smith')
>>> author = Author.objects.annotate(
...     name_length=Length('name'),
...     goes_by_length=Length('goes_by')).get()
>>> print(author.name_length, author.goes_by_length)
(14, None)
```

It can also be registered as a transform. For example:

```
>>> from django.db.models import CharField
>>> from django.db.models.functions import Length
>>> CharField.register_lookup(Length, 'length')
>>> # Get authors whose name is longer than 7 characters
>>> authors = Author.objects.filter(name__length__gt=7)
```

The ability to register the function as a transform was added.

Lower

class Lower(*expression*, ***extra*)

Accepts a single text field or expression and returns the lowercase representation.

It can also be registered as a transform as described in *Length*.

Usage example:

```
>>> from django.db.models.functions import Lower
>>> Author.objects.create(name='Margaret Smith')
>>> author = Author.objects.annotate(name_lower=Lower('name')).get()
>>> print(author.name_lower)
margaret smith
```

The ability to register the function as a transform was added.

Now

class Now

Returns the database server's current date and time when the query is executed, typically using the SQL `CURRENT_TIMESTAMP`.

Usage example:

```
>>> from django.db.models.functions import Now
>>> Article.objects.filter(published__lte=Now())
<QuerySet [<Article: How to Django>]>
```

PostgreSQL considerations

On PostgreSQL, the SQL `CURRENT_TIMESTAMP` returns the time that the current transaction started. Therefore for cross-database compatibility, `Now()` uses `STATEMENT_TIMESTAMP` instead. If you need the transaction timestamp, use *django.contrib.postgres.functions.TransactionNow*.

Substr

class Substr(*expression*, *pos*, *length=None*, ***extra*)

Returns a substring of length `length` from the field or expression starting at position `pos`. The position is 1-indexed, so the position must be greater than 0. If `length` is `None`, then the rest of the string will be returned.

Usage example:

```
>>> # Set the alias to the first 5 characters of the name as lowercase
>>> from django.db.models.functions import Substr, Lower
>>> Author.objects.create(name='Margaret Smith')
>>> Author.objects.update(alias=Lower(Substr('name', 1, 5)))
1
>>> print(Author.objects.get(name='Margaret Smith').alias)
marga
```

Upper

class Upper(*expression*, ***extra*)

Accepts a single text field or expression and returns the uppercase representation.

It can also be registered as a transform as described in *Length*.

Usage example:

```
>>> from django.db.models.functions import Upper
>>> Author.objects.create(name='Margaret Smith')
>>> author = Author.objects.annotate(name_upper=Upper('name')).get()
>>> print(author.name_upper)
MARGARET SMITH
```

The ability to register the function as a transform was added.

Date Functions

We'll be using the following model in examples of each function:

```
class Experiment(models.Model):
    start_datetime = models.DateTimeField()
    start_date = models.DateField(null=True, blank=True)
    end_datetime = models.DateTimeField(null=True, blank=True)
    end_date = models.DateField(null=True, blank=True)
```

Extract

class Extract(*expression*, *lookup_name=None*, *tzinfo=None*, ***extra*)

Extracts a component of a date as a number.

Takes an `expression` representing a `DateField` or `DateTimeField` and a `lookup_name`, and returns the part of the date referenced by `lookup_name` as an `IntegerField`. Django usually uses the databases' extract function, so you may use any `lookup_name` that your database supports. A `tzinfo` subclass, usually provided by `pytz`, can be passed to extract a value in a specific timezone.

Given the datetime `2015-06-15 23:30:01.000321+00:00`, the built-in `lookup_names` return:

- "year": 2015
- "month": 6
- "day": 15
- "week_day": 2
- "hour": 23
- "minute": 30
- "second": 1

If a different timezone like `Australia/Melbourne` is active in Django, then the datetime is converted to the timezone before the value is extracted. The timezone offset for Melbourne in the example date above is +10:00. The values returned when this timezone is active will be the same as above except for:

- "day": 16
- "week_day": 3
- "hour": 9

week_day values

The `week_day` `lookup_type` is calculated differently from most databases and from Python's standard functions. This function will return `1` for Sunday, `2` for Monday, through `7` for Saturday.

The equivalent calculation in Python is:

```
>>> from datetime import datetime
>>> dt = datetime(2015, 6, 15)
>>> (dt.isoweekday() % 7) + 1
2
```

Each `lookup_name` above has a corresponding `Extract` subclass (listed below) that should typically be used instead of the more verbose equivalent, e.g. use `ExtractYear(...)` rather than `Extract(..., lookup_name='year')`.

Usage example:

```
>>> from datetime import datetime
>>> from django.db.models.functions import Extract
>>> start = datetime(2015, 6, 15)
>>> end = datetime(2015, 7, 2)
>>> Experiment.objects.create(
...     start_datetime=start, start_date=start.date(),
...     end_datetime=end, end_date=end.date())
>>> # Add the experiment start year as a field in the QuerySet.
>>> experiment = Experiment.objects.annotate(
...     start_year=Extract('start_datetime', 'year')).get()
>>> experiment.start_year
2015
>>> # How many experiments completed in the same year in which they started?
>>> Experiment.objects.filter(
...     start_datetime__year=Extract('end_datetime', 'year')).count()
1
```

DateField extracts

class ExtractYear(*expression*, *tzinfo=None*, ***extra*)

lookup_name = 'year'

class ExtractMonth(*expression*, *tzinfo=None*, ***extra*)

lookup_name = 'month'

class ExtractDay(*expression*, *tzinfo=None*, ***extra*)

lookup_name = 'day'

class ExtractWeekDay(*expression*, *tzinfo=None*, ***extra*)

lookup_name = 'week_day'

These are logically equivalent to `Extract('date_field', lookup_name)`. Each class is also a `Transform` registered on `DateField` and `DateTimeField` as `__(lookup_name)`, e.g. `__year`.

Since DateFields don't have a time component, only Extract subclasses that deal with date-parts can be used with DateField:

```
>>> from datetime import datetime
>>> from django.utils import timezone
>>> from django.db.models.functions import (
...     ExtractYear, ExtractMonth, ExtractDay, ExtractWeekDay
... )
>>> start_2015 = datetime(2015, 6, 15, 23, 30, 1, tzinfo=timezone.utc)
>>> end_2015 = datetime(2015, 6, 16, 13, 11, 27, tzinfo=timezone.utc)
>>> Experiment.objects.create(
...     start_datetime=start_2015, start_date=start_2015.date(),
...     end_datetime=end_2015, end_date=end_2015.date())
>>> Experiment.objects.annotate(
...      year=ExtractYear('start_date'),
...      month=ExtractMonth('start_date'),
...      day=ExtractDay('start_date'),
...      weekday=ExtractWeekDay('start_date'),
... ).values('year', 'month', 'day', 'weekday').get(
...      end_date__year=ExtractYear('start_date'),
... )
{'year': 2015, 'month': 6, 'day': 15, 'weekday': 2}
```

DateTimeField extracts In addition to the following, all extracts for DateField listed above may also be used on DateTimeFields .

class ExtractHour(*expression*, *tzinfo=None*, ***extra*)

lookup_name = 'hour'

class ExtractMinute(*expression*, *tzinfo=None*, ***extra*)

lookup_name = 'minute'

class ExtractSecond(*expression*, *tzinfo=None*, ***extra*)

lookup_name = 'second'

These are logically equivalent to Extract('datetime_field', lookup_name). Each class is also a Transform registered on DateTimeField as __(lookup_name), e.g. __minute.

DateTimeField examples:

```
>>> from datetime import datetime
>>> from django.utils import timezone
>>> from django.db.models.functions import (
...     ExtractYear, ExtractMonth, ExtractDay, ExtractWeekDay,
...     ExtractHour, ExtractMinute, ExtractSecond,
... )
>>> start_2015 = datetime(2015, 6, 15, 23, 30, 1, tzinfo=timezone.utc)
>>> end_2015 = datetime(2015, 6, 16, 13, 11, 27, tzinfo=timezone.utc)
>>> Experiment.objects.create(
...     start_datetime=start_2015, start_date=start_2015.date(),
...     end_datetime=end_2015, end_date=end_2015.date())
>>> Experiment.objects.annotate(
...      year=ExtractYear('start_datetime'),
...      month=ExtractMonth('start_datetime'),
```

```
...     day=ExtractDay('start_datetime'),
...     weekday=ExtractWeekDay('start_datetime'),
...     hour=ExtractHour('start_datetime'),
...     minute=ExtractMinute('start_datetime'),
...     second=ExtractSecond('start_datetime'),
... ).values(
...     'year', 'month', 'day', 'weekday', 'hour', 'minute', 'second',
... ).get(end_datetime__year=ExtractYear('start_datetime'))
{'year': 2015, 'month': 6, 'day': 15, 'weekday': 2, 'hour': 23, 'minute': 30, 'second': 1}
```

When *USE_TZ* is `True` then datetimes are stored in the database in UTC. If a different timezone is active in Django, the datetime is converted to that timezone before the value is extracted. The example below converts to the Melbourne timezone (UTC +10:00), which changes the day, weekday, and hour values that are returned:

```
>>> import pytz
>>> tzinfo = pytz.timezone('Australia/Melbourne')  # UTC+10:00
>>> with timezone.override(tzinfo):
...     Experiment.objects.annotate(
...         day=ExtractDay('start_datetime'),
...         weekday=ExtractWeekDay('start_datetime'),
...         hour=ExtractHour('start_datetime'),
...     ).values('day', 'weekday', 'hour').get(
...         end_datetime__year=ExtractYear('start_datetime'),
...     )
{'day': 16, 'weekday': 3, 'hour': 9}
```

Explicitly passing the timezone to the `Extract` function behaves in the same way, and takes priority over an active timezone:

```
>>> import pytz
>>> tzinfo = pytz.timezone('Australia/Melbourne')
>>> Experiment.objects.annotate(
...     day=ExtractDay('start_datetime', tzinfo=melb),
...     weekday=ExtractWeekDay('start_datetime', tzinfo=melb),
...     hour=ExtractHour('start_datetime', tzinfo=melb),
... ).values('day', 'weekday', 'hour').get(
...     end_datetime__year=ExtractYear('start_datetime'),
... )
{'day': 16, 'weekday': 3, 'hour': 9}
```

Trunc

class Trunc(*expression*, *kind*, *output_field=None*, *tzinfo=None*, ***extra*)

Truncates a date up to a significant component.

When you only care if something happened in a particular year, hour, or day, but not the exact second, then `Trunc` (and its subclasses) can be useful to filter or aggregate your data. For example, you can use `Trunc` to calculate the number of sales per day.

`Trunc` takes a single `expression`, representing a `DateField` or `DateTimeField`, a `kind` representing a date part, and an `output_field` that's either `DateTimeField()` or `DateField()`. It returns a datetime or date, depending on `output_field`, with fields up to `kind` set to their minimum value. If `output_field` is omitted, it will default to the `output_field` of `expression`. A `tzinfo` subclass, usually provided by `pytz`, can be passed to truncate a value in a specific timezone.

Given the datetime `2015-06-15 14:30:50.000321+00:00`, the built-in `kinds` return:

- “year”: 2015-01-01 00:00:00+00:00
- “month”: 2015-06-01 00:00:00+00:00
- “day”: 2015-06-15 00:00:00+00:00
- “hour”: 2015-06-15 14:00:00+00:00
- “minute”: 2015-06-15 14:30:00+00:00
- “second”: 2015-06-15 14:30:50+00:00

If a different timezone like `Australia/Melbourne` is active in Django, then the datetime is converted to the new timezone before the value is truncated. The timezone offset for Melbourne in the example date above is +10:00. The values returned when this timezone is active will be:

- “year”: 2015-01-01 00:00:00+11:00
- “month”: 2015-06-01 00:00:00+10:00
- “day”: 2015-06-16 00:00:00+10:00
- “hour”: 2015-06-16 00:00:00+10:00
- “minute”: 2015-06-16 00:30:00+10:00
- “second”: 2015-06-16 00:30:50+10:00

The year has an offset of +11:00 because the result transitioned into daylight saving time.

Each `kind` above has a corresponding `Trunc` subclass (listed below) that should typically be used instead of the more verbose equivalent, e.g. use `TruncYear(...)` rather than `Trunc(..., kind='year')`.

The subclasses are all defined as transforms, but they aren't registered with any fields, because the obvious lookup names are already reserved by the `Extract` subclasses.

Usage example:

```
>>> from datetime import datetime
>>> from django.db.models import Count, DateTimeField
>>> from django.db.models.functions import Trunc
>>> Experiment.objects.create(start_datetime=datetime(2015, 6, 15, 14, 30, 50, 321))
>>> Experiment.objects.create(start_datetime=datetime(2015, 6, 15, 14, 40, 2, 123))
>>> Experiment.objects.create(start_datetime=datetime(2015, 12, 25, 10, 5, 27, 999))
>>> experiments_per_day = Experiment.objects.annotate(
...     start_day=Trunc('start_datetime', 'day', output_field=DateTimeField())
... ).values('start_day').annotate(experiments=Count('id'))
>>> for exp in experiments_per_day:
...     print(exp['start_day'], exp['experiments'])
...
2015-06-15 00:00:00 2
2015-12-25 00:00:00 1
>>> experiments = Experiment.objects.annotate(
...     start_day=Trunc('start_datetime', 'day', output_field=DateTimeField())
... ).filter(start_day=datetime(2015, 6, 15))
>>> for exp in experiments:
...     print(exp.start_datetime)
...
2015-06-15 14:30:50.000321
2015-06-15 14:40:02.000123
```

`DateField` truncation

class TruncYear(*expression*, *output_field=None*, *tzinfo=None*, ***extra*)

kind = 'year'

class TruncMonth(*expression*, *output_field=None*, *tzinfo=None*, ***extra*)

kind = 'month'

These are logically equivalent to `Trunc('date_field', kind)`. They truncate all parts of the date up to `kind` which allows grouping or filtering dates with less precision. `expression` can have an `output_field` of either `DateField` or `DateTimeField`.

Since `DateField`s don't have a time component, only `Trunc` subclasses that deal with date-parts can be used with `DateField`:

```
>>> from datetime import datetime
>>> from django.db.models import Count
>>> from django.db.models.functions import TruncMonth, TruncYear
>>> from django.utils import timezone
>>> start1 = datetime(2014, 6, 15, 14, 30, 50, 321, tzinfo=timezone.utc)
>>> start2 = datetime(2015, 6, 15, 14, 40, 2, 123, tzinfo=timezone.utc)
>>> start3 = datetime(2015, 12, 31, 17, 5, 27, 999, tzinfo=timezone.utc)
>>> Experiment.objects.create(start_datetime=start1, start_date=start1.date())
>>> Experiment.objects.create(start_datetime=start2, start_date=start2.date())
>>> Experiment.objects.create(start_datetime=start3, start_date=start3.date())
>>> experiments_per_year = Experiment.objects.annotate(
...     year=TruncYear('start_date')).values('year').annotate(
...     experiments=Count('id'))
>>> for exp in experiments_per_year:
...     print(exp['year'], exp['experiments'])
...
2014-01-01 1
2015-01-01 2

>>> import pytz
>>> melb = pytz.timezone('Australia/Melbourne')
>>> experiments_per_month = Experiment.objects.annotate(
...     month=TruncMonth('start_datetime', tzinfo=melb)).values('month').annotate(
...     experiments=Count('id'))
>>> for exp in experiments_per_month:
...     print(exp['month'], exp['experiments'])
...
2015-06-01 00:00:00+10:00 1
2016-01-01 00:00:00+11:00 1
2014-06-01 00:00:00+10:00 1
```

DateTimeField truncation

class TruncDate(*expression*, ***extra*)

lookup_name = 'date'

output_field = DateField()

`TruncDate` casts `expression` to a date rather than using the built-in SQL truncate function. It's also registered as a transform on `DateTimeField` as `__date`.

class TruncDay(*expression*, *output_field=None*, *tzinfo=None*, ***extra*)

kind = 'day'

class TruncHour(*expression*, *output_field=None*, *tzinfo=None*, ***extra*)

kind = 'hour'

class TruncMinute(*expression*, *output_field=None*, *tzinfo=None*, ***extra*)

kind = 'minute'

class TruncSecond(*expression*, *output_field=None*, *tzinfo=None*, ***extra*)

kind = 'second'

These are logically equivalent to `Trunc('datetime_field', kind)`. They truncate all parts of the date up to `kind` and allow grouping or filtering datetimes with less precision. `expression` must have an `output_field` of `DateTimeField`.

Usage example:

```
>>> from datetime import date, datetime
>>> from django.db.models import Count
>>> from django.db.models.functions import (
...     TruncDate, TruncDay, TruncHour, TruncMinute, TruncSecond,
... )
>>> from django.utils import timezone
>>> import pytz
>>> start1 = datetime(2014, 6, 15, 14, 30, 50, 321, tzinfo=timezone.utc)
>>> Experiment.objects.create(start_datetime=start1, start_date=start1.date())
>>> melb = pytz.timezone('Australia/Melbourne')
>>> Experiment.objects.annotate(
...     date=TruncDate('start_datetime'),
...     day=TruncDay('start_datetime', tzinfo=melb),
...     hour=TruncHour('start_datetime', tzinfo=melb),
...     minute=TruncMinute('start_datetime'),
...     second=TruncSecond('start_datetime'),
... ).values('date', 'day', 'hour', 'minute', 'second').get()
{'date': datetime.date(2014, 6, 15),
 'day': datetime.datetime(2014, 6, 16, 0, 0, tzinfo=<DstTzInfo 'Australia/Melbourne' AEST+10:00:00 S
 'hour': datetime.datetime(2014, 6, 16, 0, 0, tzinfo=<DstTzInfo 'Australia/Melbourne' AEST+10:00:00 S
 'minute': 'minute': datetime.datetime(2014, 6, 15, 14, 30, tzinfo=<UTC>),
 'second': datetime.datetime(2014, 6, 15, 14, 30, 50, tzinfo=<UTC>)
}
```

6.16 Request and response objects

6.16.1 Quick overview

Django uses request and response objects to pass state through the system.

When a page is requested, Django creates an *HttpRequest* object that contains metadata about the request. Then Django loads the appropriate view, passing the *HttpRequest* as the first argument to the view function. Each view is responsible for returning an *HttpResponse* object.

This document explains the APIs for *HttpRequest* and *HttpResponse* objects, which are defined in the *django.http* module.

6.16.2 `HttpRequest` objects

class `HttpRequest`

Attributes

All attributes should be considered read-only, unless stated otherwise.

`HttpRequest.`**`scheme`**
A string representing the scheme of the request (`http` or `https` usually).

`HttpRequest.`**`body`**
The raw HTTP request body as a byte string. This is useful for processing data in different ways than conventional HTML forms: binary images, XML payload etc. For processing conventional form data, use `HttpRequest.POST`.

You can also read from an HttpRequest using a file-like interface. See *`HttpRequest.read()`*.

`HttpRequest.`**`path`**
A string representing the full path to the requested page, not including the scheme or domain.

Example: `"/music/bands/the_beatles/"`

`HttpRequest.`**`path_info`**
Under some Web server configurations, the portion of the URL after the host name is split up into a script prefix portion and a path info portion. The `path_info` attribute always contains the path info portion of the path, no matter what Web server is being used. Using this instead of *`path`* can make your code easier to move between test and deployment servers.

For example, if the `WSGIScriptAlias` for your application is set to `"/minfo"`, then `path` might be `"/minfo/music/bands/the_beatles/"` and `path_info` would be `"/music/bands/the_beatles/"`.

`HttpRequest.`**`method`**
A string representing the HTTP method used in the request. This is guaranteed to be uppercase. Example:

```
if request.method == 'GET':
    do_something()
elif request.method == 'POST':
    do_something_else()
```

`HttpRequest.`**`encoding`**
A string representing the current encoding used to decode form submission data (or `None`, which means the *`DEFAULT_CHARSET`* setting is used). You can write to this attribute to change the encoding used when accessing the form data. Any subsequent attribute accesses (such as reading from `GET` or `POST`) will use the new `encoding` value. Useful if you know the form data is not in the *`DEFAULT_CHARSET`* encoding.

`HttpRequest.`**`content_type`**
A string representing the MIME type of the request, parsed from the `CONTENT_TYPE` header.

`HttpRequest.`**`content_params`**
A dictionary of key/value parameters included in the `CONTENT_TYPE` header.

`HttpRequest.`**`GET`**
A dictionary-like object containing all given HTTP GET parameters. See the *`QueryDict`* documentation below.

`HttpRequest.`**`POST`**
A dictionary-like object containing all given HTTP POST parameters, providing that the request contains form

data. See the *QueryDict* documentation below. If you need to access raw or non-form data posted in the request, access this through the *HttpRequest.body* attribute instead.

It's possible that a request can come in via POST with an empty `POST` dictionary – if, say, a form is requested via the POST HTTP method but does not include form data. Therefore, you shouldn't use `if request.POST` to check for use of the POST method; instead, use `if request.method == "POST"` (see above).

Note: `POST` does *not* include file-upload information. See `FILES`.

`HttpRequest.`**`COOKIES`**

A standard Python dictionary containing all cookies. Keys and values are strings.

`HttpRequest.`**`FILES`**

A dictionary-like object containing all uploaded files. Each key in `FILES` is the `name` from the `<input type="file" name="" />`. Each value in `FILES` is an *UploadedFile*.

See Managing files for more information.

Note that `FILES` will only contain data if the request method was POST and the `<form>` that posted to the request had `enctype="multipart/form-data"`. Otherwise, `FILES` will be a blank dictionary-like object.

`HttpRequest.`**`META`**

A standard Python dictionary containing all available HTTP headers. Available headers depend on the client and server, but here are some examples:

- `CONTENT_LENGTH` – The length of the request body (as a string).
- `CONTENT_TYPE` – The MIME type of the request body.
- `HTTP_ACCEPT` – Acceptable content types for the response.
- `HTTP_ACCEPT_ENCODING` – Acceptable encodings for the response.
- `HTTP_ACCEPT_LANGUAGE` – Acceptable languages for the response.
- `HTTP_HOST` – The HTTP Host header sent by the client.
- `HTTP_REFERER` – The referring page, if any.
- `HTTP_USER_AGENT` – The client's user-agent string.
- `QUERY_STRING` – The query string, as a single (unparsed) string.
- `REMOTE_ADDR` – The IP address of the client.
- `REMOTE_HOST` – The hostname of the client.
- `REMOTE_USER` – The user authenticated by the Web server, if any.
- `REQUEST_METHOD` – A string such as `"GET"` or `"POST"`.
- `SERVER_NAME` – The hostname of the server.
- `SERVER_PORT` – The port of the server (as a string).

With the exception of `CONTENT_LENGTH` and `CONTENT_TYPE`, as given above, any HTTP headers in the request are converted to `META` keys by converting all characters to uppercase, replacing any hyphens with underscores and adding an `HTTP_` prefix to the name. So, for example, a header called `X-Bender` would be mapped to the `META` key `HTTP_X_BENDER`.

Note that *runserver* strips all headers with underscores in the name, so you won't see them in `META`. This prevents header-spoofing based on ambiguity between underscores and dashes both being normalizing to underscores in WSGI environment variables. It matches the behavior of Web servers like Nginx and Apache 2.4+.

HttpRequest.**resolver_match**
An instance of *ResolverMatch* representing the resolved URL. This attribute is only set after URL resolving took place, which means it's available in all views but not in middleware which are executed before URL resolving takes place (you can use it in *process_view()* though).

Attributes set by application code

Django doesn't set these attributes itself but makes use of them if set by your application.

HttpRequest.**current_app**
The *url* template tag will use its value as the `current_app` argument to *reverse()*.

HttpRequest.**urlconf**
This will be used as the root URLconf for the current request, overriding the *ROOT_URLCONF* setting. See *How Django processes a request* for details.

`urlconf` can be set to `None` to revert any changes made by previous middleware and return to using the *ROOT_URLCONF*.

Setting `urlconf=None` raised *ImproperlyConfigured* in older versions.

Attributes set by middleware

Some of the middleware included in Django's contrib apps set attributes on the request. If you don't see the attribute on a request, be sure the appropriate middleware class is listed in *MIDDLEWARE*.

HttpRequest.**session**
From the *SessionMiddleware*: A readable and writable, dictionary-like object that represents the current session.

HttpRequest.**site**
From the *CurrentSiteMiddleware*: An instance of *Site* or *RequestSite* as returned by *get_current_site()* representing the current site.

HttpRequest.**user**
From the *AuthenticationMiddleware*: An instance of *AUTH_USER_MODEL* representing the currently logged-in user. If the user isn't currently logged in, `user` will be set to an instance of *AnonymousUser*. You can tell them apart with *is_authenticated*, like so:

```
    if request.user.is_authenticated:
        ... # Do something for logged-in users.
    else:
        ... # Do something for anonymous users.
```

Methods

HttpRequest.**get_host**()
Returns the originating host of the request using information from the `HTTP_X_FORWARDED_HOST` (if *USE_X_FORWARDED_HOST* is enabled) and `HTTP_HOST` headers, in that order. If they don't provide a value, the method uses a combination of `SERVER_NAME` and `SERVER_PORT` as detailed in **PEP 3333**.

Example: `"127.0.0.1:8000"`

: The *get_host()* method fails when the host is behind multiple proxies. One solution is to use middleware to rewrite the proxy headers, as in the following example:

```
    from django.utils.deprecation import MiddlewareMixin

    class MultipleProxyMiddleware(MiddlewareMixin):
        FORWARDED_FOR_FIELDS = [
            'HTTP_X_FORWARDED_FOR',
            'HTTP_X_FORWARDED_HOST',
            'HTTP_X_FORWARDED_SERVER',
        ]

        def process_request(self, request):
            """
            Rewrites the proxy headers so that only the most
            recent proxy is used.
            """
            for field in self.FORWARDED_FOR_FIELDS:
                if field in request.META:
                    if ',' in request.META[field]:
                        parts = request.META[field].split(',')
                        request.META[field] = parts[-1].strip()
```

This middleware should be positioned before any other middleware that relies on the value of *get_host()* – for instance, *CommonMiddleware* or *CsrfViewMiddleware*.

`HttpRequest.`**`get_port`**`()`

Returns the originating port of the request using information from the `HTTP_X_FORWARDED_PORT` (if *USE_X_FORWARDED_PORT* is enabled) and `SERVER_PORT` `META` variables, in that order.

`HttpRequest.`**`get_full_path`**`()`

Returns the `path`, plus an appended query string, if applicable.

Example: `"/music/bands/the_beatles/?print=true"`

`HttpRequest.`**`build_absolute_uri`**`(`*location*`)`

Returns the absolute URI form of `location`. If no location is provided, the location will be set to `request.get_full_path()`.

If the location is already an absolute URI, it will not be altered. Otherwise the absolute URI is built using the server variables available in this request.

Example: `"https://example.com/music/bands/the_beatles/?print=true"`

: Mixing HTTP and HTTPS on the same site is discouraged, therefore *build_absolute_uri()* will always generate an absolute URI with the same scheme the current request has. If you need to redirect users to HTTPS, it's best to let your Web server redirect all HTTP traffic to HTTPS.

`HttpRequest.`**`get_signed_cookie`**`(`*key*, *default=RAISE_ERROR*, *salt=''*, *max_age=None*`)`

Returns a cookie value for a signed cookie, or raises a `django.core.signing.BadSignature` exception if the signature is no longer valid. If you provide the `default` argument the exception will be suppressed and that default value will be returned instead.

The optional `salt` argument can be used to provide extra protection against brute force attacks on your secret key. If supplied, the `max_age` argument will be checked against the signed timestamp attached to the cookie value to ensure the cookie is not older than `max_age` seconds.

For example:

```
>>> request.get_signed_cookie('name')
'Tony'
>>> request.get_signed_cookie('name', salt='name-salt')
'Tony' # assuming cookie was set using the same salt
>>> request.get_signed_cookie('non-existing-cookie')
...
KeyError: 'non-existing-cookie'
>>> request.get_signed_cookie('non-existing-cookie', False)
False
>>> request.get_signed_cookie('cookie-that-was-tampered-with')
...
BadSignature: ...
>>> request.get_signed_cookie('name', max_age=60)
...
SignatureExpired: Signature age 1677.3839159 > 60 seconds
>>> request.get_signed_cookie('name', False, max_age=60)
False
```

See cryptographic signing for more information.

`HttpRequest.`**`is_secure`**`()`
Returns `True` if the request is secure; that is, if it was made with HTTPS.

`HttpRequest.`**`is_ajax`**`()`
Returns `True` if the request was made via an `XMLHttpRequest`, by checking the `HTTP_X_REQUESTED_WITH` header for the string `'XMLHttpRequest'`. Most modern JavaScript libraries send this header. If you write your own XMLHttpRequest call (on the browser side), you'll have to set this header manually if you want `is_ajax()` to work.

If a response varies on whether or not it's requested via AJAX and you are using some form of caching like Django's *`cache middleware`*, you should decorate the view with *`vary_on_headers('HTTP_X_REQUESTED_WITH')`* so that the responses are properly cached.

`HttpRequest.`**`read`**`(`*size=None*`)`

`HttpRequest.`**`readline`**`()`

`HttpRequest.`**`readlines`**`()`

`HttpRequest.`**`xreadlines`**`()`

`HttpRequest.`**`__iter__`**`()`
Methods implementing a file-like interface for reading from an HttpRequest instance. This makes it possible to consume an incoming request in a streaming fashion. A common use-case would be to process a big XML payload with an iterative parser without constructing a whole XML tree in memory.

Given this standard interface, an HttpRequest instance can be passed directly to an XML parser such as ElementTree:

```
import xml.etree.ElementTree as ET
for element in ET.iterparse(request):
    process(element)
```

6.16.3 `QueryDict` objects

class `QueryDict`

In an *`HttpRequest`* object, the `GET` and `POST` attributes are instances of `django.http.QueryDict`, a dictionary-like class customized to deal with multiple values for the same key. This is necessary because some HTML form elements, notably `<select multiple>`, pass multiple values for the same key.

The QueryDicts at request.POST and request.GET will be immutable when accessed in a normal request/response cycle. To get a mutable version you need to use .copy().

Methods

QueryDict implements all the standard dictionary methods because it's a subclass of dictionary. Exceptions are outlined here:

QueryDict.**__init__**(*query_string=None*, *mutable=False*, *encoding=None*)
Instantiates a QueryDict object based on query_string.

```
>>> QueryDict('a=1&a=2&c=3')
<QueryDict: {'a': ['1', '2'], 'c': ['3']}>
```

If query_string is not passed in, the resulting QueryDict will be empty (it will have no keys or values).

Most QueryDicts you encounter, and in particular those at request.POST and request.GET, will be immutable. If you are instantiating one yourself, you can make it mutable by passing mutable=True to its __init__().

Strings for setting both keys and values will be converted from encoding to unicode. If encoding is not set, it defaults to *DEFAULT_CHARSET*.

QueryDict.**__getitem__**(*key*)
Returns the value for the given key. If the key has more than one value, __getitem__() returns the last value. Raises django.utils.datastructures.MultiValueDictKeyError if the key does not exist. (This is a subclass of Python's standard KeyError, so you can stick to catching KeyError.)

QueryDict.**__setitem__**(*key*, *value*)
Sets the given key to [value] (a Python list whose single element is value). Note that this, as other dictionary functions that have side effects, can only be called on a mutable QueryDict (such as one that was created via copy()).

QueryDict.**__contains__**(*key*)
Returns True if the given key is set. This lets you do, e.g., if "foo" in request.GET.

QueryDict.**get**(*key*, *default=None*)
Uses the same logic as __getitem__() above, with a hook for returning a default value if the key doesn't exist.

QueryDict.**setdefault**(*key*, *default=None*)
Just like the standard dictionary setdefault() method, except it uses __setitem__() internally.

QueryDict.**update**(*other_dict*)
Takes either a QueryDict or standard dictionary. Just like the standard dictionary update() method, except it *appends* to the current dictionary items rather than replacing them. For example:

```
>>> q = QueryDict('a=1', mutable=True)
>>> q.update({'a': '2'})
>>> q.getlist('a')
['1', '2']
>>> q['a'] # returns the last
'2'
```

QueryDict.**items**()
Just like the standard dictionary items() method, except this uses the same last-value logic as __getitem__(). For example:

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.items()
[('a', '3')]
```

`QueryDict.`**`iteritems`**`()`
Just like the standard dictionary `iteritems()` method. Like *`QueryDict.items()`* this uses the same last-value logic as *`QueryDict.__getitem__()`*.

`QueryDict.`**`iterlists`**`()`
Like *`QueryDict.iteritems()`* except it includes all values, as a list, for each member of the dictionary.

`QueryDict.`**`values`**`()`
Just like the standard dictionary `values()` method, except this uses the same last-value logic as `__getitem__()`. For example:

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.values()
['3']
```

`QueryDict.`**`itervalues`**`()`
Just like *`QueryDict.values()`*, except an iterator.

In addition, `QueryDict` has the following methods:

`QueryDict.`**`copy`**`()`
Returns a copy of the object, using `copy.deepcopy()` from the Python standard library. This copy will be mutable even if the original was not.

`QueryDict.`**`getlist`**`(`*key*, *default=None*`)`
Returns the data with the requested key, as a Python list. Returns an empty list if the key doesn't exist and no default value was provided. It's guaranteed to return a list of some sort unless the default value provided is not a list.

`QueryDict.`**`setlist`**`(`*key*, *list_*`)`
Sets the given key to `list_` (unlike `__setitem__()`).

`QueryDict.`**`appendlist`**`(`*key*, *item*`)`
Appends an item to the internal list associated with key.

`QueryDict.`**`setlistdefault`**`(`*key*, *default_list=None*`)`
Just like `setdefault`, except it takes a list of values instead of a single value.

`QueryDict.`**`lists`**`()`
Like *`items()`*, except it includes all values, as a list, for each member of the dictionary. For example:

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.lists()
[('a', ['1', '2', '3'])]
```

`QueryDict.`**`pop`**`(`*key*`)`
Returns a list of values for the given key and removes them from the dictionary. Raises `KeyError` if the key does not exist. For example:

```
>>> q = QueryDict('a=1&a=2&a=3', mutable=True)
>>> q.pop('a')
['1', '2', '3']
```

`QueryDict.`**`popitem`**`()`
Removes an arbitrary member of the dictionary (since there's no concept of ordering), and returns a two value tuple containing the key and a list of all values for the key. Raises `KeyError` when called on an empty dictionary. For example:

```
>>> q = QueryDict('a=1&a=2&a=3', mutable=True)
>>> q.popitem()
('a', ['1', '2', '3'])
```

`QueryDict.dict()`
Returns `dict` representation of `QueryDict`. For every (key, list) pair in `QueryDict`, `dict` will have (key, item), where item is one element of the list, using same logic as *QueryDict.__getitem__()*:

```
>>> q = QueryDict('a=1&a=3&a=5')
>>> q.dict()
{'a': '5'}
```

`QueryDict.urlencode`(*safe=None*)
Returns a string of the data in query-string format. Example:

```
>>> q = QueryDict('a=2&b=3&b=5')
>>> q.urlencode()
'a=2&b=3&b=5'
```

Optionally, urlencode can be passed characters which do not require encoding. For example:

```
>>> q = QueryDict(mutable=True)
>>> q['next'] = '/a&b/'
>>> q.urlencode(safe='/')
'next=/a%26b/'
```

6.16.4 `HttpResponse` objects

class `HttpResponse`

In contrast to *HttpRequest* objects, which are created automatically by Django, *HttpResponse* objects are your responsibility. Each view you write is responsible for instantiating, populating and returning an *HttpResponse*.

The *HttpResponse* class lives in the *django.http* module.

Usage

Passing strings

Typical usage is to pass the contents of the page, as a string, to the *HttpResponse* constructor:

```
>>> from django.http import HttpResponse
>>> response = HttpResponse("Here's the text of the Web page.")
>>> response = HttpResponse("Text only, please.", content_type="text/plain")
```

But if you want to add content incrementally, you can use `response` as a file-like object:

```
>>> response = HttpResponse()
>>> response.write("<p>Here's the text of the Web page.</p>")
>>> response.write("<p>Here's another paragraph.</p>")
```

Passing iterators

Finally, you can pass `HttpResponse` an iterator rather than strings. `HttpResponse` will consume the iterator immediately, store its content as a string, and discard it. Objects with a `close()` method such as files and generators are immediately closed.

If you need the response to be streamed from the iterator to the client, you must use the *StreamingHttpResponse* class instead.

Objects with a `close()` method used to be closed when the WSGI server called `close()` on the response.

Setting header fields

To set or remove a header field in your response, treat it like a dictionary:

```
>>> response = HttpResponse()
>>> response['Age'] = 120
>>> del response['Age']
```

Note that unlike a dictionary, `del` doesn't raise `KeyError` if the header field doesn't exist.

For setting the `Cache-Control` and `Vary` header fields, it is recommended to use the *patch_cache_control()* and *patch_vary_headers()* methods from *django.utils.cache*, since these fields can have multiple, comma-separated values. The "patch" methods ensure that other values, e.g. added by a middleware, are not removed.

HTTP header fields cannot contain newlines. An attempt to set a header field containing a newline character (CR or LF) will raise `BadHeaderError`

Telling the browser to treat the response as a file attachment

To tell the browser to treat the response as a file attachment, use the `content_type` argument and set the `Content-Disposition` header. For example, this is how you might return a Microsoft Excel spreadsheet:

```
>>> response = HttpResponse(my_data, content_type='application/vnd.ms-excel')
>>> response['Content-Disposition'] = 'attachment; filename="foo.xls"'
```

There's nothing Django-specific about the `Content-Disposition` header, but it's easy to forget the syntax, so we've included it here.

Attributes

`HttpResponse.`**`content`**
: A bytestring representing the content, encoded from a Unicode object if necessary.

`HttpResponse.`**`charset`**
: A string denoting the charset in which the response will be encoded. If not given at `HttpResponse` instantiation time, it will be extracted from `content_type` and if that is unsuccessful, the *DEFAULT_CHARSET* setting will be used.

`HttpResponse.`**`status_code`**
: The **HTTP status code** for the response.

  Unless *reason_phrase* is explicitly set, modifying the value of `status_code` outside the constructor will also modify the value of `reason_phrase`.

`HttpResponse.`**`reason_phrase`**
: The HTTP reason phrase for the response.

  `reason_phrase` no longer defaults to all capital letters. It now uses the **HTTP standard's** default reason phrases.

  Unless explicitly set, `reason_phrase` is determined by the current value of *status_code*.

HttpResponse.**streaming**
This is always False.

This attribute exists so middleware can treat streaming responses differently from regular responses.

HttpResponse.**closed**
True if the response has been closed.

Methods

HttpResponse.**__init__**(*content=''*, *content_type=None*, *status=200*, *reason=None*, *charset=None*)
Instantiates an HttpResponse object with the given page content and content type.

content should be an iterator or a string. If it's an iterator, it should return strings, and those strings will be joined together to form the content of the response. If it is not an iterator or a string, it will be converted to a string when accessed.

content_type is the MIME type optionally completed by a character set encoding and is used to fill the HTTP Content-Type header. If not specified, it is formed by the *DEFAULT_CONTENT_TYPE* and *DEFAULT_CHARSET* settings, by default: "*text/html; charset=utf-8*".

status is the **HTTP status code** for the response.

reason is the HTTP response phrase. If not provided, a default phrase will be used.

charset is the charset in which the response will be encoded. If not given it will be extracted from content_type, and if that is unsuccessful, the *DEFAULT_CHARSET* setting will be used.

HttpResponse.**__setitem__**(*header*, *value*)
Sets the given header name to the given value. Both header and value should be strings.

HttpResponse.**__delitem__**(*header*)
Deletes the header with the given name. Fails silently if the header doesn't exist. Case-insensitive.

HttpResponse.**__getitem__**(*header*)
Returns the value for the given header name. Case-insensitive.

HttpResponse.**has_header**(*header*)
Returns True or False based on a case-insensitive check for a header with the given name.

HttpResponse.**setdefault**(*header*, *value*)
Sets a header unless it has already been set.

HttpResponse.**set_cookie**(*key*, *value=''*, *max_age=None*, *expires=None*, *path='/'*, *domain=None*, *secure=None*, *httponly=False*)
Sets a cookie. The parameters are the same as in the Morsel cookie object in the Python standard library.

- max_age should be a number of seconds, or None (default) if the cookie should last only as long as the client's browser session. If expires is not specified, it will be calculated.
- expires should either be a string in the format "Wdy, DD-Mon-YY HH:MM:SS GMT" or a datetime.datetime object in UTC. If expires is a datetime object, the max_age will be calculated.
- Use domain if you want to set a cross-domain cookie. For example, domain=".lawrence.com" will set a cookie that is readable by the domains www.lawrence.com, blogs.lawrence.com and calendars.lawrence.com. Otherwise, a cookie will only be readable by the domain that set it.
- Use httponly=True if you want to prevent client-side JavaScript from having access to the cookie.

HTTPOnly is a flag included in a Set-Cookie HTTP response header. It is not part of the **RFC 2109** standard for cookies, and it isn't honored consistently by all browsers. However, when it is honored, it can be a useful way to mitigate the risk of a client-side script from accessing the protected cookie data.

: Both **RFC 2109** and **RFC 6265** state that user agents should support cookies of at least 4096 bytes. For many browsers this is also the maximum size. Django will not raise an exception if there's an attempt to store a cookie of more than 4096 bytes, but many browsers will not set the cookie correctly.

`HttpResponse.`**`set_signed_cookie`**(*key*, *value*, *salt=''*, *max_age=None*, *expires=None*, *path='/'*, *domain=None*, *secure=None*, *httponly=True*)

Like *`set_cookie()`*, but cryptographic signing the cookie before setting it. Use in conjunction with *`HttpRequest.get_signed_cookie()`*. You can use the optional `salt` argument for added key strength, but you will need to remember to pass it to the corresponding *`HttpRequest.get_signed_cookie()`* call.

`HttpResponse.`**`delete_cookie`**(*key*, *path='/'*, *domain=None*)

Deletes the cookie with the given key. Fails silently if the key doesn't exist.

Due to the way cookies work, `path` and `domain` should be the same values you used in `set_cookie()` – otherwise the cookie may not be deleted.

`HttpResponse.`**`write`**(*content*)

This method makes an *`HttpResponse`* instance a file-like object.

`HttpResponse.`**`flush`**()

This method makes an *`HttpResponse`* instance a file-like object.

`HttpResponse.`**`tell`**()

This method makes an *`HttpResponse`* instance a file-like object.

`HttpResponse.`**`getvalue`**()

Returns the value of *`HttpResponse.content`*. This method makes an *`HttpResponse`* instance a stream-like object.

`HttpResponse.`**`readable`**()

Always `False`. This method makes an *`HttpResponse`* instance a stream-like object.

`HttpResponse.`**`seekable`**()

Always `False`. This method makes an *`HttpResponse`* instance a stream-like object.

`HttpResponse.`**`writable`**()

Always `True`. This method makes an *`HttpResponse`* instance a stream-like object.

`HttpResponse.`**`writelines`**(*lines*)

Writes a list of lines to the response. Line separators are not added. This method makes an *`HttpResponse`* instance a stream-like object.

`HttpResponse` subclasses

Django includes a number of `HttpResponse` subclasses that handle different types of HTTP responses. Like `HttpResponse`, these subclasses live in *`django.http`*.

class `HttpResponseRedirect`

The first argument to the constructor is required – the path to redirect to. This can be a fully qualified URL (e.g. `'https://www.yahoo.com/search/'`), an absolute path with no domain (e.g. `'/search/'`), or even a relative path (e.g. `'search/'`). In that last case, the client browser will reconstruct the full URL itself according to the current path. See *`HttpResponse`* for other optional constructor arguments. Note that this returns an HTTP status code 302.

url
This read-only attribute represents the URL the response will redirect to (equivalent to the `Location` response header).

class HttpResponsePermanentRedirect
Like *`HttpResponseRedirect`*, but it returns a permanent redirect (HTTP status code 301) instead of a "found" redirect (status code 302).

class HttpResponseNotModified
The constructor doesn't take any arguments and no content should be added to this response. Use this to designate that a page hasn't been modified since the user's last request (status code 304).

class HttpResponseBadRequest
Acts just like *`HttpResponse`* but uses a 400 status code.

class HttpResponseNotFound
Acts just like *`HttpResponse`* but uses a 404 status code.

class HttpResponseForbidden
Acts just like *`HttpResponse`* but uses a 403 status code.

class HttpResponseNotAllowed
Like *`HttpResponse`*, but uses a 405 status code. The first argument to the constructor is required: a list of permitted methods (e.g. `['GET', 'POST']`).

class HttpResponseGone
Acts just like *`HttpResponse`* but uses a 410 status code.

class HttpResponseServerError
Acts just like *`HttpResponse`* but uses a 500 status code.

: If a custom subclass of *`HttpResponse`* implements a `render` method, Django will treat it as emulating a *`SimpleTemplateResponse`*, and the `render` method must itself return a valid response object.

6.16.5 `JsonResponse` objects

class JsonResponse(*data*, *encoder=DjangoJSONEncoder*, *safe=True*, *json_dumps_params=None*, ***kwargs*)
An *`HttpResponse`* subclass that helps to create a JSON-encoded response. It inherits most behavior from its superclass with a couple differences:

Its default `Content-Type` header is set to `application/json`.

The first parameter, `data`, should be a `dict` instance. If the `safe` parameter is set to `False` (see below) it can be any JSON-serializable object.

The `encoder`, which defaults to *`django.core.serializers.json.DjangoJSONEncoder`*, will be used to serialize the data. See *JSON serialization* for more details about this serializer.

The `safe` boolean parameter defaults to `True`. If it's set to `False`, any object can be passed for serialization (otherwise only `dict` instances are allowed). If `safe` is `True` and a non-`dict` object is passed as the first argument, a `TypeError` will be raised.

The `json_dumps_params` parameter is a dictionary of keyword arguments to pass to the `json.dumps()` call used to generate the response.

The `json_dumps_params` argument was added.

Usage

Typical usage could look like:

```
>>> from django.http import JsonResponse
>>> response = JsonResponse({'foo': 'bar'})
>>> response.content
b'{"foo": "bar"}'
```

Serializing non-dictionary objects

In order to serialize objects other than `dict` you must set the `safe` parameter to `False`:

```
>>> response = JsonResponse([1, 2, 3], safe=False)
```

Without passing `safe=False`, a `TypeError` will be raised.

: Before the 5th edition of ECMAScript it was possible to poison the JavaScript `Array` constructor. For this reason, Django does not allow passing non-dict objects to the *`JsonResponse`* constructor by default. However, most modern browsers implement EcmaScript 5 which removes this attack vector. Therefore it is possible to disable this security precaution.

Changing the default JSON encoder

If you need to use a different JSON encoder class you can pass the `encoder` parameter to the constructor method:

```
>>> response = JsonResponse(data, encoder=MyJSONEncoder)
```

6.16.6 `StreamingHttpResponse` objects

class `StreamingHttpResponse`

The *`StreamingHttpResponse`* class is used to stream a response from Django to the browser. You might want to do this if generating the response takes too long or uses too much memory. For instance, it's useful for *generating large CSV files*.

Performance considerations

Django is designed for short-lived requests. Streaming responses will tie a worker process for the entire duration of the response. This may result in poor performance.

Generally speaking, you should perform expensive tasks outside of the request-response cycle, rather than resorting to a streamed response.

The *`StreamingHttpResponse`* is not a subclass of *`HttpResponse`*, because it features a slightly different API. However, it is almost identical, with the following notable differences:

- It should be given an iterator that yields strings as content.
- You cannot access its content, except by iterating the response object itself. This should only occur when the response is returned to the client.
- It has no `content` attribute. Instead, it has a *`streaming_content`* attribute.

- You cannot use the file-like object `tell()` or `write()` methods. Doing so will raise an exception.

`StreamingHttpResponse` should only be used in situations where it is absolutely required that the whole content isn't iterated before transferring the data to the client. Because the content can't be accessed, many middlewares can't function normally. For example the `ETag` and `Content-Length` headers can't be generated for streaming responses.

Attributes

`StreamingHttpResponse.`**`streaming_content`**
An iterator of strings representing the content.

`StreamingHttpResponse.`**`status_code`**
The **HTTP status code** for the response.

Unless *`reason_phrase`* is explicitly set, modifying the value of `status_code` outside the constructor will also modify the value of `reason_phrase`.

`StreamingHttpResponse.`**`reason_phrase`**
The HTTP reason phrase for the response.

`reason_phrase` no longer defaults to all capital letters. It now uses the **HTTP standard's** default reason phrases.

Unless explicitly set, `reason_phrase` is determined by the current value of *`status_code`*.

`StreamingHttpResponse.`**`streaming`**
This is always `True`.

6.16.7 `FileResponse` objects

class `FileResponse`

`FileResponse` is a subclass of *`StreamingHttpResponse`* optimized for binary files. It uses wsgi.file_wrapper if provided by the wsgi server, otherwise it streams the file out in small chunks.

`FileResponse` expects a file open in binary mode like so:

```
>>> from django.http import FileResponse
>>> response = FileResponse(open('myfile.png', 'rb'))
```

6.17 `SchemaEditor`

class `BaseDatabaseSchemaEditor`

Django's migration system is split into two parts; the logic for calculating and storing what operations should be run (`django.db.migrations`), and the database abstraction layer that turns things like "create a model" or "delete a field" into SQL - which is the job of the `SchemaEditor`.

It's unlikely that you will want to interact directly with `SchemaEditor` as a normal developer using Django, but if you want to write your own migration system, or have more advanced needs, it's a lot nicer than writing SQL.

Each database backend in Django supplies its own version of `SchemaEditor`, and it's always accessible via the `connection.schema_editor()` context manager:

```
with connection.schema_editor() as schema_editor:
    schema_editor.delete_model(MyModel)
```

It must be used via the context manager as this allows it to manage things like transactions and deferred SQL (like creating `ForeignKey` constraints).

It exposes all possible operations as methods, that should be called in the order you wish changes to be applied. Some possible operations or types of change are not possible on all databases - for example, MyISAM does not support foreign key constraints.

If you are writing or maintaining a third-party database backend for Django, you will need to provide a `SchemaEditor` implementation in order to work with 1.7's migration functionality - however, as long as your database is relatively standard in its use of SQL and relational design, you should be able to subclass one of the built-in Django `SchemaEditor` classes and just tweak the syntax a little. Also note that there are a few new database features that migrations will look for: `can_rollback_ddl` and `supports_combined_alters` are the most important.

6.17.1 Methods

execute()

`BaseDatabaseSchemaEditor.`**`execute`**(*sql*, *params=[]*)

Executes the SQL statement passed in, with parameters if supplied. This is a simple wrapper around the normal database cursors that allows capture of the SQL to a `.sql` file if the user wishes.

create_model()

`BaseDatabaseSchemaEditor.`**`create_model`**(*model*)

Creates a new table in the database for the provided model, along with any unique constraints or indexes it requires.

delete_model()

`BaseDatabaseSchemaEditor.`**`delete_model`**(*model*)

Drops the model's table in the database along with any unique constraints or indexes it has.

alter_unique_together()

`BaseDatabaseSchemaEditor.`**`alter_unique_together`**(*model*, *old_unique_together*, *new_unique_together*)

Changes a model's *unique_together* value; this will add or remove unique constraints from the model's table until they match the new value.

alter_index_together()

`BaseDatabaseSchemaEditor.`**`alter_index_together`**(*model*, *old_index_together*, *new_index_together*)

Changes a model's *index_together* value; this will add or remove indexes from the model's table until they match the new value.

alter_db_table()

BaseDatabaseSchemaEditor.**alter_db_table**(*model*, *old_db_table*, *new_db_table*)

Renames the model's table from `old_db_table` to `new_db_table`.

alter_db_tablespace()

BaseDatabaseSchemaEditor.**alter_db_tablespace**(*model*, *old_db_tablespace*, *new_db_tablespace*)

Moves the model's table from one tablespace to another.

add_field()

BaseDatabaseSchemaEditor.**add_field**(*model*, *field*)

Adds a column (or sometimes multiple) to the model's table to represent the field. This will also add indexes or a unique constraint if the field has `db_index=True` or `unique=True`.

If the field is a `ManyToManyField` without a value for `through`, instead of creating a column, it will make a table to represent the relationship. If `through` is provided, it is a no-op.

If the field is a `ForeignKey`, this will also add the foreign key constraint to the column.

remove_field()

BaseDatabaseSchemaEditor.**remove_field**(*model*, *field*)

Removes the column(s) representing the field from the model's table, along with any unique constraints, foreign key constraints, or indexes caused by that field.

If the field is a ManyToManyField without a value for `through`, it will remove the table created to track the relationship. If `through` is provided, it is a no-op.

alter_field()

BaseDatabaseSchemaEditor.**alter_field**(*model*, *old_field*, *new_field*, *strict=False*)

This transforms the field on the model from the old field to the new one. This includes changing the name of the column (the *db_column* attribute), changing the type of the field (if the field class changes), changing the `NULL` status of the field, adding or removing field-only unique constraints and indexes, changing primary key, and changing the destination of `ForeignKey` constraints.

The most common transformation this cannot do is transforming a `ManyToManyField` into a normal Field or vice-versa; Django cannot do this without losing data, and so it will refuse to do it. Instead, *remove_field()* and *add_field()* should be called separately.

If the database has the `supports_combined_alters`, Django will try and do as many of these in a single database call as possible; otherwise, it will issue a separate ALTER statement for each change, but will not issue ALTERs where no change is required (as South often did).

6.17.2 Attributes

All attributes should be considered read-only unless stated otherwise.

connection

SchemaEditor.**connection**

A connection object to the database. A useful attribute of the connection is `alias` which can be used to determine the name of the database being accessed.

This is useful when doing data migrations for *migrations with multiple databases*.

6.18 Settings

- *Core Settings*
- *Auth*
- *Messages*
- *Sessions*
- *Sites*
- *Static Files*
- *Core Settings Topical Index*

: Be careful when you override settings, especially when the default value is a non-empty list or dictionary, such as *MIDDLEWARE_CLASSES* and *STATICFILES_FINDERS*. Make sure you keep the components required by the features of Django you wish to use.

6.18.1 Core Settings

Here's a list of settings available in Django core and their default values. Settings provided by contrib apps are listed below, followed by a topical index of the core settings. For introductory material, see the settings topic guide.

ABSOLUTE_URL_OVERRIDES

Default: `{}` (Empty dictionary)

A dictionary mapping `"app_label.model_name"` strings to functions that take a model object and return its URL. This is a way of inserting or overriding `get_absolute_url()` methods on a per-installation basis. Example:

```
ABSOLUTE_URL_OVERRIDES = {
    'blogs.weblog': lambda o: "/blogs/%s/" % o.slug,
    'news.story': lambda o: "/stories/%s/%s/" % (o.pub_year, o.slug),
}
```

Note that the model name used in this setting should be all lower-case, regardless of the case of the actual model class name.

ADMINS

Default: `[]` (Empty list)

A list of all the people who get code error notifications. When `DEBUG=False` and a view raises an exception, Django will email these people with the full exception information. Each item in the list should be a tuple of (Full name, email address). Example:

```
[('John', 'john@example.com'), ('Mary', 'mary@example.com')]
```

Note that Django will email *all* of these people whenever an error happens. See Error reporting for more information.

ALLOWED_HOSTS

Default: `[]` (Empty list)

A list of strings representing the host/domain names that this Django site can serve. This is a security measure to prevent *HTTP Host header attacks*, which are possible even under many seemingly-safe web server configurations.

Values in this list can be fully qualified names (e.g. `'www.example.com'`), in which case they will be matched against the request's `Host` header exactly (case-insensitive, not including port). A value beginning with a period can be used as a subdomain wildcard: `'.example.com'` will match `example.com`, `www.example.com`, and any other subdomain of `example.com`. A value of `'*'` will match anything; in this case you are responsible to provide your own validation of the `Host` header (perhaps in a middleware; if so this middleware must be listed first in *MIDDLEWARE*).

Django also allows the fully qualified domain name (FQDN) of any entries. Some browsers include a trailing dot in the `Host` header which Django strips when performing host validation.

If the `Host` header (or `X-Forwarded-Host` if *USE_X_FORWARDED_HOST* is enabled) does not match any value in this list, the *django.http.HttpRequest.get_host()* method will raise *SuspiciousOperation*.

When *DEBUG* is `True` and `ALLOWED_HOSTS` is empty, the host is validated against `['localhost', '127.0.0.1', '[::1]']`.

This validation only applies via *get_host()*; if your code accesses the `Host` header directly from `request.META` you are bypassing this security protection.

In older versions, `ALLOWED_HOSTS` wasn't checked if `DEBUG=True`. This was also changed in Django 1.9.11 and 1.8.16 to prevent a DNS rebinding attack.

APPEND_SLASH

Default: `True`

When set to `True`, if the request URL does not match any of the patterns in the URLconf and it doesn't end in a slash, an HTTP redirect is issued to the same URL with a slash appended. Note that the redirect may cause any data submitted in a POST request to be lost.

The *APPEND_SLASH* setting is only used if *CommonMiddleware* is installed (see Middleware). See also *PREPEND_WWW*.

CACHES

Default:

```
{
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
    }
}
```

A dictionary containing the settings for all caches to be used with Django. It is a nested dictionary whose contents maps cache aliases to a dictionary containing the options for an individual cache.

The *CACHES* setting must configure a `default` cache; any number of additional caches may also be specified. If you are using a cache backend other than the local memory cache, or you need to define multiple caches, other options will be required. The following cache options are available.

BACKEND

Default: `''` (Empty string)

The cache backend to use. The built-in cache backends are:

- `'django.core.cache.backends.db.DatabaseCache'`
- `'django.core.cache.backends.dummy.DummyCache'`
- `'django.core.cache.backends.filebased.FileBasedCache'`
- `'django.core.cache.backends.locmem.LocMemCache'`
- `'django.core.cache.backends.memcached.MemcachedCache'`
- `'django.core.cache.backends.memcached.PyLibMCCache'`

You can use a cache backend that doesn't ship with Django by setting *BACKEND* to a fully-qualified path of a cache backend class (i.e. `mypackage.backends.whatever.WhateverCache`).

KEY_FUNCTION

A string containing a dotted path to a function (or any callable) that defines how to compose a prefix, version and key into a final cache key. The default implementation is equivalent to the function:

```
def make_key(key, key_prefix, version):
    return ':'.join([key_prefix, str(version), key])
```

You may use any key function you want, as long as it has the same argument signature.

See the *cache documentation* for more information.

KEY_PREFIX

Default: `''` (Empty string)

A string that will be automatically included (prepended by default) to all cache keys used by the Django server.

See the *cache documentation* for more information.

LOCATION

Default: `''` (Empty string)

The location of the cache to use. This might be the directory for a file system cache, a host and port for a memcache server, or simply an identifying name for a local memory cache. e.g.:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': '/var/tmp/django_cache',
    }
}
```

OPTIONS

Default: `None`

Extra parameters to pass to the cache backend. Available parameters vary depending on your cache backend.

Some information on available parameters can be found in the *cache arguments* documentation. For more information, consult your backend module's own documentation.

TIMEOUT

Default: `300`

The number of seconds before a cache entry is considered stale. If the value of this settings is `None`, cache entries will not expire.

VERSION

Default: `1`

The default version number for cache keys generated by the Django server.

See the *cache documentation* for more information.

CACHE_MIDDLEWARE_ALIAS

Default: `default`

The cache connection to use for the *cache middleware*.

CACHE_MIDDLEWARE_KEY_PREFIX

Default: `''` (Empty string)

A string which will be prefixed to the cache keys generated by the *cache middleware*. This prefix is combined with the *KEY_PREFIX* setting; it does not replace it.

See Django's cache framework.

CACHE_MIDDLEWARE_SECONDS

Default: `600`

The default number of seconds to cache a page for the *cache middleware*.

See Django's cache framework.

CSRF_COOKIE_AGE

Default: `31449600` (approximately 1 year, in seconds)

The age of CSRF cookies, in seconds.

The reason for setting a long-lived expiration time is to avoid problems in the case of a user closing a browser or bookmarking a page and then loading that page from a browser cache. Without persistent cookies, the form submission would fail in this case.

Some browsers (specifically Internet Explorer) can disallow the use of persistent cookies or can have the indexes to the cookie jar corrupted on disk, thereby causing CSRF protection checks to (sometimes intermittently) fail. Change this setting to `None` to use session-based CSRF cookies, which keep the cookies in-memory instead of on persistent storage.

CSRF_COOKIE_DOMAIN

Default: `None`

The domain to be used when setting the CSRF cookie. This can be useful for easily allowing cross-subdomain requests to be excluded from the normal cross site request forgery protection. It should be set to a string such as `".example.com"` to allow a POST request from a form on one subdomain to be accepted by a view served from another subdomain.

Please note that the presence of this setting does not imply that Django's CSRF protection is safe from cross-subdomain attacks by default - please see the *CSRF limitations* section.

CSRF_COOKIE_HTTPONLY

Default: `False`

Whether to use `HttpOnly` flag on the CSRF cookie. If this is set to `True`, client-side JavaScript will not to be able to access the CSRF cookie.

This can help prevent malicious JavaScript from bypassing CSRF protection. If you enable this and need to send the value of the CSRF token with Ajax requests, your JavaScript will need to pull the value from a hidden CSRF token form input on the page instead of from the cookie.

See *SESSION_COOKIE_HTTPONLY* for details on `HttpOnly`.

CSRF_COOKIE_NAME

Default: `'csrftoken'`

The name of the cookie to use for the CSRF authentication token. This can be whatever you want (as long as it's different from the other cookie names in your application). See Cross Site Request Forgery protection.

CSRF_COOKIE_PATH

Default: `'/'`

The path set on the CSRF cookie. This should either match the URL path of your Django installation or be a parent of that path.

This is useful if you have multiple Django instances running under the same hostname. They can use different cookie paths, and each instance will only see its own CSRF cookie.

CSRF_COOKIE_SECURE

Default: `False`

Whether to use a secure cookie for the CSRF cookie. If this is set to `True`, the cookie will be marked as "secure," which means browsers may ensure that the cookie is only sent with an HTTPS connection.

CSRF_FAILURE_VIEW

Default: `'django.views.csrf.csrf_failure'`

A dotted path to the view function to be used when an incoming request is rejected by the CSRF protection. The function should have this signature:

```
def csrf_failure(request, reason=""):
    ...
```

where `reason` is a short message (intended for developers or logging, not for end users) indicating the reason the request was rejected. It should return an *`HttpResponseForbidden`*.

`django.views.csrf.csrf_failure()` accepts an additional `template_name` parameter that defaults to `'403_csrf.html'`. If a template with that name exists, it will be used to render the page.

The `template_name` parameter and the behavior of searching for a template called `403_csrf.html` were added to `csrf_failure()`.

CSRF_HEADER_NAME

Default: `'HTTP_X_CSRFTOKEN'`

The name of the request header used for CSRF authentication.

As with other HTTP headers in `request.META`, the header name received from the server is normalized by converting all characters to uppercase, replacing any hyphens with underscores, and adding an `'HTTP_'` prefix to the name. For example, if your client sends a `'X-XSRF-TOKEN'` header, the setting should be `'HTTP_X_XSRF_TOKEN'`.

CSRF_TRUSTED_ORIGINS

Default: `[]` (Empty list)

A list of hosts which are trusted origins for unsafe requests (e.g. `POST`). For a *`secure`* unsafe request, Django's CSRF protection requires that the request have a `Referer` header that matches the origin present in the `Host` header. This prevents, for example, a `POST` request from `subdomain.example.com` from succeeding against `api.example.com`. If you need cross-origin unsafe requests over HTTPS, continuing the example, add `"subdomain.example.com"` to this list. The setting also supports subdomains, so you could add `".example.com"`, for example, to allow access from all subdomains of `example.com`.

DATABASES

Default: `{}` (Empty dictionary)

A dictionary containing the settings for all databases to be used with Django. It is a nested dictionary whose contents map a database alias to a dictionary containing the options for an individual database.

The *`DATABASES`* setting must configure a `default` database; any number of additional databases may also be specified.

The simplest possible settings file is for a single-database setup using SQLite. This can be configured using the following:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'mydatabase',
    }
}
```

When connecting to other database backends, such as MySQL, Oracle, or PostgreSQL, additional connection parameters will be required. See the *ENGINE* setting below on how to specify other database types. This example is for PostgreSQL:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'mydatabase',
        'USER': 'mydatabaseuser',
        'PASSWORD': 'mypassword',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
}
```

The following inner options that may be required for more complex configurations are available:

ATOMIC_REQUESTS

Default: `False`

Set this to `True` to wrap each view in a transaction on this database. See *Tying transactions to HTTP requests*.

AUTOCOMMIT

Default: `True`

Set this to `False` if you want to *disable Django's transaction management* and implement your own.

ENGINE

Default: `''` (Empty string)

The database backend to use. The built-in database backends are:

- `'django.db.backends.postgresql'`
- `'django.db.backends.mysql'`
- `'django.db.backends.sqlite3'`
- `'django.db.backends.oracle'`

You can use a database backend that doesn't ship with Django by setting `ENGINE` to a fully-qualified path (i.e. `mypackage.backends.whatever`).

The `django.db.backends.postgresql` backend is named `django.db.backends.postgresql_psycopg2` in older releases. For backwards compatibility, the old name still works in newer versions.

HOST

Default: `''` (Empty string)

Which host to use when connecting to the database. An empty string means localhost. Not used with SQLite.

If this value starts with a forward slash (`'/'`) and you're using MySQL, MySQL will connect via a Unix socket to the specified socket. For example:

```
"HOST": '/var/run/mysql'
```

If you're using MySQL and this value *doesn't* start with a forward slash, then this value is assumed to be the host.

If you're using PostgreSQL, by default (empty *HOST*), the connection to the database is done through UNIX domain sockets ('local' lines in `pg_hba.conf`). If your UNIX domain socket is not in the standard location, use the same value of `unix_socket_directory` from `postgresql.conf`. If you want to connect through TCP sockets, set *HOST* to 'localhost' or '127.0.0.1' ('host' lines in `pg_hba.conf`). On Windows, you should always define *HOST*, as UNIX domain sockets are not available.

NAME

Default: `''` (Empty string)

The name of the database to use. For SQLite, it's the full path to the database file. When specifying the path, always use forward slashes, even on Windows (e.g. `C:/homes/user/mysite/sqlite3.db`).

CONN_MAX_AGE

Default: `0`

The lifetime of a database connection, in seconds. Use `0` to close database connections at the end of each request — Django's historical behavior — and `None` for unlimited persistent connections.

OPTIONS

Default: `{}` (Empty dictionary)

Extra parameters to use when connecting to the database. Available parameters vary depending on your database backend.

Some information on available parameters can be found in the Database Backends documentation. For more information, consult your backend module's own documentation.

PASSWORD

Default: `''` (Empty string)

The password to use when connecting to the database. Not used with SQLite.

PORT

Default: `''` (Empty string)

The port to use when connecting to the database. An empty string means the default port. Not used with SQLite.

TIME_ZONE

Default: None

A string representing the time zone for datetimes stored in this database (assuming that it doesn't support time zones) or None. The same values are accepted as in the general *TIME_ZONE* setting.

This allows interacting with third-party databases that store datetimes in local time rather than UTC. To avoid issues around DST changes, you shouldn't set this option for databases managed by Django.

Setting this option requires installing pytz.

When *USE_TZ* is True and the database doesn't support time zones (e.g. SQLite, MySQL, Oracle), Django reads and writes datetimes in local time according to this option if it is set and in UTC if it isn't.

When *USE_TZ* is True and the database supports time zones (e.g. PostgreSQL), it is an error to set this option.

Before Django 1.9, the PostgreSQL database backend accepted an undocumented TIME_ZONE option, which caused data corruption.

When *USE_TZ* is False, it is an error to set this option.

USER

Default: '' (Empty string)

The username to use when connecting to the database. Not used with SQLite.

TEST

Default: {} (Empty dictionary)

A dictionary of settings for test databases; for more details about the creation and use of test databases, see *The test database*.

Here's an example with a test database configuration:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'USER': 'mydatabaseuser',
        'NAME': 'mydatabase',
        'TEST': {
            'NAME': 'mytestdatabase',
        },
    },
}
```

The following keys in the TEST dictionary are available:

CHARSET Default: None

The character set encoding used to create the test database. The value of this string is passed directly through to the database, so its format is backend-specific.

Supported by the PostgreSQL (postgresql) and MySQL (mysql) backends.

COLLATION Default: `None`

The collation order to use when creating the test database. This value is passed directly to the backend, so its format is backend-specific.

Only supported for the `mysql` backend (see the MySQL manual for details).

DEPENDENCIES Default: `['default']`, for all databases other than `default`, which has no dependencies.

The creation-order dependencies of the database. See the documentation on *controlling the creation order of test databases* for details.

MIRROR Default: `None`

The alias of the database that this database should mirror during testing.

This setting exists to allow for testing of primary/replica (referred to as master/slave by some databases) configurations of multiple databases. See the documentation on *testing primary/replica configurations* for details.

NAME Default: `None`

The name of database to use when running the test suite.

If the default value (`None`) is used with the SQLite database engine, the tests will use a memory resident database. For all other database engines the test database will use the name `'test_' + DATABASE_NAME`.

See *The test database*.

SERIALIZE Boolean value to control whether or not the default test runner serializes the database into an in-memory JSON string before running tests (used to restore the database state between tests if you don't have transactions). You can set this to `False` to speed up creation time if you don't have any test classes with *serialized_rollback=True*.

CREATE_DB Default: `True`

This is an Oracle-specific setting.

If it is set to `False`, the test tablespaces won't be automatically created at the beginning of the tests or dropped at the end.

CREATE_USER Default: `True`

This is an Oracle-specific setting.

If it is set to `False`, the test user won't be automatically created at the beginning of the tests and dropped at the end.

USER Default: `None`

This is an Oracle-specific setting.

The username to use when connecting to the Oracle database that will be used when running tests. If not provided, Django will use `'test_' + USER`.

PASSWORD Default: `None`

This is an Oracle-specific setting.

The password to use when connecting to the Oracle database that will be used when running tests. If not provided, Django will generate a random password.

Older versions used a hardcoded default password. This was also changed in 1.9.11 and 1.8.16 to fix possible security implications.

TBLSPACE Default: `None`

This is an Oracle-specific setting.

The name of the tablespace that will be used when running tests. If not provided, Django will use `'test_' + USER`.

TBLSPACE_TMP Default: `None`

This is an Oracle-specific setting.

The name of the temporary tablespace that will be used when running tests. If not provided, Django will use `'test_' + USER + '_temp'`.

DATAFILE Default: `None`

This is an Oracle-specific setting.

The name of the datafile to use for the TBLSPACE. If not provided, Django will use `TBLSPACE + '.dbf'`.

DATAFILE_TMP Default: `None`

This is an Oracle-specific setting.

The name of the datafile to use for the TBLSPACE_TMP. If not provided, Django will use `TBLSPACE_TMP + '.dbf'`.

DATAFILE_MAXSIZE Default: `'500M'`

This is an Oracle-specific setting.

The maximum size that the DATAFILE is allowed to grow to.

DATAFILE_TMP_MAXSIZE Default: `'500M'`

This is an Oracle-specific setting.

The maximum size that the DATAFILE_TMP is allowed to grow to.

DATA_UPLOAD_MAX_MEMORY_SIZE

Default: `2621440` (i.e. 2.5 MB).

The maximum size in bytes that a request body may be before a *SuspiciousOperation* (`RequestDataTooBig`) is raised. The check is done when accessing `request.body` or `request.POST` and is calculated against the total request size excluding any file upload data. You can set this to `None` to disable the check. Applications that are expected to receive unusually large form posts should tune this setting.

The amount of request data is correlated to the amount of memory needed to process the request and populate the GET and POST dictionaries. Large requests could be used as a denial-of-service attack vector if left unchecked. Since web servers don't typically perform deep request inspection, it's not possible to perform a similar check at that level.

See also *FILE_UPLOAD_MAX_MEMORY_SIZE*.

DATA_UPLOAD_MAX_NUMBER_FIELDS

Default: `1000`

The maximum number of parameters that may be received via GET or POST before a *SuspiciousOperation* (`TooManyFields`) is raised. You can set this to `None` to disable the check. Applications that are expected to receive an unusually large number of form fields should tune this setting.

The number of request parameters is correlated to the amount of time needed to process the request and populate the GET and POST dictionaries. Large requests could be used as a denial-of-service attack vector if left unchecked. Since web servers don't typically perform deep request inspection, it's not possible to perform a similar check at that level.

DATABASE_ROUTERS

Default: `[]` (Empty list)

The list of routers that will be used to determine which database to use when performing a database query.

See the documentation on *automatic database routing in multi database configurations*.

DATE_FORMAT

Default: `'N j, Y'` (e.g. `Feb. 4, 2003`)

The default formatting to use for displaying date fields in any part of the system. Note that if *USE_L10N* is set to `True`, then the locale-dictated format has higher precedence and will be applied instead. See *allowed date format strings*.

See also *DATETIME_FORMAT*, *TIME_FORMAT* and *SHORT_DATE_FORMAT*.

DATE_INPUT_FORMATS

Default:

```
[
    '%Y-%m-%d', '%m/%d/%Y', '%m/%d/%y', # '2006-10-25', '10/25/2006', '10/25/06'
    '%b %d %Y', '%b %d, %Y',            # 'Oct 25 2006', 'Oct 25, 2006'
    '%d %b %Y', '%d %b, %Y',            # '25 Oct 2006', '25 Oct, 2006'
    '%B %d %Y', '%B %d, %Y',            # 'October 25 2006', 'October 25, 2006'
    '%d %B %Y', '%d %B, %Y',            # '25 October 2006', '25 October, 2006'
]
```

A list of formats that will be accepted when inputting data on a date field. Formats will be tried in order, using the first valid one. Note that these format strings use Python's datetime module syntax, not the format strings from the *date* template filter.

When *USE_L10N* is `True`, the locale-dictated format has higher precedence and will be applied instead.

See also *DATETIME_INPUT_FORMATS* and *TIME_INPUT_FORMATS*.

DATETIME_FORMAT

Default: 'N j, Y, P' (e.g. Feb. 4, 2003, 4 p.m.)

The default formatting to use for displaying datetime fields in any part of the system. Note that if *USE_L10N* is set to True, then the locale-dictated format has higher precedence and will be applied instead. See *allowed date format strings*.

See also *DATE_FORMAT*, *TIME_FORMAT* and *SHORT_DATETIME_FORMAT*.

DATETIME_INPUT_FORMATS

Default:

```
[
    '%Y-%m-%d %H:%M:%S',      # '2006-10-25 14:30:59'
    '%Y-%m-%d %H:%M:%S.%f',   # '2006-10-25 14:30:59.000200'
    '%Y-%m-%d %H:%M',         # '2006-10-25 14:30'
    '%Y-%m-%d',               # '2006-10-25'
    '%m/%d/%Y %H:%M:%S',      # '10/25/2006 14:30:59'
    '%m/%d/%Y %H:%M:%S.%f',   # '10/25/2006 14:30:59.000200'
    '%m/%d/%Y %H:%M',         # '10/25/2006 14:30'
    '%m/%d/%Y',               # '10/25/2006'
    '%m/%d/%y %H:%M:%S',      # '10/25/06 14:30:59'
    '%m/%d/%y %H:%M:%S.%f',   # '10/25/06 14:30:59.000200'
    '%m/%d/%y %H:%M',         # '10/25/06 14:30'
    '%m/%d/%y',               # '10/25/06'
]
```

A list of formats that will be accepted when inputting data on a datetime field. Formats will be tried in order, using the first valid one. Note that these format strings use Python's datetime module syntax, not the format strings from the *date* template filter.

When *USE_L10N* is True, the locale-dictated format has higher precedence and will be applied instead.

See also *DATE_INPUT_FORMATS* and *TIME_INPUT_FORMATS*.

DEBUG

Default: False

A boolean that turns on/off debug mode.

Never deploy a site into production with *DEBUG* turned on.

Did you catch that? NEVER deploy a site into production with *DEBUG* turned on.

One of the main features of debug mode is the display of detailed error pages. If your app raises an exception when *DEBUG* is True, Django will display a detailed traceback, including a lot of metadata about your environment, such as all the currently defined Django settings (from settings.py).

As a security measure, Django will *not* include settings that might be sensitive, such as *SECRET_KEY*. Specifically, it will exclude any setting whose name includes any of the following:

- 'API'
- 'KEY'
- 'PASS'
- 'SECRET'

- 'SIGNATURE'
- 'TOKEN'

Note that these are *partial* matches. 'PASS' will also match PASSWORD, just as 'TOKEN' will also match TOKENIZED and so on.

Still, note that there are always going to be sections of your debug output that are inappropriate for public consumption. File paths, configuration options and the like all give attackers extra information about your server.

It is also important to remember that when running with *DEBUG* turned on, Django will remember every SQL query it executes. This is useful when you're debugging, but it'll rapidly consume memory on a production server.

Finally, if *DEBUG* is False, you also need to properly set the *ALLOWED_HOSTS* setting. Failing to do so will result in all requests being returned as "Bad Request (400)".

: The default settings.py file created by *django-admin startproject* sets DEBUG = True for convenience.

DEBUG_PROPAGATE_EXCEPTIONS

Default: False

If set to True, Django's normal exception handling of view functions will be suppressed, and exceptions will propagate upwards. This can be useful for some test setups, and should never be used on a live site.

DECIMAL_SEPARATOR

Default: '.' (Dot)

Default decimal separator used when formatting decimal numbers.

Note that if *USE_L10N* is set to True, then the locale-dictated format has higher precedence and will be applied instead.

See also *NUMBER_GROUPING*, *THOUSAND_SEPARATOR* and *USE_THOUSAND_SEPARATOR*.

DEFAULT_CHARSET

Default: 'utf-8'

Default charset to use for all HttpResponse objects, if a MIME type isn't manually specified. Used with *DEFAULT_CONTENT_TYPE* to construct the Content-Type header.

DEFAULT_CONTENT_TYPE

Default: 'text/html'

Default content type to use for all HttpResponse objects, if a MIME type isn't manually specified. Used with *DEFAULT_CHARSET* to construct the Content-Type header.

DEFAULT_EXCEPTION_REPORTER_FILTER

Default: *'django.views.debug.SafeExceptionReporterFilter'*

Default exception reporter filter class to be used if none has been assigned to the *HttpRequest* instance yet. See *Filtering error reports*.

DEFAULT_FILE_STORAGE

Default: *'django.core.files.storage.FileSystemStorage'*

Default file storage class to be used for any file-related operations that don't specify a particular storage system. See Managing files.

DEFAULT_FROM_EMAIL

Default: `'webmaster@localhost'`

Default email address to use for various automated correspondence from the site manager(s). This doesn't include error messages sent to *ADMINS* and *MANAGERS*; for that, see *SERVER_EMAIL*.

DEFAULT_INDEX_TABLESPACE

Default: `''` (Empty string)

Default tablespace to use for indexes on fields that don't specify one, if the backend supports it (see Tablespaces).

DEFAULT_TABLESPACE

Default: `''` (Empty string)

Default tablespace to use for models that don't specify one, if the backend supports it (see Tablespaces).

DISALLOWED_USER_AGENTS

Default: `[]` (Empty list)

List of compiled regular expression objects representing User-Agent strings that are not allowed to visit any page, systemwide. Use this for bad robots/crawlers. This is only used if `CommonMiddleware` is installed (see Middleware).

EMAIL_BACKEND

Default: *'django.core.mail.backends.smtp.EmailBackend'*

The backend to use for sending emails. For the list of available backends see Sending email.

EMAIL_FILE_PATH

Default: Not defined

The directory used by the `file` email backend to store output files.

EMAIL_HOST

Default: `'localhost'`

The host to use for sending email.

See also *EMAIL_PORT*.

EMAIL_HOST_PASSWORD

Default: `''` (Empty string)

Password to use for the SMTP server defined in *EMAIL_HOST*. This setting is used in conjunction with *EMAIL_HOST_USER* when authenticating to the SMTP server. If either of these settings is empty, Django won't attempt authentication.

See also *EMAIL_HOST_USER*.

EMAIL_HOST_USER

Default: `''` (Empty string)

Username to use for the SMTP server defined in *EMAIL_HOST*. If empty, Django won't attempt authentication.

See also *EMAIL_HOST_PASSWORD*.

EMAIL_PORT

Default: `25`

Port to use for the SMTP server defined in *EMAIL_HOST*.

EMAIL_SUBJECT_PREFIX

Default: `'[Django] '`

Subject-line prefix for email messages sent with `django.core.mail.mail_admins` or `django.core.mail.mail_managers`. You'll probably want to include the trailing space.

EMAIL_USE_TLS

Default: `False`

Whether to use a TLS (secure) connection when talking to the SMTP server. This is used for explicit TLS connections, generally on port 587. If you are experiencing hanging connections, see the implicit TLS setting *EMAIL_USE_SSL*.

EMAIL_USE_SSL

Default: `False`

Whether to use an implicit TLS (secure) connection when talking to the SMTP server. In most email documentation this type of TLS connection is referred to as SSL. It is generally used on port 465. If you are experiencing problems, see the explicit TLS setting *EMAIL_USE_TLS*.

Note that *EMAIL_USE_TLS*/*EMAIL_USE_SSL* are mutually exclusive, so only set one of those settings to `True`.

EMAIL_SSL_CERTFILE

Default: `None`

If *EMAIL_USE_SSL* or *EMAIL_USE_TLS* is `True`, you can optionally specify the path to a PEM-formatted certificate chain file to use for the SSL connection.

EMAIL_SSL_KEYFILE

Default: `None`

If *EMAIL_USE_SSL* or *EMAIL_USE_TLS* is `True`, you can optionally specify the path to a PEM-formatted private key file to use for the SSL connection.

Note that setting *EMAIL_SSL_CERTFILE* and *EMAIL_SSL_KEYFILE* doesn't result in any certificate checking. They're passed to the underlying SSL connection. Please refer to the documentation of Python's `ssl.wrap_socket()` function for details on how the certificate chain file and private key file are handled.

EMAIL_TIMEOUT

Default: `None`

Specifies a timeout in seconds for blocking operations like the connection attempt.

FILE_CHARSET

Default: `'utf-8'`

The character encoding used to decode any files read from disk. This includes template files and initial SQL data files.

FILE_UPLOAD_HANDLERS

Default:

```
[
    'django.core.files.uploadhandler.MemoryFileUploadHandler',
    'django.core.files.uploadhandler.TemporaryFileUploadHandler',
]
```

A list of handlers to use for uploading. Changing this setting allows complete customization – even replacement – of Django's upload process.

See Managing files for details.

FILE_UPLOAD_MAX_MEMORY_SIZE

Default: `2621440` (i.e. 2.5 MB).

The maximum size (in bytes) that an upload will be before it gets streamed to the file system. See Managing files for details.

See also *DATA_UPLOAD_MAX_MEMORY_SIZE*.

FILE_UPLOAD_DIRECTORY_PERMISSIONS

Default: `None`

The numeric mode to apply to directories created in the process of uploading files.

This setting also determines the default permissions for collected static directories when using the *collectstatic* management command. See *collectstatic* for details on overriding it.

This value mirrors the functionality and caveats of the *FILE_UPLOAD_PERMISSIONS* setting.

FILE_UPLOAD_PERMISSIONS

Default: `None`

The numeric mode (i.e. `0o644`) to set newly uploaded files to. For more information about what these modes mean, see the documentation for `os.chmod()`.

If this isn't given or is `None`, you'll get operating-system dependent behavior. On most platforms, temporary files will have a mode of `0o600`, and files saved from memory will be saved using the system's standard umask.

For security reasons, these permissions aren't applied to the temporary files that are stored in *FILE_UPLOAD_TEMP_DIR*.

This setting also determines the default permissions for collected static files when using the *collectstatic* management command. See *collectstatic* for details on overriding it.

: Always prefix the mode with a 0.
If you're not familiar with file modes, please note that the leading `0` is very important: it indicates an octal number, which is the way that modes must be specified. If you try to use `644`, you'll get totally incorrect behavior.

FILE_UPLOAD_TEMP_DIR

Default: `None`

The directory to store data to (typically files larger than *FILE_UPLOAD_MAX_MEMORY_SIZE*) temporarily while uploading files. If `None`, Django will use the standard temporary directory for the operating system. For example, this will default to `/tmp` on *nix-style operating systems.

See Managing files for details.

FIRST_DAY_OF_WEEK

Default: `0` (Sunday)

A number representing the first day of the week. This is especially useful when displaying a calendar. This value is only used when not using format internationalization, or when a format cannot be found for the current locale.

The value must be an integer from 0 to 6, where 0 means Sunday, 1 means Monday and so on.

FIXTURE_DIRS

Default: `[]` (Empty list)

List of directories searched for fixture files, in addition to the `fixtures` directory of each application, in search order.

Note that these paths should use Unix-style forward slashes, even on Windows.

See *Providing initial data with fixtures* and *Fixture loading*.

FORCE_SCRIPT_NAME

Default: `None`

If not `None`, this will be used as the value of the `SCRIPT_NAME` environment variable in any HTTP request. This setting can be used to override the server-provided value of `SCRIPT_NAME`, which may be a rewritten version of the preferred value or not supplied at all. It is also used by *`django.setup()`* to set the URL resolver script prefix outside of the request/response cycle (e.g. in management commands and standalone scripts) to generate correct URLs when `SCRIPT_NAME` is not `/`.

The setting's use in *`django.setup()`* was added.

FORMAT_MODULE_PATH

Default: `None`

A full Python path to a Python package that contains format definitions for project locales. If not `None`, Django will check for a `formats.py` file, under the directory named as the current locale, and will use the formats defined in this file.

For example, if *`FORMAT_MODULE_PATH`* is set to `mysite.formats`, and current language is `en` (English), Django will expect a directory tree like:

```
mysite/
    formats/
        __init__.py
        en/
            __init__.py
            formats.py
```

You can also set this setting to a list of Python paths, for example:

```
FORMAT_MODULE_PATH = [
    'mysite.formats',
    'some_app.formats',
]
```

When Django searches for a certain format, it will go through all given Python paths until it finds a module that actually defines the given format. This means that formats defined in packages farther up in the list will take precedence over the same formats in packages farther down.

Available formats are *`DATE_FORMAT`*, *`TIME_FORMAT`*, *`DATETIME_FORMAT`*, *`YEAR_MONTH_FORMAT`*, *`MONTH_DAY_FORMAT`*, *`SHORT_DATE_FORMAT`*, *`SHORT_DATETIME_FORMAT`*, *`FIRST_DAY_OF_WEEK`*, *`DECIMAL_SEPARATOR`*, *`THOUSAND_SEPARATOR`* and *`NUMBER_GROUPING`*.

IGNORABLE_404_URLS

Default: `[]` (Empty list)

List of compiled regular expression objects describing URLs that should be ignored when reporting HTTP 404 errors via email (see Error reporting). Regular expressions are matched against *`request's full paths`* (including query string, if any). Use this if your site does not provide a commonly requested file such as `favicon.ico` or `robots.txt`, or if it gets hammered by script kiddies.

This is only used if *BrokenLinkEmailsMiddleware* is enabled (see Middleware).

INSTALLED_APPS

Default: [] (Empty list)

A list of strings designating all applications that are enabled in this Django installation. Each string should be a dotted Python path to:

- an application configuration class (preferred), or
- a package containing an application.

Learn more about application configurations.

Use the application registry for introspection

Your code should never access *INSTALLED_APPS* directly. Use *django.apps.apps* instead.

Application names and labels must be unique in INSTALLED_APPS

Application *names* — the dotted Python path to the application package — must be unique. There is no way to include the same application twice, short of duplicating its code under another name.

Application *labels* — by default the final part of the name — must be unique too. For example, you can't include both `django.contrib.auth` and `myproject.auth`. However, you can relabel an application with a custom configuration that defines a different *label*.

These rules apply regardless of whether *INSTALLED_APPS* references application configuration classes or application packages.

When several applications provide different versions of the same resource (template, static file, management command, translation), the application listed first in *INSTALLED_APPS* has precedence.

INTERNAL_IPS

Default: [] (Empty list)

A list of IP addresses, as strings, that:

- Allow the *debug()* context processor to add some variables to the template context.
- Can use the *admindocs bookmarklets* even if not logged in as a staff user.
- Are marked as "internal" (as opposed to "EXTERNAL") in *AdminEmailHandler* emails.

LANGUAGE_CODE

Default: 'en-us'

A string representing the language code for this installation. This should be in standard *language ID format*. For example, U.S. English is "en-us". See also the list of language identifiers and Internationalization and localization.

USE_I18N must be active for this setting to have any effect.

It serves two purposes:

- If the locale middleware isn't in use, it decides which translation is served to all users.
- If the locale middleware is active, it provides a fallback language in case the user's preferred language can't be determined or is not supported by the website. It also provides the fallback translation when a translation for a given literal doesn't exist for the user's preferred language.

See *How Django discovers language preference* for more details.

LANGUAGE_COOKIE_AGE

Default: `None` (expires at browser close)

The age of the language cookie, in seconds.

LANGUAGE_COOKIE_DOMAIN

Default: `None`

The domain to use for the language cookie. Set this to a string such as `".example.com"` (note the leading dot!) for cross-domain cookies, or use `None` for a standard domain cookie.

Be cautious when updating this setting on a production site. If you update this setting to enable cross-domain cookies on a site that previously used standard domain cookies, existing user cookies that have the old domain will not be updated. This will result in site users being unable to switch the language as long as these cookies persist. The only safe and reliable option to perform the switch is to change the language cookie name permanently (via the *LANGUAGE_COOKIE_NAME* setting) and to add a middleware that copies the value from the old cookie to a new one and then deletes the old one.

LANGUAGE_COOKIE_NAME

Default: `'django_language'`

The name of the cookie to use for the language cookie. This can be whatever you want (as long as it's different from the other cookie names in your application). See Internationalization and localization.

LANGUAGE_COOKIE_PATH

Default: `'/'`

The path set on the language cookie. This should either match the URL path of your Django installation or be a parent of that path.

This is useful if you have multiple Django instances running under the same hostname. They can use different cookie paths and each instance will only see its own language cookie.

Be cautious when updating this setting on a production site. If you update this setting to use a deeper path than it previously used, existing user cookies that have the old path will not be updated. This will result in site users being unable to switch the language as long as these cookies persist. The only safe and reliable option to perform the switch is to change the language cookie name permanently (via the *LANGUAGE_COOKIE_NAME* setting), and to add a middleware that copies the value from the old cookie to a new one and then deletes the one.

LANGUAGES

Default: A list of all available languages. This list is continually growing and including a copy here would inevitably become rapidly out of date. You can see the current list of translated languages by looking in `django/conf/global_settings.py` (or view the online source).

The list is a list of two-tuples in the format (*language code*, `language name`) – for example, `('ja', 'Japanese')`. This specifies which languages are available for language selection. See Internationalization and localization.

Generally, the default value should suffice. Only set this setting if you want to restrict language selection to a subset of the Django-provided languages.

If you define a custom *LANGUAGES* setting, you can mark the language names as translation strings using the *ugettext_lazy()* function.

Here's a sample settings file:

```
from django.utils.translation import ugettext_lazy as _

LANGUAGES = [
    ('de', _('German')),
    ('en', _('English')),
]
```

LOCALE_PATHS

Default: `[]` (Empty list)

A list of directories where Django looks for translation files. See *How Django discovers translations*.

Example:

```
LOCALE_PATHS = [
    '/home/www/project/common_files/locale',
    '/var/local/translations/locale',
]
```

Django will look within each of these paths for the `<locale_code>/LC_MESSAGES` directories containing the actual translation files.

LOGGING

Default: A logging configuration dictionary.

A data structure containing configuration information. The contents of this data structure will be passed as the argument to the configuration method described in *LOGGING_CONFIG*.

Among other things, the default logging configuration passes HTTP 500 server errors to an email log handler when *DEBUG* is `False`. See also *Configuring logging*.

You can see the default logging configuration by looking in `django/utils/log.py` (or view the online source).

LOGGING_CONFIG

Default: `'logging.config.dictConfig'`

A path to a callable that will be used to configure logging in the Django project. Points at a instance of Python's dictConfig configuration method by default.

If you set *LOGGING_CONFIG* to `None`, the logging configuration process will be skipped.

MANAGERS

Default: `[]` (Empty list)

A list in the same format as *ADMINS* that specifies who should get broken link notifications when *BrokenLinkEmailsMiddleware* is enabled.

MEDIA_ROOT

Default: `''` (Empty string)

Absolute filesystem path to the directory that will hold user-uploaded files.

Example: `"/var/www/example.com/media/"`

See also *MEDIA_URL*.

: *MEDIA_ROOT* and *STATIC_ROOT* must have different values. Before *STATIC_ROOT* was introduced, it was common to rely or fallback on *MEDIA_ROOT* to also serve static files; however, since this can have serious security implications, there is a validation check to prevent it.

MEDIA_URL

Default: `''` (Empty string)

URL that handles the media served from *MEDIA_ROOT*, used for managing stored files. It must end in a slash if set to a non-empty value. You will need to *configure these files to be served* in both development and production environments.

If you want to use `{{ MEDIA_URL }}` in your templates, add `'django.template.context_processors.media'` in the `'context_processors'` option of *TEMPLATES*.

Example: `"http://media.example.com/"`

: There are security risks if you are accepting uploaded content from untrusted users! See the security guide's topic on *User-uploaded content* for mitigation details.

: *MEDIA_URL* and *STATIC_URL* must have different values. See *MEDIA_ROOT* for more details.

MIDDLEWARE

Default:: `None`

A list of middleware to use. See Middleware.

MIDDLEWARE_CLASSES

1.10 : Old-style middleware that uses `settings.MIDDLEWARE_CLASSES` are deprecated. *Adapt old, custom middleware* and use the *MIDDLEWARE* setting.

Default:

```
[
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
]
```

A list of middleware classes to use. This was the default setting used in Django 1.9 and earlier. Django 1.10 introduced a new style of middleware. If you have an older project using this setting you should *update any middleware you've written yourself* to the new style and then use the *MIDDLEWARE* setting.

MIGRATION_MODULES

Default: `{}` (Empty dictionary)

A dictionary specifying the package where migration modules can be found on a per-app basis. The default value of this setting is an empty dictionary, but the default package name for migration modules is `migrations`.

Example:

```
{'blog': 'blog.db_migrations'}
```

In this case, migrations pertaining to the `blog` app will be contained in the `blog.db_migrations` package.

If you provide the `app_label` argument, *makemigrations* will automatically create the package if it doesn't already exist.

When you supply `None` as a value for an app, Django will consider the app as an app without migrations regardless of an existing `migrations` submodule. This can be used, for example, in a test settings file to skip migrations while testing (tables will still be created for the apps' models). If this is used in your general project settings, remember to use the *migrate --run-syncdb* option if you want to create tables for the app.

MONTH_DAY_FORMAT

Default: `'F j'`

The default formatting to use for date fields on Django admin change-list pages – and, possibly, by other parts of the system – in cases when only the month and day are displayed.

For example, when a Django admin change-list page is being filtered by a date drilldown, the header for a given day displays the day and month. Different locales have different formats. For example, U.S. English would say "January 1," whereas Spanish might say "1 Enero."

Note that if *USE_L10N* is set to `True`, then the corresponding locale-dictated format has higher precedence and will be applied.

See *allowed date format strings*. See also *DATE_FORMAT*, *DATETIME_FORMAT*, *TIME_FORMAT* and *YEAR_MONTH_FORMAT*.

NUMBER_GROUPING

Default: `0`

Number of digits grouped together on the integer part of a number.

Common use is to display a thousand separator. If this setting is `0`, then no grouping will be applied to the number. If this setting is greater than `0`, then *THOUSAND_SEPARATOR* will be used as the separator between those groups.

Note that if *USE_L10N* is set to `True`, then the locale-dictated format has higher precedence and will be applied instead.

See also *DECIMAL_SEPARATOR*, *THOUSAND_SEPARATOR* and *USE_THOUSAND_SEPARATOR*.

PREPEND_WWW

Default: `False`

Whether to prepend the "www." subdomain to URLs that don't have it. This is only used if *CommonMiddleware* is installed (see Middleware). See also *APPEND_SLASH*.

ROOT_URLCONF

Default: Not defined

A string representing the full Python import path to your root URLconf. For example: `"mydjangoapps.urls"`. Can be overridden on a per-request basis by setting the attribute `urlconf` on the incoming `HttpRequest` object. See *How Django processes a request* for details.

SECRET_KEY

Default: `''` (Empty string)

A secret key for a particular Django installation. This is used to provide cryptographic signing, and should be set to a unique, unpredictable value.

django-admin startproject automatically adds a randomly-generated `SECRET_KEY` to each new project.

Django will refuse to start if *SECRET_KEY* is not set.

: Keep this value secret.
Running Django with a known *SECRET_KEY* defeats many of Django's security protections, and can lead to privilege escalation and remote code execution vulnerabilities.

The secret key is used for:

- All sessions if you are using any other session backend than `django.contrib.sessions.backends.cache`, or are using the default *get_session_auth_hash()*.
- All messages if you are using *CookieStorage* or *FallbackStorage*.
- All *password_reset()* tokens.
- Any usage of cryptographic signing, unless a different key is provided.

If you rotate your secret key, all of the above will be invalidated. Secret keys are not used for passwords of users and key rotation will not affect them.

: The default `settings.py` file created by *django-admin startproject* creates a unique `SECRET_KEY` for convenience.

SECURE_BROWSER_XSS_FILTER

Default: `False`

If `True`, the *`SecurityMiddleware`* sets the *X-XSS-Protection: 1; mode=block* header on all responses that do not already have it.

SECURE_CONTENT_TYPE_NOSNIFF

Default: `False`

If `True`, the *`SecurityMiddleware`* sets the *X-Content-Type-Options: nosniff* header on all responses that do not already have it.

SECURE_HSTS_INCLUDE_SUBDOMAINS

Default: `False`

If `True`, the *`SecurityMiddleware`* adds the `includeSubDomains` directive to the *HTTP Strict Transport Security* header. It has no effect unless *`SECURE_HSTS_SECONDS`* is set to a non-zero value.

: Setting this incorrectly can irreversibly (for the value of *`SECURE_HSTS_SECONDS`*) break your site. Read the *HTTP Strict Transport Security* documentation first.

SECURE_HSTS_SECONDS

Default: `0`

If set to a non-zero integer value, the *`SecurityMiddleware`* sets the *HTTP Strict Transport Security* header on all responses that do not already have it.

: Setting this incorrectly can irreversibly (for some time) break your site. Read the *HTTP Strict Transport Security* documentation first.

SECURE_PROXY_SSL_HEADER

Default: `None`

A tuple representing a HTTP header/value combination that signifies a request is secure. This controls the behavior of the request object's `is_secure()` method.

This takes some explanation. By default, `is_secure()` is able to determine whether a request is secure by looking at whether the requested URL uses "https://". This is important for Django's CSRF protection, and may be used by your own code or third-party apps.

If your Django app is behind a proxy, though, the proxy may be "swallowing" the fact that a request is HTTPS, using a non-HTTPS connection between the proxy and Django. In this case, `is_secure()` would always return `False` – even for requests that were made via HTTPS by the end user.

In this situation, you'll want to configure your proxy to set a custom HTTP header that tells Django whether the request came in via HTTPS, and you'll want to set `SECURE_PROXY_SSL_HEADER` so that Django knows what header to look for.

You'll need to set a tuple with two elements – the name of the header to look for and the required value. For example:

```
SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')
```

Here, we're telling Django that we trust the `X-Forwarded-Proto` header that comes from our proxy, and any time its value is `'https'`, then the request is guaranteed to be secure (i.e., it originally came in via HTTPS). Obviously, you should *only* set this setting if you control your proxy or have some other guarantee that it sets/strips this header appropriately.

Note that the header needs to be in the format as used by `request.META` – all caps and likely starting with `HTTP_`. (Remember, Django automatically adds `'HTTP_'` to the start of x-header names before making the header available in `request.META`.)

: You will probably open security holes in your site if you set this without knowing what you're doing. And if you fail to set it when you should. Seriously.
Make sure ALL of the following are true before setting this (assuming the values from the example above):

- Your Django app is behind a proxy.
- Your proxy strips the `X-Forwarded-Proto` header from all incoming requests. In other words, if end users include that header in their requests, the proxy will discard it.
- Your proxy sets the `X-Forwarded-Proto` header and sends it to Django, but only for requests that originally come in via HTTPS.

If any of those are not true, you should keep this setting set to `None` and find another way of determining HTTPS, perhaps via custom middleware.

SECURE_REDIRECT_EXEMPT

Default: `[]` (Empty list)

If a URL path matches a regular expression in this list, the request will not be redirected to HTTPS. If *SECURE_SSL_REDIRECT* is `False`, this setting has no effect.

SECURE_SSL_HOST

Default: `None`

If a string (e.g. `secure.example.com`), all SSL redirects will be directed to this host rather than the originally-requested host (e.g. `www.example.com`). If *SECURE_SSL_REDIRECT* is `False`, this setting has no effect.

SECURE_SSL_REDIRECT

Default: `False`

If `True`, the *SecurityMiddleware redirects* all non-HTTPS requests to HTTPS (except for those URLs matching a regular expression listed in *SECURE_REDIRECT_EXEMPT*).

: If turning this to `True` causes infinite redirects, it probably means your site is running behind a proxy and can't tell which requests are secure and which are not. Your proxy likely sets a header to indicate secure requests; you can correct the problem by finding out what that header is and configuring the *SECURE_PROXY_SSL_HEADER* setting accordingly.

SERIALIZATION_MODULES

Default: Not defined

A dictionary of modules containing serializer definitions (provided as strings), keyed by a string identifier for that serialization type. For example, to define a YAML serializer, use:

```
SERIALIZATION_MODULES = {'yaml': 'path.to.yaml_serializer'}
```

SERVER_EMAIL

Default: `'root@localhost'`

The email address that error messages come from, such as those sent to *ADMINS* and *MANAGERS*.

Why are my emails sent from a different address?

This address is used only for error messages. It is *not* the address that regular email messages sent with *send_mail()* come from; for that, see *DEFAULT_FROM_EMAIL*.

SHORT_DATE_FORMAT

Default: `'m/d/Y'` (e.g. `12/31/2003`)

An available formatting that can be used for displaying date fields on templates. Note that if *USE_L10N* is set to `True`, then the corresponding locale-dictated format has higher precedence and will be applied. See *allowed date format strings*.

See also *DATE_FORMAT* and *SHORT_DATETIME_FORMAT*.

SHORT_DATETIME_FORMAT

Default: `'m/d/Y P'` (e.g. `12/31/2003 4 p.m.`)

An available formatting that can be used for displaying datetime fields on templates. Note that if *USE_L10N* is set to `True`, then the corresponding locale-dictated format has higher precedence and will be applied. See *allowed date format strings*.

See also *DATE_FORMAT* and *SHORT_DATE_FORMAT*.

SIGNING_BACKEND

Default: `'django.core.signing.TimestampSigner'`

The backend used for signing cookies and other data.

See also the Cryptographic signing documentation.

SILENCED_SYSTEM_CHECKS

Default: `[]` (Empty list)

A list of identifiers of messages generated by the system check framework (i.e. `["models.W001"]`) that you wish to permanently acknowledge and ignore. Silenced checks will not be output to the console.

In older versions, silenced messages of ERROR level or higher were printed to the console.

See also the System check framework documentation.

TEMPLATES

Default: [] (Empty list)

A list containing the settings for all template engines to be used with Django. Each item of the list is a dictionary containing the options for an individual engine.

Here's a simple setup that tells the Django template engine to load templates from the templates subdirectory inside each installed application:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'APP_DIRS': True,
    },
]
```

The following options are available for all backends.

BACKEND

Default: Not defined

The template backend to use. The built-in template backends are:

- 'django.template.backends.django.DjangoTemplates'
- 'django.template.backends.jinja2.Jinja2'

You can use a template backend that doesn't ship with Django by setting BACKEND to a fully-qualified path (i.e. 'mypackage.whatever.Backend').

NAME

Default: see below

The alias for this particular template engine. It's an identifier that allows selecting an engine for rendering. Aliases must be unique across all configured template engines.

It defaults to the name of the module defining the engine class, i.e. the next to last piece of *BACKEND*, when it isn't provided. For example if the backend is 'mypackage.whatever.Backend' then its default name is 'whatever'.

DIRS

Default: [] (Empty list)

Directories where the engine should look for template source files, in search order.

APP_DIRS

Default: `False`

Whether the engine should look for template source files inside installed applications.

: The default `settings.py` file created by *`django-admin startproject`* sets `'APP_DIRS': True`.

OPTIONS

Default: `{}` (Empty dict)

Extra parameters to pass to the template backend. Available parameters vary depending on the template backend. See *`DjangoTemplates`* and *`Jinja2`* for the options of the built-in backends.

TEST_RUNNER

Default: `'django.test.runner.DiscoverRunner'`

The name of the class to use for starting the test suite. See *Using different testing frameworks*.

TEST_NON_SERIALIZED_APPS

Default: `[]` (Empty list)

In order to restore the database state between tests for `TransactionTestCase`s and database backends without transactions, Django will *serialize the contents of all apps* when it starts the test run so it can then reload from that copy before running tests that need it.

This slows down the startup time of the test runner; if you have apps that you know don't need this feature, you can add their full names in here (e.g. `'django.contrib.contenttypes'`) to exclude them from this serialization process.

THOUSAND_SEPARATOR

Default: `','` (Comma)

Default thousand separator used when formatting numbers. This setting is used only when *`USE_THOUSAND_SEPARATOR`* is `True` and *`NUMBER_GROUPING`* is greater than `0`.

Note that if *`USE_L10N`* is set to `True`, then the locale-dictated format has higher precedence and will be applied instead.

See also *`NUMBER_GROUPING`*, *`DECIMAL_SEPARATOR`* and *`USE_THOUSAND_SEPARATOR`*.

TIME_FORMAT

Default: `'P'` (e.g. `4 p.m.`)

The default formatting to use for displaying time fields in any part of the system. Note that if *`USE_L10N`* is set to `True`, then the locale-dictated format has higher precedence and will be applied instead. See *`allowed date format strings`*.

See also *`DATE_FORMAT`* and *`DATETIME_FORMAT`*.

TIME_INPUT_FORMATS

Default:

```
[
    '%H:%M:%S',     # '14:30:59'
    '%H:%M:%S.%f',  # '14:30:59.000200'
    '%H:%M',        # '14:30'
]
```

A list of formats that will be accepted when inputting data on a time field. Formats will be tried in order, using the first valid one. Note that these format strings use Python's datetime module syntax, not the format strings from the *date* template filter.

When *USE_L10N* is `True`, the locale-dictated format has higher precedence and will be applied instead.

See also *DATE_INPUT_FORMATS* and *DATETIME_INPUT_FORMATS*.

TIME_ZONE

Default: `'America/Chicago'`

A string representing the time zone for this installation, or `None`. See the list of time zones.

: Since Django was first released with the *TIME_ZONE* set to `'America/Chicago'`, the global setting (used if nothing is defined in your project's `settings.py`) remains `'America/Chicago'` for backwards compatibility. New project templates default to `'UTC'`.

Note that this isn't necessarily the time zone of the server. For example, one server may serve multiple Django-powered sites, each with a separate time zone setting.

When *USE_TZ* is `False`, this is the time zone in which Django will store all datetimes. When *USE_TZ* is `True`, this is the default time zone that Django will use to display datetimes in templates and to interpret datetimes entered in forms.

Django sets the `os.environ['TZ']` variable to the time zone you specify in the *TIME_ZONE* setting. Thus, all your views and models will automatically operate in this time zone. However, Django won't set the `TZ` environment variable under the following conditions:

- If you're using the manual configuration option as described in *manually configuring settings*, or
- If you specify `TIME_ZONE = None`. This will cause Django to fall back to using the system timezone. However, this is discouraged when *USE_TZ = True*, because it makes conversions between local time and UTC less reliable.

If Django doesn't set the `TZ` environment variable, it's up to you to ensure your processes are running in the correct environment.

: Django cannot reliably use alternate time zones in a Windows environment. If you're running Django on Windows, *TIME_ZONE* must be set to match the system time zone.

USE_ETAGS

Default: `False`

A boolean that specifies whether to output the `ETag` header. This saves bandwidth but slows down performance. This is used by the *CommonMiddleware* and in the cache framework.

USE_I18N

Default: `True`

A boolean that specifies whether Django's translation system should be enabled. This provides an easy way to turn it off, for performance. If this is set to `False`, Django will make some optimizations so as not to load the translation machinery.

See also *LANGUAGE_CODE*, *USE_L10N* and *USE_TZ*.

: The default `settings.py` file created by *django-admin startproject* includes `USE_I18N = True` for convenience.

USE_L10N

Default: `False`

A boolean that specifies if localized formatting of data will be enabled by default or not. If this is set to `True`, e.g. Django will display numbers and dates using the format of the current locale.

See also *LANGUAGE_CODE*, *USE_I18N* and *USE_TZ*.

: The default `settings.py` file created by *django-admin startproject* includes `USE_L10N = True` for convenience.

USE_THOUSAND_SEPARATOR

Default: `False`

A boolean that specifies whether to display numbers using a thousand separator. When *USE_L10N* is set to `True` and if this is also set to `True`, Django will use the values of *THOUSAND_SEPARATOR* and *NUMBER_GROUPING* to format numbers unless the locale already has an existing thousands separator. If there is a thousands separator in the locale format, it will have higher precedence and will be applied instead.

See also *DECIMAL_SEPARATOR*, *NUMBER_GROUPING* and *THOUSAND_SEPARATOR*.

USE_TZ

Default: `False`

A boolean that specifies if datetimes will be timezone-aware by default or not. If this is set to `True`, Django will use timezone-aware datetimes internally. Otherwise, Django will use naive datetimes in local time.

See also *TIME_ZONE*, *USE_I18N* and *USE_L10N*.

: The default `settings.py` file created by *django-admin startproject* includes `USE_TZ = True` for convenience.

USE_X_FORWARDED_HOST

Default: `False`

A boolean that specifies whether to use the `X-Forwarded-Host` header in preference to the `Host` header. This should only be enabled if a proxy which sets this header is in use.

This setting takes priority over *USE_X_FORWARDED_PORT*. Per **RFC 7239#page-7**, the `X-Forwarded-Host` header can include the port number, in which case you shouldn't use *USE_X_FORWARDED_PORT*.

USE_X_FORWARDED_PORT

Default: `False`

A boolean that specifies whether to use the `X-Forwarded-Port` header in preference to the `SERVER_PORT` `META` variable. This should only be enabled if a proxy which sets this header is in use.

USE_X_FORWARDED_HOST takes priority over this setting.

WSGI_APPLICATION

Default: `None`

The full Python path of the WSGI application object that Django's built-in servers (e.g. *runserver*) will use. The *django-admin startproject* management command will create a simple `wsgi.py` file with an `application` callable in it, and point this setting to that `application`.

If not set, the return value of `django.core.wsgi.get_wsgi_application()` will be used. In this case, the behavior of *runserver* will be identical to previous Django versions.

YEAR_MONTH_FORMAT

Default: `'F Y'`

The default formatting to use for date fields on Django admin change-list pages – and, possibly, by other parts of the system – in cases when only the year and month are displayed.

For example, when a Django admin change-list page is being filtered by a date drilldown, the header for a given month displays the month and the year. Different locales have different formats. For example, U.S. English would say "January 2006," whereas another locale might say "2006/January."

Note that if *USE_L10N* is set to `True`, then the corresponding locale-dictated format has higher precedence and will be applied.

See *allowed date format strings*. See also *DATE_FORMAT*, *DATETIME_FORMAT*, *TIME_FORMAT* and *MONTH_DAY_FORMAT*.

X_FRAME_OPTIONS

Default: `'SAMEORIGIN'`

The default value for the X-Frame-Options header used by *XFrameOptionsMiddleware*. See the clickjacking protection documentation.

6.18.2 Auth

Settings for *django.contrib.auth*.

AUTHENTICATION_BACKENDS

Default: `['django.contrib.auth.backends.ModelBackend']`

A list of authentication backend classes (as strings) to use when attempting to authenticate a user. See the *authentication backends documentation* for details.

AUTH_USER_MODEL

Default: `'auth.User'`

The model to use to represent a User. See *Substituting a custom User model*.

: You cannot change the AUTH_USER_MODEL setting during the lifetime of a project (i.e. once you have made and migrated models that depend on it) without serious effort. It is intended to be set at the project start, and the model it refers to must be available in the first migration of the app that it lives in. See *Substituting a custom User model* for more details.

LOGIN_REDIRECT_URL

Default: `'/accounts/profile/'`

The URL where requests are redirected after login when the `contrib.auth.login` view gets no `next` parameter.

This is used by the *login_required()* decorator, for example.

This setting also accepts *named URL patterns* which can be used to reduce configuration duplication since you don't have to define the URL in two places (`settings` and URLconf).

LOGIN_URL

Default: `'/accounts/login/'`

The URL where requests are redirected for login, especially when using the *login_required()* decorator.

This setting also accepts *named URL patterns* which can be used to reduce configuration duplication since you don't have to define the URL in two places (`settings` and URLconf).

LOGOUT_REDIRECT_URL

Default: `None`

The URL where requests are redirected after a user logs out using the *logout()* view (if the view doesn't get a `next_page` argument).

If `None`, no redirect will be performed and the logout view will be rendered.

This setting also accepts *named URL patterns* which can be used to reduce configuration duplication since you don't have to define the URL in two places (`settings` and URLconf).

PASSWORD_RESET_TIMEOUT_DAYS

Default: `3`

The number of days a password reset link is valid for. Used by the *django.contrib.auth* password reset mechanism.

PASSWORD_HASHERS

See *How Django stores passwords*.

Default:

```
[
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
    'django.contrib.auth.hashers.Argon2PasswordHasher',
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
    'django.contrib.auth.hashers.BCryptPasswordHasher',
]
```

The following hashers were removed from the defaults:

```
'django.contrib.auth.hashers.SHA1PasswordHasher'
'django.contrib.auth.hashers.MD5PasswordHasher'
'django.contrib.auth.hashers.UnsaltedSHA1PasswordHasher'
'django.contrib.auth.hashers.UnsaltedMD5PasswordHasher'
'django.contrib.auth.hashers.CryptPasswordHasher'
```

Consider using a *wrapped password hasher* to strengthen the hashes in your database. If that's not feasible, add this setting to your project and add back any hashers that you need.

Also, the `Argon2PasswordHasher` was added.

AUTH_PASSWORD_VALIDATORS

Default: `[]` (Empty list)

The list of validators that are used to check the strength of user's passwords. See *Password validation* for more details. By default, no validation is performed and all passwords are accepted.

6.18.3 Messages

Settings for *django.contrib.messages*.

MESSAGE_LEVEL

Default: `messages.INFO`

Sets the minimum message level that will be recorded by the messages framework. See *message levels* for more details.

Important

If you override `MESSAGE_LEVEL` in your settings file and rely on any of the built-in constants, you must import the constants module directly to avoid the potential for circular imports, e.g.:

```
from django.contrib.messages import constants as message_constants
MESSAGE_LEVEL = message_constants.DEBUG
```

If desired, you may specify the numeric values for the constants directly according to the values in the above *constants table*.

MESSAGE_STORAGE

Default: `'django.contrib.messages.storage.fallback.FallbackStorage'`

Controls where Django stores message data. Valid values are:

- `'django.contrib.messages.storage.fallback.FallbackStorage'`
- `'django.contrib.messages.storage.session.SessionStorage'`
- `'django.contrib.messages.storage.cookie.CookieStorage'`

See *message storage backends* for more details.

The backends that use cookies – *CookieStorage* and *FallbackStorage* – use the value of *SESSION_COOKIE_DOMAIN*, *SESSION_COOKIE_SECURE* and *SESSION_COOKIE_HTTPONLY* when setting their cookies.

MESSAGE_TAGS

Default:

```
{
    messages.DEBUG: 'debug',
    messages.INFO: 'info',
    messages.SUCCESS: 'success',
    messages.WARNING: 'warning',
    messages.ERROR: 'error',
}
```

This sets the mapping of message level to message tag, which is typically rendered as a CSS class in HTML. If you specify a value, it will extend the default. This means you only have to specify those values which you need to override. See *Displaying messages* above for more details.

Important

If you override `MESSAGE_TAGS` in your settings file and rely on any of the built-in constants, you must import the `constants` module directly to avoid the potential for circular imports, e.g.:

```
from django.contrib.messages import constants as message_constants
MESSAGE_TAGS = {message_constants.INFO: ''}
```

If desired, you may specify the numeric values for the constants directly according to the values in the above *constants table*.

6.18.4 Sessions

Settings for *django.contrib.sessions*.

SESSION_CACHE_ALIAS

Default: `'default'`

If you're using *cache-based session storage*, this selects the cache to use.

SESSION_COOKIE_AGE

Default: `1209600` (2 weeks, in seconds)

The age of session cookies, in seconds.

SESSION_COOKIE_DOMAIN

Default: `None`

The domain to use for session cookies. Set this to a string such as `".example.com"` (note the leading dot!) for cross-domain cookies, or use `None` for a standard domain cookie.

Be cautious when updating this setting on a production site. If you update this setting to enable cross-domain cookies on a site that previously used standard domain cookies, existing user cookies will be set to the old domain. This may result in them being unable to log in as long as these cookies persist.

This setting also affects cookies set by *`django.contrib.messages`*.

SESSION_COOKIE_HTTPONLY

Default: `True`

Whether to use `HTTPOnly` flag on the session cookie. If this is set to `True`, client-side JavaScript will not to be able to access the session cookie.

HTTPOnly is a flag included in a Set-Cookie HTTP response header. It is not part of the **RFC 2109** standard for cookies, and it isn't honored consistently by all browsers. However, when it is honored, it can be a useful way to mitigate the risk of client side script accessing the protected cookie data.

Turning it on makes it less trivial for an attacker to escalate a cross-site scripting vulnerability into full hijacking of a user's session. There's not much excuse for leaving this off, either: if your code depends on reading session cookies from JavaScript, you're probably doing it wrong.

SESSION_COOKIE_NAME

Default: `'sessionid'`

The name of the cookie to use for sessions. This can be whatever you want (as long as it's different from the other cookie names in your application).

SESSION_COOKIE_PATH

Default: `'/'`

The path set on the session cookie. This should either match the URL path of your Django installation or be parent of that path.

This is useful if you have multiple Django instances running under the same hostname. They can use different cookie paths, and each instance will only see its own session cookie.

SESSION_COOKIE_SECURE

Default: `False`

Whether to use a secure cookie for the session cookie. If this is set to `True`, the cookie will be marked as "secure," which means browsers may ensure that the cookie is only sent under an HTTPS connection.

Since it's trivial for a packet sniffer (e.g. Firesheep) to hijack a user's session if the session cookie is sent unencrypted, there's really no good excuse to leave this off. It will prevent you from using sessions on insecure requests and that's a good thing.

SESSION_ENGINE

Default: `'django.contrib.sessions.backends.db'`

Controls where Django stores session data. Included engines are:

- `'django.contrib.sessions.backends.db'`
- `'django.contrib.sessions.backends.file'`
- `'django.contrib.sessions.backends.cache'`
- `'django.contrib.sessions.backends.cached_db'`
- `'django.contrib.sessions.backends.signed_cookies'`

See *Configuring the session engine* for more details.

SESSION_EXPIRE_AT_BROWSER_CLOSE

Default: `False`

Whether to expire the session when the user closes their browser. See *Browser-length sessions vs. persistent sessions*.

SESSION_FILE_PATH

Default: `None`

If you're using file-based session storage, this sets the directory in which Django will store session data. When the default value (`None`) is used, Django will use the standard temporary directory for the system.

SESSION_SAVE_EVERY_REQUEST

Default: `False`

Whether to save the session data on every request. If this is `False` (default), then the session data will only be saved if it has been modified – that is, if any of its dictionary values have been assigned or deleted. Empty sessions won't be created, even if this setting is active.

SESSION_SERIALIZER

Default: `'django.contrib.sessions.serializers.JSONSerializer'`

Full import path of a serializer class to use for serializing session data. Included serializers are:

- 'django.contrib.sessions.serializers.PickleSerializer'
- 'django.contrib.sessions.serializers.JSONSerializer'

See *Session serialization* for details, including a warning regarding possible remote code execution when using *PickleSerializer*.

6.18.5 Sites

Settings for *django.contrib.sites*.

SITE_ID

Default: Not defined

The ID, as an integer, of the current site in the django_site database table. This is used so that application data can hook into specific sites and a single database can manage content for multiple sites.

6.18.6 Static Files

Settings for *django.contrib.staticfiles*.

STATIC_ROOT

Default: None

The absolute path to the directory where *collectstatic* will collect static files for deployment.

Example: "/var/www/example.com/static/"

If the staticfiles contrib app is enabled (as in the default project template), the *collectstatic* management command will collect static files into this directory. See the how-to on managing static files for more details about usage.

: This should be an initially empty destination directory for collecting your static files from their permanent locations into one directory for ease of deployment; it is **not** a place to store your static files permanently. You should do that in directories that will be found by staticfiles's *finders*, which by default, are 'static/' app sub-directories and any directories you include in *STATICFILES_DIRS*).

STATIC_URL

Default: None

URL to use when referring to static files located in *STATIC_ROOT*.

Example: "/static/" or "http://static.example.com/"

If not None, this will be used as the base path for *asset definitions* (the Media class) and the staticfiles app.

It must end in a slash if set to a non-empty value.

You may need to *configure these files to be served in development* and will definitely need to do so in production.

STATICFILES_DIRS

Default: `[]` (Empty list)

This setting defines the additional locations the staticfiles app will traverse if the `FileSystemFinder` finder is enabled, e.g. if you use the *`collectstatic`* or *`findstatic`* management command or use the static file serving view.

This should be set to a list of strings that contain full paths to your additional files directory(ies) e.g.:

```
STATICFILES_DIRS = [
    "/home/special.polls.com/polls/static",
    "/home/polls.com/polls/static",
    "/opt/webfiles/common",
]
```

Note that these paths should use Unix-style forward slashes, even on Windows (e.g. `"C:/Users/user/mysite/extra_static_content"`).

Prefixes (optional)

In case you want to refer to files in one of the locations with an additional namespace, you can **optionally** provide a prefix as `(prefix, path)` tuples, e.g.:

```
STATICFILES_DIRS = [
    # ...
    ("downloads", "/opt/webfiles/stats"),
]
```

For example, assuming you have *`STATIC_URL`* set to `'/static/'`, the *`collectstatic`* management command would collect the "stats" files in a `'downloads'` subdirectory of *`STATIC_ROOT`*.

This would allow you to refer to the local file `'/opt/webfiles/stats/polls_20101022.tar.gz'` with `'/static/downloads/polls_20101022.tar.gz'` in your templates, e.g.:

```
<a href="{% static "downloads/polls_20101022.tar.gz" %}">
```

STATICFILES_STORAGE

Default: `'django.contrib.staticfiles.storage.StaticFilesStorage'`

The file storage engine to use when collecting static files with the *`collectstatic`* management command.

A ready-to-use instance of the storage backend defined in this setting can be found at `django.contrib.staticfiles.storage.staticfiles_storage`.

For an example, see *Serving static files from a cloud service or CDN*.

STATICFILES_FINDERS

Default:

```
[
    'django.contrib.staticfiles.finders.FileSystemFinder',
    'django.contrib.staticfiles.finders.AppDirectoriesFinder',
]
```

The list of finder backends that know how to find static files in various locations.

The default will find files stored in the *STATICFILES_DIRS* setting (using `django.contrib.staticfiles.finders.FileSystemFinder`) and in a `static` subdirectory of each app (using `django.contrib.staticfiles.finders.AppDirectoriesFinder`). If multiple files with the same name are present, the first file that is found will be used.

One finder is disabled by default: `django.contrib.staticfiles.finders.DefaultStorageFinder`. If added to your *STATICFILES_FINDERS* setting, it will look for static files in the default file storage as defined by the *DEFAULT_FILE_STORAGE* setting.

: When using the `AppDirectoriesFinder` finder, make sure your apps can be found by staticfiles. Simply add the app to the *INSTALLED_APPS* setting of your site.

Static file finders are currently considered a private interface, and this interface is thus undocumented.

6.18.7 Core Settings Topical Index

Cache

- *CACHES*
- *CACHE_MIDDLEWARE_ALIAS*
- *CACHE_MIDDLEWARE_KEY_PREFIX*
- *CACHE_MIDDLEWARE_SECONDS*

Database

- *DATABASES*
- *DATABASE_ROUTERS*
- *DEFAULT_INDEX_TABLESPACE*
- *DEFAULT_TABLESPACE*

Debugging

- *DEBUG*
- *DEBUG_PROPAGATE_EXCEPTIONS*

Email

- *ADMINS*
- *DEFAULT_CHARSET*
- *DEFAULT_FROM_EMAIL*
- *EMAIL_BACKEND*
- *EMAIL_FILE_PATH*
- *EMAIL_HOST*

- *EMAIL_HOST_PASSWORD*
- *EMAIL_HOST_USER*
- *EMAIL_PORT*
- *EMAIL_SSL_CERTFILE*
- *EMAIL_SSL_KEYFILE*
- *EMAIL_SUBJECT_PREFIX*
- *EMAIL_TIMEOUT*
- *EMAIL_USE_TLS*
- *MANAGERS*
- *SERVER_EMAIL*

Error reporting

- *DEFAULT_EXCEPTION_REPORTER_FILTER*
- *IGNORABLE_404_URLS*
- *MANAGERS*
- *SILENCED_SYSTEM_CHECKS*

File uploads

- *DEFAULT_FILE_STORAGE*
- *FILE_CHARSET*
- *FILE_UPLOAD_HANDLERS*
- *FILE_UPLOAD_MAX_MEMORY_SIZE*
- *FILE_UPLOAD_PERMISSIONS*
- *FILE_UPLOAD_TEMP_DIR*
- *MEDIA_ROOT*
- *MEDIA_URL*

Globalization (`i18n`/`l10n`)

- *DATE_FORMAT*
- *DATE_INPUT_FORMATS*
- *DATETIME_FORMAT*
- *DATETIME_INPUT_FORMATS*
- *DECIMAL_SEPARATOR*
- *FIRST_DAY_OF_WEEK*
- *FORMAT_MODULE_PATH*
- *LANGUAGE_CODE*

- *LANGUAGE_COOKIE_AGE*
- *LANGUAGE_COOKIE_DOMAIN*
- *LANGUAGE_COOKIE_NAME*
- *LANGUAGE_COOKIE_PATH*
- *LANGUAGES*
- *LOCALE_PATHS*
- *MONTH_DAY_FORMAT*
- *NUMBER_GROUPING*
- *SHORT_DATE_FORMAT*
- *SHORT_DATETIME_FORMAT*
- *THOUSAND_SEPARATOR*
- *TIME_FORMAT*
- *TIME_INPUT_FORMATS*
- *TIME_ZONE*
- *USE_I18N*
- *USE_L10N*
- *USE_THOUSAND_SEPARATOR*
- *USE_TZ*
- *YEAR_MONTH_FORMAT*

HTTP

- *DATA_UPLOAD_MAX_MEMORY_SIZE*
- *DATA_UPLOAD_MAX_NUMBER_FIELDS*
- *DEFAULT_CHARSET*
- *DEFAULT_CONTENT_TYPE*
- *DISALLOWED_USER_AGENTS*
- *FORCE_SCRIPT_NAME*
- *INTERNAL_IPS*
- *MIDDLEWARE*
- *MIDDLEWARE_CLASSES*
- Security
  - *SECURE_BROWSER_XSS_FILTER*
  - *SECURE_CONTENT_TYPE_NOSNIFF*
  - *SECURE_HSTS_INCLUDE_SUBDOMAINS*
  - *SECURE_HSTS_SECONDS*
  - *SECURE_PROXY_SSL_HEADER*

  - *SECURE_REDIRECT_EXEMPT*
  - *SECURE_SSL_HOST*
  - *SECURE_SSL_REDIRECT*
- *SIGNING_BACKEND*
- *USE_ETAGS*
- *USE_X_FORWARDED_HOST*
- *USE_X_FORWARDED_PORT*
- *WSGI_APPLICATION*

Logging

- *LOGGING*
- *LOGGING_CONFIG*

Models

- *ABSOLUTE_URL_OVERRIDES*
- *FIXTURE_DIRS*
- *INSTALLED_APPS*

Security

- Cross Site Request Forgery Protection
  - *CSRF_COOKIE_DOMAIN*
  - *CSRF_COOKIE_NAME*
  - *CSRF_COOKIE_PATH*
  - *CSRF_COOKIE_SECURE*
  - *CSRF_FAILURE_VIEW*
  - *CSRF_HEADER_NAME*
  - *CSRF_TRUSTED_ORIGINS*
- *SECRET_KEY*
- *X_FRAME_OPTIONS*

Serialization

- *DEFAULT_CHARSET*
- *SERIALIZATION_MODULES*

Templates

- *TEMPLATES*

Testing

- Database: *TEST*
- *TEST_NON_SERIALIZED_APPS*
- *TEST_RUNNER*

URLs

- *APPEND_SLASH*
- *PREPEND_WWW*
- *ROOT_URLCONF*

6.19 Signals

A list of all the signals that Django sends. All built-in signals are sent using the *send()* method.

:

See the documentation on the signal dispatcher for information regarding how to register for and receive signals.

The authentication framework sends *signals when a user is logged in / out*.

6.19.1 Model signals

The *django.db.models.signals* module defines a set of signals sent by the model system.

: Many of these signals are sent by various model methods like `__init__()` or *save()* that you can override in your own code.
If you override these methods on your model, you must call the parent class' methods for this signals to be sent.
Note also that Django stores signal handlers as weak references by default, so if your handler is a local function, it may be garbage collected. To prevent this, pass `weak=False` when you call the signal's *connect()*.

: Model signals `sender` model can be lazily referenced when connecting a receiver by specifying its full application label. For example, an `Answer` model defined in the `polls` application could be referenced as `'polls.Answer'`. This sort of reference can be quite handy when dealing with circular import dependencies and swappable models.

pre_init

`django.db.models.signals.`**`pre_init`**

Whenever you instantiate a Django model, this signal is sent at the beginning of the model's `__init__()` method.

Arguments sent with this signal:

sender The model class that just had an instance created.

args A list of positional arguments passed to `__init__()`:

kwargs A dictionary of keyword arguments passed to `__init__()`:

For example, the tutorial has this line:

```
p = Poll(question="What's up?", pub_date=datetime.now())
```

The arguments sent to a *pre_init* handler would be:

Argument	Value
sender	Poll (the class itself)
args	[] (an empty list because there were no positional arguments passed to __init__().)
kwargs	{'question': "What's up?", 'pub_date': datetime.now()}

post_init

django.db.models.signals.**post_init**

Like pre_init, but this one is sent when the __init__() method finishes.

Arguments sent with this signal:

sender As above: the model class that just had an instance created.

instance The actual instance of the model that's just been created.

pre_save

django.db.models.signals.**pre_save**

This is sent at the beginning of a model's *save()* method.

Arguments sent with this signal:

sender The model class.

instance The actual instance being saved.

raw A boolean; True if the model is saved exactly as presented (i.e. when loading a fixture). One should not query/modify other records in the database as the database might not be in a consistent state yet.

using The database alias being used.

update_fields The set of fields to update as passed to *Model.save()*, or None if update_fields wasn't passed to save().

post_save

django.db.models.signals.**post_save**

Like *pre_save*, but sent at the end of the *save()* method.

Arguments sent with this signal:

sender The model class.

instance The actual instance being saved.

created A boolean; True if a new record was created.

raw A boolean; True if the model is saved exactly as presented (i.e. when loading a fixture). One should not query/modify other records in the database as the database might not be in a consistent state yet.

using The database alias being used.

update_fields The set of fields to update as passed to *Model.save()*, or None if update_fields wasn't passed to save().

pre_delete

django.db.models.signals.**pre_delete**

Sent at the beginning of a model's *delete()* method and a queryset's *delete()* method.

Arguments sent with this signal:

sender The model class.

instance The actual instance being deleted.

using The database alias being used.

post_delete

django.db.models.signals.**post_delete**

Like *pre_delete*, but sent at the end of a model's *delete()* method and a queryset's *delete()* method.

Arguments sent with this signal:

sender The model class.

instance The actual instance being deleted.

Note that the object will no longer be in the database, so be very careful what you do with this instance.

using The database alias being used.

m2m_changed

django.db.models.signals.**m2m_changed**

Sent when a *ManyToManyField* is changed on a model instance. Strictly speaking, this is not a model signal since it is sent by the *ManyToManyField*, but since it complements the *pre_save*/*post_save* and *pre_delete*/*post_delete* when it comes to tracking changes to models, it is included here.

Arguments sent with this signal:

sender The intermediate model class describing the *ManyToManyField*. This class is automatically created when a many-to-many field is defined; you can access it using the through attribute on the many-to-many field.

instance The instance whose many-to-many relation is updated. This can be an instance of the sender, or of the class the *ManyToManyField* is related to.

action A string indicating the type of update that is done on the relation. This can be one of the following:

"pre_add" Sent *before* one or more objects are added to the relation.

"post_add" Sent *after* one or more objects are added to the relation.

"pre_remove" Sent *before* one or more objects are removed from the relation.

"post_remove" Sent *after* one or more objects are removed from the relation.

"pre_clear" Sent *before* the relation is cleared.

"post_clear" Sent *after* the relation is cleared.

reverse Indicates which side of the relation is updated (i.e., if it is the forward or reverse relation that is being modified).

model The class of the objects that are added to, removed from or cleared from the relation.

pk_set For the `pre_add`, `post_add`, `pre_remove` and `post_remove` actions, this is a set of primary key values that have been added to or removed from the relation.

For the `pre_clear` and `post_clear` actions, this is `None`.

using The database alias being used.

For example, if a `Pizza` can have multiple `Topping` objects, modeled like this:

```
class Topping(models.Model):
    # ...
    pass

class Pizza(models.Model):
    # ...
    toppings = models.ManyToManyField(Topping)
```

If we connected a handler like this:

```
from django.db.models.signals import m2m_changed

def toppings_changed(sender, **kwargs):
    # Do something
    pass

m2m_changed.connect(toppings_changed, sender=Pizza.toppings.through)
```

and then did something like this:

```
>>> p = Pizza.objects.create(...)
>>> t = Topping.objects.create(...)
>>> p.toppings.add(t)
```

the arguments sent to a *m2m_changed* handler (`toppings_changed` in the example above) would be:

Argument	Value
`sender`	`Pizza.toppings.through` (the intermediate m2m class)
`instance`	`p` (the `Pizza` instance being modified)
`action`	`"pre_add"` (followed by a separate signal with `"post_add"`)
`reverse`	`False` (`Pizza` contains the *ManyToManyField*, so this call modifies the forward relation)
`model`	`Topping` (the class of the objects added to the `Pizza`)
`pk_set`	`{t.id}` (since only `Topping t` was added to the relation)
`using`	`"default"` (since the default router sends writes here)

And if we would then do something like this:

```
>>> t.pizza_set.remove(p)
```

the arguments sent to a *m2m_changed* handler would be:

Argument	Value
sender	Pizza.toppings.through (the intermediate m2m class)
instance	t (the Topping instance being modified)
action	"pre_remove" (followed by a separate signal with "post_remove")
reverse	True (Pizza contains the *ManyToManyField*, so this call modifies the reverse relation)
model	Pizza (the class of the objects removed from the Topping)
pk_set	{p.id} (since only Pizza p was removed from the relation)
using	"default" (since the default router sends writes here)

class_prepared

django.db.models.signals.**class_prepared**

Sent whenever a model class has been "prepared" – that is, once model has been defined and registered with Django's model system. Django uses this signal internally; it's not generally used in third-party applications.

Since this signal is sent during the app registry population process, and *AppConfig.ready()* runs after the app registry is fully populated, receivers cannot be connected in that method. One possibility is to connect them AppConfig.__init__() instead, taking care not to import models or trigger calls to the app registry.

Arguments that are sent with this signal:

sender The model class which was just prepared.

6.19.2 Management signals

Signals sent by django-admin.

pre_migrate

django.db.models.signals.**pre_migrate**

Sent by the *migrate* command before it starts to install an application. It's not emitted for applications that lack a models module.

Arguments sent with this signal:

sender An *AppConfig* instance for the application about to be migrated/synced.

app_config Same as sender.

verbosity Indicates how much information manage.py is printing on screen. See the *--verbosity* flag for details.

Functions which listen for *pre_migrate* should adjust what they output to the screen based on the value of this argument.

interactive If interactive is True, it's safe to prompt the user to input things on the command line. If interactive is False, functions which listen for this signal should not try to prompt for anything.

For example, the *django.contrib.auth* app only prompts to create a superuser when interactive is True.

using The alias of database on which a command will operate.

plan The migration plan that is going to be used for the migration run. While the plan is not public API, this allows for the rare cases when it is necessary to know the plan. A plan is a list of two-tuples with the first item being the instance of a migration class and the second item showing if the migration was rolled back (True) or applied (False).

apps An instance of *Apps* containing the state of the project before the migration run. It should be used instead of the global *apps* registry to retrieve the models you want to perform operations on.

post_migrate

`django.db.models.signals.`**post_migrate**

Sent at the end of the *migrate* (even if no migrations are run) and *flush* commands. It's not emitted for applications that lack a `models` module.

Handlers of this signal must not perform database schema alterations as doing so may cause the *flush* command to fail if it runs during the *migrate* command.

Arguments sent with this signal:

sender An *AppConfig* instance for the application that was just installed.

app_config Same as `sender`.

verbosity Indicates how much information manage.py is printing on screen. See the *--verbosity* flag for details.

Functions which listen for *post_migrate* should adjust what they output to the screen based on the value of this argument.

interactive If `interactive` is `True`, it's safe to prompt the user to input things on the command line. If `interactive` is `False`, functions which listen for this signal should not try to prompt for anything.

For example, the *django.contrib.auth* app only prompts to create a superuser when `interactive` is `True`.

using The database alias used for synchronization. Defaults to the `default` database.

plan The migration plan that was used for the migration run. While the plan is not public API, this allows for the rare cases when it is necessary to know the plan. A plan is a list of two-tuples with the first item being the instance of a migration class and the second item showing if the migration was rolled back (`True`) or applied (`False`).

apps An instance of *Apps* containing the state of the project after the migration run. It should be used instead of the global *apps* registry to retrieve the models you want to perform operations on.

For example, you could register a callback in an *AppConfig* like this:

```
from django.apps import AppConfig
from django.db.models.signals import post_migrate

def my_callback(sender, **kwargs):
    # Your specific logic here
    pass

class MyAppConfig(AppConfig):
    ...

    def ready(self):
        post_migrate.connect(my_callback, sender=self)
```

: If you provide an *AppConfig* instance as the sender argument, please ensure that the signal is registered in *ready()*. `AppConfig`s are recreated for tests that run with a modified set of *INSTALLED_APPS* (such as when settings are overridden) and such signals should be connected for each new `AppConfig` instance.

6.19.3 Request/response signals

Signals sent by the core framework when processing a request.

request_started

django.core.signals.**request_started**

Sent when Django begins processing an HTTP request.

Arguments sent with this signal:

sender The handler class – e.g. `django.core.handlers.wsgi.WsgiHandler` – that handled the request.

environ The `environ` dictionary provided to the request.

request_finished

django.core.signals.**request_finished**

Sent when Django finishes delivering an HTTP response to the client.

: Some WSGI servers and middleware do not always call `close` on the response object after handling a request, most notably uWSGI prior to 1.2.6 and Sentry's error reporting middleware up to 2.0.7. In those cases this signal isn't sent at all. This can result in idle connections to database and memcache servers.

Arguments sent with this signal:

sender The handler class, as above.

got_request_exception

django.core.signals.**got_request_exception**

This signal is sent whenever Django encounters an exception while processing an incoming HTTP request.

Arguments sent with this signal:

sender The handler class, as above.

request The *`HttpRequest`* object.

6.19.4 Test signals

Signals only sent when *running tests*.

setting_changed

django.test.signals.**setting_changed**

This signal is sent when the value of a setting is changed through the `django.test.TestCase.settings()` context manager or the *`django.test.override_settings()`* decorator/context manager.

It's actually sent twice: when the new value is applied ("setup") and when the original value is restored ("teardown"). Use the `enter` argument to distinguish between the two.

You can also import this signal from `django.core.signals` to avoid importing from `django.test` in non-test situations.

Arguments sent with this signal:

sender The settings handler.

setting The name of the setting.

value The value of the setting after the change. For settings that initially don't exist, in the "teardown" phase, `value` is `None`.

enter A boolean; `True` if the setting is applied, `False` if restored.

template_rendered

`django.test.signals.`**template_rendered**

Sent when the test system renders a template. This signal is not emitted during normal operation of a Django server – it is only available during testing.

Arguments sent with this signal:

sender The *`Template`* object which was rendered.

template Same as sender

context The *`Context`* with which the template was rendered.

6.19.5 Database Wrappers

Signals sent by the database wrapper when a database connection is initiated.

connection_created

`django.db.backends.signals.`**connection_created**

Sent when the database wrapper makes the initial connection to the database. This is particularly useful if you'd like to send any post connection commands to the SQL backend.

Arguments sent with this signal:

sender The database wrapper class – i.e. `django.db.backends.postgresql.DatabaseWrapper` or `django.db.backends.mysql.DatabaseWrapper`, etc.

connection The database connection that was opened. This can be used in a multiple-database configuration to differentiate connection signals from different databases.

6.20 Templates

Django's template engine provides a powerful mini-language for defining the user-facing layer of your application, encouraging a clean separation of application and presentation logic. Templates can be maintained by anyone with an understanding of HTML; no knowledge of Python is required. For introductory material, see Templates topic guide.

6.20.1 The Django template language

This document explains the language syntax of the Django template system. If you're looking for a more technical perspective on how it works and how to extend it, see The Django template language: for Python programmers.

Django's template language is designed to strike a balance between power and ease. It's designed to feel comfortable to those used to working with HTML. If you have any exposure to other text-based template languages, such as Smarty or Jinja2, you should feel right at home with Django's templates.

Philosophy

If you have a background in programming, or if you're used to languages which mix programming code directly into HTML, you'll want to bear in mind that the Django template system is not simply Python embedded into HTML. This is by design: the template system is meant to express presentation, not program logic.

The Django template system provides tags which function similarly to some programming constructs – an `if` tag for boolean tests, a `for` tag for looping, etc. – but these are not simply executed as the corresponding Python code, and the template system will not execute arbitrary Python expressions. Only the tags, filters and syntax listed below are supported by default (although you can add your own extensions to the template language as needed).

Templates

A template is simply a text file. It can generate any text-based format (HTML, XML, CSV, etc.).

A template contains **variables**, which get replaced with values when the template is evaluated, and **tags**, which control the logic of the template.

Below is a minimal template that illustrates a few basics. Each element will be explained later in this document.

```
{% extends "base_generic.html" %}

{% block title %}{{ section.title }}{% endblock %}

{% block content %}
<h1>{{ section.title }}</h1>

{% for story in story_list %}
<h2>
  <a href="{{ story.get_absolute_url }}">
    {{ story.headline|upper }}
  </a>
</h2>
<p>{{ story.tease|truncatewords:"100" }}</p>
{% endfor %}
{% endblock %}
```

Philosophy

Why use a text-based template instead of an XML-based one (like Zope's TAL)? We wanted Django's template language to be usable for more than just XML/HTML templates. At World Online, we use it for emails, JavaScript and CSV. You can use the template language for any text-based format.

Oh, and one more thing: making humans edit XML is sadistic!

Variables

Variables look like this: `{{ variable }}`. When the template engine encounters a variable, it evaluates that variable and replaces it with the result. Variable names consist of any combination of alphanumeric characters and the underscore (`"_"`). The dot (`"."`) also appears in variable sections, although that has a special meaning, as indicated below. Importantly, *you cannot have spaces or punctuation characters in variable names.*

Use a dot (`.`) to access attributes of a variable.

Behind the scenes

Technically, when the template system encounters a dot, it tries the following lookups, in this order:

- Dictionary lookup
- Attribute or method lookup
- Numeric index lookup

If the resulting value is callable, it is called with no arguments. The result of the call becomes the template value.

This lookup order can cause some unexpected behavior with objects that override dictionary lookup. For example, consider the following code snippet that attempts to loop over a `collections.defaultdict`:

```
{% for k, v in defaultdict.iteritems %}
    Do something with k and v here...
{% endfor %}
```

Because dictionary lookup happens first, that behavior kicks in and provides a default value instead of using the intended `.iteritems()` method. In this case, consider converting to a dictionary first.

In the above example, `{{ section.title }}` will be replaced with the `title` attribute of the `section` object.

If you use a variable that doesn't exist, the template system will insert the value of the `string_if_invalid` option, which is set to `''` (the empty string) by default.

Note that "bar" in a template expression like `{{ foo.bar }}` will be interpreted as a literal string and not using the value of the variable "bar", if one exists in the template context.

Filters

You can modify variables for display by using **filters**.

Filters look like this: `{{ name|lower }}`. This displays the value of the `{{ name }}` variable after being filtered through the *`lower`* filter, which converts text to lowercase. Use a pipe (`|`) to apply a filter.

Filters can be "chained." The output of one filter is applied to the next. `{{ text|escape|linebreaks }}` is a common idiom for escaping text contents, then converting line breaks to `<p>` tags.

Some filters take arguments. A filter argument looks like this: `{{ bio|truncatewords:30 }}`. This will display the first 30 words of the `bio` variable.

Filter arguments that contain spaces must be quoted; for example, to join a list with commas and spaces you'd use `{{ list|join:", " }}`.

Django provides about sixty built-in template filters. You can read all about them in the *built-in filter reference*. To give you a taste of what's available, here are some of the more commonly used template filters:

`default` If a variable is false or empty, use given default. Otherwise, use the value of the variable. For example:

```
{{ value|default:"nothing" }}
```

If `value` isn't provided or is empty, the above will display "`nothing`".

length Returns the length of the value. This works for both strings and lists. For example:

```
{{ value|length }}
```

If `value` is `['a', 'b', 'c', 'd']`, the output will be `4`.

filesizeformat Formats the value like a "human-readable" file size (i.e. `'13 KB'`, `'4.1 MB'`, `'102 bytes'`, etc.). For example:

```
{{ value|filesizeformat }}
```

If `value` is 123456789, the output would be `117.7 MB`.

Again, these are just a few examples; see the *built-in filter reference* for the complete list.

You can also create your own custom template filters; see Custom template tags and filters.

:

Django's admin interface can include a complete reference of all template tags and filters available for a given site. See The Django admin documentation generator.

Tags

Tags look like this: `{% tag %}`. Tags are more complex than variables: Some create text in the output, some control flow by performing loops or logic, and some load external information into the template to be used by later variables.

Some tags require beginning and ending tags (i.e. `{% tag %} ... tag contents ... {% endtag %}`).

Django ships with about two dozen built-in template tags. You can read all about them in the *built-in tag reference*. To give you a taste of what's available, here are some of the more commonly used tags:

for Loop over each item in an array. For example, to display a list of athletes provided in `athlete_list`:

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

if*, *elif*, and *else Evaluates a variable, and if that variable is "true" the contents of the block are displayed:

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list|length }}
{% elif athlete_in_locker_room_list %}
    Athletes should be out of the locker room soon!
{% else %}
    No athletes.
{% endif %}
```

In the above, if `athlete_list` is not empty, the number of athletes will be displayed by the `{{ athlete_list|length }}` variable. Otherwise, if `athlete_in_locker_room_list` is not empty, the message "Athletes should be out..." will be displayed. If both lists are empty, "No athletes." will be displayed.

You can also use filters and various operators in the *if* tag:

```
{% if athlete_list|length > 1 %}
   Team: {% for athlete in athlete_list %} ... {% endfor %}
{% else %}
   Athlete: {{ athlete_list.0.name }}
{% endif %}
```

While the above example works, be aware that most template filters return strings, so mathematical comparisons using filters will generally not work as you expect. `length` is an exception.

`block` and `extends` Set up *template inheritance* (see below), a powerful way of cutting down on "boilerplate" in templates.

Again, the above is only a selection of the whole list; see the *built-in tag reference* for the complete list.

You can also create your own custom template tags; see Custom template tags and filters.

:

Django's admin interface can include a complete reference of all template tags and filters available for a given site. See The Django admin documentation generator.

Comments

To comment-out part of a line in a template, use the comment syntax: `{# #}`.

For example, this template would render as `'hello'`:

```
{# greeting #}hello
```

A comment can contain any template code, invalid or not. For example:

```
{# {% if foo %}bar{% else %} #}
```

This syntax can only be used for single-line comments (no newlines are permitted between the `{#` and `#}` delimiters). If you need to comment out a multiline portion of the template, see the `comment` tag.

Template inheritance

The most powerful – and thus the most complex – part of Django's template engine is template inheritance. Template inheritance allows you to build a base "skeleton" template that contains all the common elements of your site and defines **blocks** that child templates can override.

It's easiest to understand template inheritance by starting with an example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}My amazing site{% endblock %}</title>
</head>

<body>
    <div id="sidebar">
        {% block sidebar %}
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
        {% endblock %}
```

```
    </div>

    <div id="content">
        {% block content %}{% endblock %}
    </div>
</body>
</html>
```

This template, which we'll call `base.html`, defines a simple HTML skeleton document that you might use for a simple two-column page. It's the job of "child" templates to fill the empty blocks with content.

In this example, the *`block`* tag defines three blocks that child templates can fill in. All the *`block`* tag does is to tell the template engine that a child template may override those portions of the template.

A child template might look like this:

```
{% extends "base.html" %}

{% block title %}My amazing blog{% endblock %}

{% block content %}
{% for entry in blog_entries %}
    <h2>{{ entry.title }}</h2>
    <p>{{ entry.body }}</p>
{% endfor %}
{% endblock %}
```

The *`extends`* tag is the key here. It tells the template engine that this template "extends" another template. When the template system evaluates this template, first it locates the parent – in this case, "base.html".

At that point, the template engine will notice the three *`block`* tags in `base.html` and replace those blocks with the contents of the child template. Depending on the value of `blog_entries`, the output might look like:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>My amazing blog</title>
</head>

<body>
    <div id="sidebar">
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
    </div>

    <div id="content">
        <h2>Entry one</h2>
        <p>This is my first entry.</p>

        <h2>Entry two</h2>
        <p>This is my second entry.</p>
    </div>
</body>
</html>
```

Note that since the child template didn't define the `sidebar` block, the value from the parent template is used instead. Content within a `{% block %}` tag in a parent template is always used as a fallback.

You can use as many levels of inheritance as needed. One common way of using inheritance is the following three-level approach:

- Create a `base.html` template that holds the main look-and-feel of your site.
- Create a `base_SECTIONNAME.html` template for each "section" of your site. For example, `base_news.html`, `base_sports.html`. These templates all extend `base.html` and include section-specific styles/design.
- Create individual templates for each type of page, such as a news article or blog entry. These templates extend the appropriate section template.

This approach maximizes code reuse and makes it easy to add items to shared content areas, such as section-wide navigation.

Here are some tips for working with inheritance:

- If you use `{% extends %}` in a template, it must be the first template tag in that template. Template inheritance won't work, otherwise.
- More `{% block %}` tags in your base templates are better. Remember, child templates don't have to define all parent blocks, so you can fill in reasonable defaults in a number of blocks, then only define the ones you need later. It's better to have more hooks than fewer hooks.
- If you find yourself duplicating content in a number of templates, it probably means you should move that content to a `{% block %}` in a parent template.
- If you need to get the content of the block from the parent template, the `{{ block.super }}` variable will do the trick. This is useful if you want to add to the contents of a parent block instead of completely overriding it. Data inserted using `{{ block.super }}` will not be automatically escaped (see the *next section*), since it was already escaped, if necessary, in the parent template.
- For extra readability, you can optionally give a *name* to your `{% endblock %}` tag. For example:

```
{% block content %}
...
{% endblock content %}
```

  In larger templates, this technique helps you see which `{% block %}` tags are being closed.

Finally, note that you can't define multiple `block` tags with the same name in the same template. This limitation exists because a block tag works in "both" directions. That is, a block tag doesn't just provide a hole to fill – it also defines the content that fills the hole in the *parent*. If there were two similarly-named `block` tags in a template, that template's parent wouldn't know which one of the blocks' content to use.

Automatic HTML escaping

When generating HTML from templates, there's always a risk that a variable will include characters that affect the resulting HTML. For example, consider this template fragment:

```
Hello, {{ name }}
```

At first, this seems like a harmless way to display a user's name, but consider what would happen if the user entered their name as this:

```
<script>alert('hello')</script>
```

With this name value, the template would be rendered as:

```
Hello, <script>alert('hello')</script>
```

...which means the browser would pop-up a JavaScript alert box!

Similarly, what if the name contained a `'<'` symbol, like this?

```
<b>username
```

That would result in a rendered template like this:

```
Hello, <b>username
```

...which, in turn, would result in the remainder of the Web page being bolded!

Clearly, user-submitted data shouldn't be trusted blindly and inserted directly into your Web pages, because a malicious user could use this kind of hole to do potentially bad things. This type of security exploit is called a Cross Site Scripting (XSS) attack.

To avoid this problem, you have two options:

- One, you can make sure to run each untrusted variable through the *escape* filter (documented below), which converts potentially harmful HTML characters to unharmful ones. This was the default solution in Django for its first few years, but the problem is that it puts the onus on *you*, the developer / template author, to ensure you're escaping everything. It's easy to forget to escape data.
- Two, you can take advantage of Django's automatic HTML escaping. The remainder of this section describes how auto-escaping works.

By default in Django, every template automatically escapes the output of every variable tag. Specifically, these five characters are escaped:

- < is converted to `<`
- > is converted to `>`
- `'` (single quote) is converted to `'`
- `"` (double quote) is converted to `"`
- `&` is converted to `&`

Again, we stress that this behavior is on by default. If you're using Django's template system, you're protected.

How to turn it off

If you don't want data to be auto-escaped, on a per-site, per-template level or per-variable level, you can turn it off in several ways.

Why would you want to turn it off? Because sometimes, template variables contain data that you *intend* to be rendered as raw HTML, in which case you don't want their contents to be escaped. For example, you might store a blob of HTML in your database and want to embed that directly into your template. Or, you might be using Django's template system to produce text that is *not* HTML – like an email message, for instance.

For individual variables To disable auto-escaping for an individual variable, use the *safe* filter:

```
This will be escaped: {{ data }}
This will not be escaped: {{ data|safe }}
```

Think of *safe* as shorthand for *safe from further escaping* or *can be safely interpreted as HTML*. In this example, if `data` contains `'<b>'`, the output will be:

```
This will be escaped: &lt;b&gt;
This will not be escaped: <b>
```

For template blocks To control auto-escaping for a template, wrap the template (or just a particular section of the template) in the *autoescape* tag, like so:

```
{% autoescape off %}
    Hello {{ name }}
{% endautoescape %}
```

The *autoescape* tag takes either `on` or `off` as its argument. At times, you might want to force auto-escaping when it would otherwise be disabled. Here is an example template:

```
Auto-escaping is on by default. Hello {{ name }}

{% autoescape off %}
    This will not be auto-escaped: {{ data }}.

    Nor this: {{ other_data }}
    {% autoescape on %}
        Auto-escaping applies again: {{ name }}
    {% endautoescape %}
{% endautoescape %}
```

The auto-escaping tag passes its effect onto templates that extend the current one as well as templates included via the *include* tag, just like all block tags. For example:

`base.html`

```
{% autoescape off %}
<h1>{% block title %}{% endblock %}</h1>
{% block content %}
{% endblock %}
{% endautoescape %}
```

`child.html`

```
{% extends "base.html" %}
{% block title %}This &amp; that{% endblock %}
{% block content %}{{ greeting }}{% endblock %}
```

Because auto-escaping is turned off in the base template, it will also be turned off in the child template, resulting in the following rendered HTML when the `greeting` variable contains the string `<b>Hello!</b>`:

```
<h1>This &amp; that</h1>
<b>Hello!</b>
```

Notes

Generally, template authors don't need to worry about auto-escaping very much. Developers on the Python side (people writing views and custom filters) need to think about the cases in which data shouldn't be escaped, and mark data appropriately, so things Just Work in the template.

If you're creating a template that might be used in situations where you're not sure whether auto-escaping is enabled, then add an *escape* filter to any variable that needs escaping. When auto-escaping is on, there's no danger of the *escape* filter *double-escaping* data – the *escape* filter does not affect auto-escaped variables.

String literals and automatic escaping

As we mentioned earlier, filter arguments can be strings:

```
{{ data|default:"This is a string literal." }}
```

All string literals are inserted **without** any automatic escaping into the template – they act as if they were all passed through the *safe* filter. The reasoning behind this is that the template author is in control of what goes into the string literal, so they can make sure the text is correctly escaped when the template is written.

This means you would write

```
{{ data|default:"3 &lt; 2" }}
```

...rather than:

```
{{ data|default:"3 < 2" }}  {# Bad! Don't do this. #}
```

This doesn't affect what happens to data coming from the variable itself. The variable's contents are still automatically escaped, if necessary, because they're beyond the control of the template author.

Accessing method calls

Most method calls attached to objects are also available from within templates. This means that templates have access to much more than just class attributes (like field names) and variables passed in from views. For example, the Django ORM provides the *"entry_set"* syntax for finding a collection of objects related on a foreign key. Therefore, given a model called "comment" with a foreign key relationship to a model called "task" you can loop through all comments attached to a given task like this:

```
{% for comment in task.comment_set.all %}
    {{ comment }}
{% endfor %}
```

Similarly, QuerySets provide a `count()` method to count the number of objects they contain. Therefore, you can obtain a count of all comments related to the current task with:

```
{{ task.comment_set.all.count }}
```

And of course you can easily access methods you've explicitly defined on your own models:

models.py

```
class Task(models.Model):
    def foo(self):
        return "bar"
```

template.html

```
{{ task.foo }}
```

Because Django intentionally limits the amount of logic processing available in the template language, it is not possible to pass arguments to method calls accessed from within templates. Data should be calculated in views, then passed to templates for display.

Custom tag and filter libraries

Certain applications provide custom tag and filter libraries. To access them in a template, ensure the application is in *INSTALLED_APPS* (we'd add `'django.contrib.humanize'` for this example), and then use the *load* tag in a template:

```
{% load humanize %}

{{ 45000|intcomma }}
```

In the above, the *load* tag loads the humanize tag library, which then makes the intcomma filter available for use. If you've enabled *django.contrib.admindocs*, you can consult the documentation area in your admin to find the list of custom libraries in your installation.

The *load* tag can take multiple library names, separated by spaces. Example:

```
{% load humanize i18n %}
```

See Custom template tags and filters for information on writing your own custom template libraries.

Custom libraries and template inheritance

When you load a custom tag or filter library, the tags/filters are only made available to the current template – not any parent or child templates along the template-inheritance path.

For example, if a template foo.html has {% load humanize %}, a child template (e.g., one that has {% extends "foo.html" %}) will *not* have access to the humanize template tags and filters. The child template is responsible for its own {% load humanize %}.

This is a feature for the sake of maintainability and sanity.

:

The Templates Reference Covers built-in tags, built-in filters, using an alternative template, language, and more.

6.20.2 Built-in template tags and filters

This document describes Django's built-in template tags and filters. It is recommended that you use the automatic documentation, if available, as this will also include documentation for any custom tags or filters installed.

Built-in tag reference

autoescape

Controls the current auto-escaping behavior. This tag takes either on or off as an argument and that determines whether auto-escaping is in effect inside the block. The block is closed with an endautoescape ending tag.

When auto-escaping is in effect, all variable content has HTML escaping applied to it before placing the result into the output (but after any filters have been applied). This is equivalent to manually applying the *escape* filter to each variable.

The only exceptions are variables that are already marked as "safe" from escaping, either by the code that populated the variable, or because it has had the *safe* or *escape* filters applied.

Sample usage:

```
{% autoescape on %}
    {{ body }}
{% endautoescape %}
```

block

Defines a block that can be overridden by child templates. See *Template inheritance* for more information.

comment

Ignores everything between `{% comment %}` and `{% endcomment %}`. An optional note may be inserted in the first tag. For example, this is useful when commenting out code for documenting why the code was disabled.

Sample usage:

```
<p>Rendered text with {{ pub_date|date:"c" }}</p>
{% comment "Optional note" %}
    <p>Commented out text with {{ create_date|date:"c" }}</p>
{% endcomment %}
```

`comment` tags cannot be nested.

csrf_token

This tag is used for CSRF protection, as described in the documentation for Cross Site Request Forgeries.

cycle

Produces one of its arguments each time this tag is encountered. The first argument is produced on the first encounter, the second argument on the second encounter, and so forth. Once all arguments are exhausted, the tag cycles to the first argument and produces it again.

This tag is particularly useful in a loop:

```
{% for o in some_list %}
    <tr class="{% cycle 'row1' 'row2' %}">
        ...
    </tr>
{% endfor %}
```

The first iteration produces HTML that refers to class `row1`, the second to `row2`, the third to `row1` again, and so on for each iteration of the loop.

You can use variables, too. For example, if you have two template variables, `rowvalue1` and `rowvalue2`, you can alternate between their values like this:

```
{% for o in some_list %}
    <tr class="{% cycle rowvalue1 rowvalue2 %}">
        ...
    </tr>
{% endfor %}
```

Variables included in the cycle will be escaped. You can disable auto-escaping with:

```
{% for o in some_list %}
    <tr class="{% autoescape off %}{% cycle rowvalue1 rowvalue2 %}{% endautoescape %}">
        ...
    </tr>
{% endfor %}
```

You can mix variables and strings:

```
{% for o in some_list %}
    <tr class="{% cycle 'row1' rowvalue2 'row3' %}">
        ...
    </tr>
{% endfor %}
```

In some cases you might want to refer to the current value of a cycle without advancing to the next value. To do this, just give the `{% cycle %}` tag a name, using "as", like this:

```
{% cycle 'row1' 'row2' as rowcolors %}
```

From then on, you can insert the current value of the cycle wherever you'd like in your template by referencing the cycle name as a context variable. If you want to move the cycle to the next value independently of the original `cycle` tag, you can use another `cycle` tag and specify the name of the variable. So, the following template:

```
<tr>
    <td class="{% cycle 'row1' 'row2' as rowcolors %}">...</td>
    <td class="{{ rowcolors }}">...</td>
</tr>
<tr>
    <td class="{% cycle rowcolors %}">...</td>
    <td class="{{ rowcolors }}">...</td>
</tr>
```

would output:

```
<tr>
    <td class="row1">...</td>
    <td class="row1">...</td>
</tr>
<tr>
    <td class="row2">...</td>
    <td class="row2">...</td>
</tr>
```

You can use any number of values in a `cycle` tag, separated by spaces. Values enclosed in single quotes (`'`) or double quotes (`"`) are treated as string literals, while values without quotes are treated as template variables.

By default, when you use the `as` keyword with the cycle tag, the usage of `{% cycle %}` that initiates the cycle will itself produce the first value in the cycle. This could be a problem if you want to use the value in a nested loop or an included template. If you only want to declare the cycle but not produce the first value, you can add a `silent` keyword as the last keyword in the tag. For example:

```
{% for obj in some_list %}
    {% cycle 'row1' 'row2' as rowcolors silent %}
    <tr class="{{ rowcolors }}">{% include "subtemplate.html" %}</tr>
{% endfor %}
```

This will output a list of `<tr>` elements with `class` alternating between `row1` and `row2`. The subtemplate will have access to `rowcolors` in its context and the value will match the class of the `<tr>` that encloses it. If the `silent` keyword were to be omitted, `row1` and `row2` would be emitted as normal text, outside the `<tr>` element.

When the silent keyword is used on a cycle definition, the silence automatically applies to all subsequent uses of that specific cycle tag. The following template would output *nothing*, even though the second call to `{% cycle %}` doesn't specify `silent`:

```
{% cycle 'row1' 'row2' as rowcolors silent %}
{% cycle rowcolors %}
```

debug

Outputs a whole load of debugging information, including the current context and imported modules.

extends

Signals that this template extends a parent template.

This tag can be used in two ways:

- `{% extends "base.html" %}` (with quotes) uses the literal value `"base.html"` as the name of the parent template to extend.
- `{% extends variable %}` uses the value of `variable`. If the variable evaluates to a string, Django will use that string as the name of the parent template. If the variable evaluates to a `Template` object, Django will use that object as the parent template.

See *Template inheritance* for more information.

A string argument may be a relative path starting with `./` or `../`. For example, assume the following directory structure:

```
dir1/
    template.html
    base2.html
    my/
        base3.html
base1.html
```

In `template.html`, the following paths would be valid:

```
{% extends "./base2.html" %}
{% extends "../base1.html" %}
{% extends "./my/base3.html" %}
```

The ability to use relative paths was added.

filter

Filters the contents of the block through one or more filters. Multiple filters can be specified with pipes and filters can have arguments, just as in variable syntax.

Note that the block includes *all* the text between the `filter` and `endfilter` tags.

Sample usage:

```
{% filter force_escape|lower %}
    This text will be HTML-escaped, and will appear in all lowercase.
{% endfilter %}
```

: The *escape* and *safe* filters are not acceptable arguments. Instead, use the *autoescape* tag to manage autoescaping for blocks of template code.

---

firstof

Outputs the first argument variable that is not `False`. Outputs nothing if all the passed variables are `False`.

Sample usage:

```
{% firstof var1 var2 var3 %}
```

This is equivalent to:

```
{% if var1 %}
    {{ var1 }}
{% elif var2 %}
    {{ var2 }}
{% elif var3 %}
    {{ var3 }}
{% endif %}
```

You can also use a literal string as a fallback value in case all passed variables are False:

```
{% firstof var1 var2 var3 "fallback value" %}
```

This tag auto-escapes variable values. You can disable auto-escaping with:

```
{% autoescape off %}
    {% firstof var1 var2 var3 "<strong>fallback value</strong>" %}
{% endautoescape %}
```

Or if only some variables should be escaped, you can use:

```
{% firstof var1 var2|safe var3 "<strong>fallback value</strong>"|safe %}
```

You can use the syntax `{% firstof var1 var2 var3 as value %}` to store the output inside a variable.

The "as" syntax was added.

for

Loops over each item in an array, making the item available in a context variable. For example, to display a list of athletes provided in `athlete_list`:

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

You can loop over a list in reverse by using `{% for obj in list reversed %}`.

If you need to loop over a list of lists, you can unpack the values in each sublist into individual variables. For example, if your context contains a list of (x,y) coordinates called `points`, you could use the following to output the list of points:

```
{% for x, y in points %}
    There is a point at {{ x }},{{ y }}
{% endfor %}
```

This can also be useful if you need to access the items in a dictionary. For example, if your context contained a dictionary `data`, the following would display the keys and values of the dictionary:

```
{% for key, value in data.items %}
    {{ key }}: {{ value }}
{% endfor %}
```

Keep in mind that for the dot operator, dictionary key lookup takes precedence over method lookup. Therefore if the `data` dictionary contains a key named `'items'`, `data.items` will return `data['items']` instead of `data.items()`. Avoid adding keys that are named like dictionary methods if you want to use those methods in a template (`items`, `values`, `keys`, etc.). Read more about the lookup order of the dot operator in the *documentation of template variables*.

The for loop sets a number of variables available within the loop:

Variable	Description
`forloop.counter`	The current iteration of the loop (1-indexed)
`forloop.counter0`	The current iteration of the loop (0-indexed)
`forloop.revcounter`	The number of iterations from the end of the loop (1-indexed)
`forloop.revcounter0`	The number of iterations from the end of the loop (0-indexed)
`forloop.first`	True if this is the first time through the loop
`forloop.last`	True if this is the last time through the loop
`forloop.parentloop`	For nested loops, this is the loop surrounding the current one

for ... empty

The `for` tag can take an optional `{% empty %}` clause whose text is displayed if the given array is empty or could not be found:

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% empty %}
    <li>Sorry, no athletes in this list.</li>
{% endfor %}
</ul>
```

The above is equivalent to – but shorter, cleaner, and possibly faster than – the following:

```
<ul>
  {% if athlete_list %}
    {% for athlete in athlete_list %}
      <li>{{ athlete.name }}</li>
    {% endfor %}
  {% else %}
    <li>Sorry, no athletes in this list.</li>
  {% endif %}
</ul>
```

if

The `{% if %}` tag evaluates a variable, and if that variable is "true" (i.e. exists, is not empty, and is not a false boolean value) the contents of the block are output:

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list|length }}
{% elif athlete_in_locker_room_list %}
    Athletes should be out of the locker room soon!
```

```
{% else %}
    No athletes.
{% endif %}
```

In the above, if `athlete_list` is not empty, the number of athletes will be displayed by the `{{ athlete_list|length }}` variable.

As you can see, the `if` tag may take one or several `{% elif %}` clauses, as well as an `{% else %}` clause that will be displayed if all previous conditions fail. These clauses are optional.

Boolean operators *if* tags may use `and`, `or` or `not` to test a number of variables or to negate a given variable:

```
{% if athlete_list and coach_list %}
    Both athletes and coaches are available.
{% endif %}

{% if not athlete_list %}
    There are no athletes.
{% endif %}

{% if athlete_list or coach_list %}
    There are some athletes or some coaches.
{% endif %}

{% if not athlete_list or coach_list %}
    There are no athletes or there are some coaches.
{% endif %}

{% if athlete_list and not coach_list %}
    There are some athletes and absolutely no coaches.
{% endif %}
```

Use of both `and` and `or` clauses within the same tag is allowed, with `and` having higher precedence than `or` e.g.:

```
{% if athlete_list and coach_list or cheerleader_list %}
```

will be interpreted like:

```
if (athlete_list and coach_list) or cheerleader_list
```

Use of actual parentheses in the *if* tag is invalid syntax. If you need them to indicate precedence, you should use nested *if* tags.

if tags may also use the operators ==, !=, <, >, <=, >=, `in`, `not in`, `is`, and `is not` which work as follows:

== operator Equality. Example:

```
{% if somevar == "x" %}
  This appears if variable somevar equals the string "x"
{% endif %}
```

!= operator Inequality. Example:

```
{% if somevar != "x" %}
  This appears if variable somevar does not equal the string "x",
  or if somevar is not found in the context
{% endif %}
```

< operator Less than. Example:

```
{% if somevar < 100 %}
  This appears if variable somevar is less than 100.
{% endif %}
```

> operator Greater than. Example:

```
{% if somevar > 0 %}
  This appears if variable somevar is greater than 0.
{% endif %}
```

<= operator Less than or equal to. Example:

```
{% if somevar <= 100 %}
  This appears if variable somevar is less than 100 or equal to 100.
{% endif %}
```

>= operator Greater than or equal to. Example:

```
{% if somevar >= 1 %}
  This appears if variable somevar is greater than 1 or equal to 1.
{% endif %}
```

in operator Contained within. This operator is supported by many Python containers to test whether the given value is in the container. The following are some examples of how `x in y` will be interpreted:

```
{% if "bc" in "abcdef" %}
  This appears since "bc" is a substring of "abcdef"
{% endif %}

{% if "hello" in greetings %}
  If greetings is a list or set, one element of which is the string
  "hello", this will appear.
{% endif %}

{% if user in users %}
  If users is a QuerySet, this will appear if user is an
  instance that belongs to the QuerySet.
{% endif %}
```

not in operator Not contained within. This is the negation of the `in` operator.

is operator Object identity. Tests if two values are the same object. Example:

```
{% if somevar is True %}
  This appears if and only if somevar is True.
{% endif %}

{% if somevar is None %}
  This appears if somevar is None, or if somevar is not found in the context.
{% endif %}
```

`is not` operator Negated object identity. Tests if two values are not the same object. This is the negation of the `is` operator. Example:

```
{% if somevar is not True %}
  This appears if somevar is not True, or if somevar is not found in the
  context.
{% endif %}

{% if somevar is not None %}
  This appears if and only if somevar is not None.
{% endif %}
```

Filters You can also use filters in the *if* expression. For example:

```
{% if messages|length >= 100 %}
   You have lots of messages today!
{% endif %}
```

Complex expressions All of the above can be combined to form complex expressions. For such expressions, it can be important to know how the operators are grouped when the expression is evaluated - that is, the precedence rules. The precedence of the operators, from lowest to highest, is as follows:

- `or`
- `and`
- `not`
- `in`
- `==`, `!=`, `<`, `>`, `<=`, `>=`

(This follows Python exactly). So, for example, the following complex *if* tag:

```
{% if a == b or c == d and e %}
```

...will be interpreted as:

```
(a == b) or ((c == d) and e)
```

If you need different precedence, you will need to use nested *if* tags. Sometimes that is better for clarity anyway, for the sake of those who do not know the precedence rules.

The comparison operators cannot be 'chained' like in Python or in mathematical notation. For example, instead of using:

```
{% if a > b > c %}  (WRONG)
```

you should use:

```
{% if a > b and b > c %}
```

`ifequal` and `ifnotequal`

`{% ifequal a b %} ... {% endifequal %}` is an obsolete way to write `{% if a == b %} ... {% endif %}`. Likewise, `{% ifnotequal a b %} ... {% endifnotequal %}` is superseded by `{% if a != b %} ... {% endif %}`. The `ifequal` and `ifnotequal` tags will be deprecated in a future release.

ifchanged

Check if a value has changed from the last iteration of a loop.

The `{% ifchanged %}` block tag is used within a loop. It has two possible uses.

1. Checks its own rendered contents against its previous state and only displays the content if it has changed. For example, this displays a list of days, only displaying the month if it changes:

```
<h1>Archive for {{ year }}</h1>

{% for date in days %}
    {% ifchanged %}<h3>{{ date|date:"F" }}</h3>{% endifchanged %}
    <a href="{{ date|date:"M/d"|lower }}/">{{ date|date:"j" }}</a>
{% endfor %}
```

2. If given one or more variables, check whether any variable has changed. For example, the following shows the date every time it changes, while showing the hour if either the hour or the date has changed:

```
{% for date in days %}
    {% ifchanged date.date %} {{ date.date }} {% endifchanged %}
    {% ifchanged date.hour date.date %}
        {{ date.hour }}
    {% endifchanged %}
{% endfor %}
```

The `ifchanged` tag can also take an optional `{% else %}` clause that will be displayed if the value has not changed:

```
{% for match in matches %}
    <div style="background-color:
        {% ifchanged match.ballot_id %}
            {% cycle "red" "blue" %}
        {% else %}
            gray
        {% endifchanged %}
    ">{{ match }}</div>
{% endfor %}
```

include

Loads a template and renders it with the current context. This is a way of "including" other templates within a template.

The template name can either be a variable or a hard-coded (quoted) string, in either single or double quotes.

This example includes the contents of the template `"foo/bar.html"`:

```
{% include "foo/bar.html" %}
```

A string argument may be a relative path starting with `./` or `../` as described in the *`extends`* tag.

The ability to use a relative path was added.

This example includes the contents of the template whose name is contained in the variable `template_name`:

```
{% include template_name %}
```

The variable may also be any object with a `render()` method that accepts a context. This allows you to reference a compiled `Template` in your context.

An included template is rendered within the context of the template that includes it. This example produces the output "Hello, John!":

- Context: variable person is set to "John" and variable greeting is set to "Hello".
- Template:

```
{% include "name_snippet.html" %}
```

- The name_snippet.html template:

```
{{ greeting }}, {{ person|default:"friend" }}!
```

You can pass additional context to the template using keyword arguments:

```
{% include "name_snippet.html" with person="Jane" greeting="Hello" %}
```

If you want to render the context only with the variables provided (or even no variables at all), use the only option. No other variables are available to the included template:

```
{% include "name_snippet.html" with greeting="Hi" only %}
```

If the included template causes an exception while it's rendered (including if it's missing or has syntax errors), the behavior varies depending on the *template engine's* debug option (if not set, this option defaults to the value of *DEBUG*). When debug mode is turned on, an exception like *TemplateDoesNotExist* or *TemplateSyntaxError* will be raised. When debug mode is turned off, {% include %} logs a warning to the django.template logger with the exception that happens while rendering the included template and returns an empty string.

Template logging now includes the warning logging mentioned above.

: The *include* tag should be considered as an implementation of "render this subtemplate and include the HTML", not as "parse this subtemplate and include its contents as if it were part of the parent". This means that there is no shared state between included templates – each include is a completely independent rendering process.

Blocks are evaluated *before* they are included. This means that a template that includes blocks from another will contain blocks that have *already been evaluated and rendered* - not blocks that can be overridden by, for example, an extending template.

---

load

Loads a custom template tag set.

For example, the following template would load all the tags and filters registered in somelibrary and otherlibrary located in package package:

```
{% load somelibrary package.otherlibrary %}
```

You can also selectively load individual filters or tags from a library, using the from argument. In this example, the template tags/filters named foo and bar will be loaded from somelibrary:

```
{% load foo bar from somelibrary %}
```

See Custom tag and filter libraries for more information.

lorem

Displays random "lorem ipsum" Latin text. This is useful for providing sample data in templates.

Usage:

```
{% lorem [count] [method] [random] %}
```

The `{% lorem %}` tag can be used with zero, one, two or three arguments. The arguments are:

Argument	Description
`count`	A number (or variable) containing the number of paragraphs or words to generate (default is 1).
`method`	Either `w` for words, `p` for HTML paragraphs or `b` for plain-text paragraph blocks (default is `b`).
`random`	The word `random`, which if given, does not use the common paragraph ("Lorem ipsum dolor sit amet...") when generating text.

Examples:

- `{% lorem %}` will output the common "lorem ipsum" paragraph.
- `{% lorem 3 p %}` will output the common "lorem ipsum" paragraph and two random paragraphs each wrapped in HTML <p> tags.
- `{% lorem 2 w random %}` will output two random Latin words.

now

Displays the current date and/or time, using a format according to the given string. Such string can contain format specifiers characters as described in the *date* filter section.

Example:

```
It is {% now "jS F Y H:i" %}
```

Note that you can backslash-escape a format string if you want to use the "raw" value. In this example, both "o" and "f" are backslash-escaped, because otherwise each is a format string that displays the year and the time, respectively:

```
It is the {% now "jS \o\f F" %}
```

This would display as "It is the 4th of September".

: The format passed can also be one of the predefined ones *DATE_FORMAT*, *DATETIME_FORMAT*, *SHORT_DATE_FORMAT* or *SHORT_DATETIME_FORMAT*. The predefined formats may vary depending on the current locale and if Format localization is enabled, e.g.:

```
It is {% now "SHORT_DATETIME_FORMAT" %}
```

You can also use the syntax `{% now "Y" as current_year %}` to store the output (as a string) inside a variable. This is useful if you want to use `{% now %}` inside a template tag like *blocktrans* for example:

```
{% now "Y" as current_year %}
{% blocktrans %}Copyright {{ current_year }}{% endblocktrans %}
```

regroup

Regroups a list of alike objects by a common attribute.

This complex tag is best illustrated by way of an example: say that `cities` is a list of cities represented by dictionaries containing `"name"`, `"population"`, and `"country"` keys:

```
cities = [
    {'name': 'Mumbai', 'population': '19,000,000', 'country': 'India'},
    {'name': 'Calcutta', 'population': '15,000,000', 'country': 'India'},
    {'name': 'New York', 'population': '20,000,000', 'country': 'USA'},
    {'name': 'Chicago', 'population': '7,000,000', 'country': 'USA'},
    {'name': 'Tokyo', 'population': '33,000,000', 'country': 'Japan'},
]
```

...and you'd like to display a hierarchical list that is ordered by country, like this:

- India
  - Mumbai: 19,000,000
  - Calcutta: 15,000,000
- USA
  - New York: 20,000,000
  - Chicago: 7,000,000
- Japan
  - Tokyo: 33,000,000

You can use the `{% regroup %}` tag to group the list of cities by country. The following snippet of template code would accomplish this:

```
{% regroup cities by country as country_list %}

<ul>
{% for country in country_list %}
    <li>{{ country.grouper }}
    <ul>
        {% for city in country.list %}
          <li>{{ city.name }}: {{ city.population }}</li>
        {% endfor %}
    </ul>
    </li>
{% endfor %}
</ul>
```

Let's walk through this example. `{% regroup %}` takes three arguments: the list you want to regroup, the attribute to group by, and the name of the resulting list. Here, we're regrouping the `cities` list by the `country` attribute and calling the result `country_list`.

`{% regroup %}` produces a list (in this case, `country_list`) of **group objects**. Each group object has two attributes:

- `grouper` – the item that was grouped by (e.g., the string "India" or "Japan").
- `list` – a list of all items in this group (e.g., a list of all cities with country='India').

Note that `{% regroup %}` does not order its input! Our example relies on the fact that the `cities` list was ordered by `country` in the first place. If the `cities` list did *not* order its members by `country`, the regrouping would

naively display more than one group for a single country. For example, say the `cities` list was set to this (note that the countries are not grouped together):

```
cities = [
    {'name': 'Mumbai', 'population': '19,000,000', 'country': 'India'},
    {'name': 'New York', 'population': '20,000,000', 'country': 'USA'},
    {'name': 'Calcutta', 'population': '15,000,000', 'country': 'India'},
    {'name': 'Chicago', 'population': '7,000,000', 'country': 'USA'},
    {'name': 'Tokyo', 'population': '33,000,000', 'country': 'Japan'},
]
```

With this input for `cities`, the example `{% regroup %}` template code above would result in the following output:

- India
  - Mumbai: 19,000,000
- USA
  - New York: 20,000,000
- India
  - Calcutta: 15,000,000
- USA
  - Chicago: 7,000,000
- Japan
  - Tokyo: 33,000,000

The easiest solution to this gotcha is to make sure in your view code that the data is ordered according to how you want to display it.

Another solution is to sort the data in the template using the *dictsort* filter, if your data is in a list of dictionaries:

```
{% regroup cities|dictsort:"country" by country as country_list %}
```

Grouping on other properties Any valid template lookup is a legal grouping attribute for the regroup tag, including methods, attributes, dictionary keys and list items. For example, if the "country" field is a foreign key to a class with an attribute "description," you could use:

```
{% regroup cities by country.description as country_list %}
```

Or, if `country` is a field with `choices`, it will have a *get_FOO_display()* method available as an attribute, allowing you to group on the display string rather than the `choices` key:

```
{% regroup cities by get_country_display as country_list %}
```

`{{ country.grouper }}` will now display the value fields from the `choices` set rather than the keys.

spaceless

Removes whitespace between HTML tags. This includes tab characters and newlines.

Example usage:

```
{% spaceless %}
    <p>
        <a href="foo/">Foo</a>
    </p>
{% endspaceless %}
```

This example would return this HTML:

```
<p><a href="foo/">Foo</a></p>
```

Only space between *tags* is removed – not space between tags and text. In this example, the space around `Hello` won't be stripped:

```
{% spaceless %}
    <strong>
        Hello
    </strong>
{% endspaceless %}
```

templatetag

Outputs one of the syntax characters used to compose template tags.

Since the template system has no concept of "escaping", to display one of the bits used in template tags, you must use the `{% templatetag %}` tag.

The argument tells which template bit to output:

Argument	Outputs
`openblock`	`{%`
`closeblock`	`%}`
`openvariable`	`{{`
`closevariable`	`}}`
`openbrace`	`{`
`closebrace`	`}`
`opencomment`	`{#`
`closecomment`	`#}`

Sample usage:

```
{% templatetag openblock %} url 'entry_list' {% templatetag closeblock %}
```

url

Returns an absolute path reference (a URL without the domain name) matching a given view and optional parameters. Any special characters in the resulting path will be encoded using *iri_to_uri()*.

This is a way to output links without violating the DRY principle by having to hard-code URLs in your templates:

```
{% url 'some-url-name' v1 v2 %}
```

The first argument is a *url()* `name`. It can be a quoted literal or any other context variable. Additional arguments are optional and should be space-separated values that will be used as arguments in the URL. The example above shows passing positional arguments. Alternatively you may use keyword syntax:

```
{% url 'some-url-name' arg1=v1 arg2=v2 %}
```

Do not mix both positional and keyword syntax in a single call. All arguments required by the URLconf should be present.

For example, suppose you have a view, `app_views.client`, whose URLconf takes a client ID (here, `client()` is a method inside the views file `app_views.py`). The URLconf line might look like this:

```
('^client/([0-9]+)/$', app_views.client, name='app-views-client')
```

If this app's URLconf is included into the project's URLconf under a path such as this:

```
('^clients/', include('project_name.app_name.urls'))
```

...then, in a template, you can create a link to this view like this:

```
{% url 'app-views-client' client.id %}
```

The template tag will output the string `/clients/client/123/`.

Note that if the URL you're reversing doesn't exist, you'll get an *NoReverseMatch* exception raised, which will cause your site to display an error page.

If you'd like to retrieve a URL without displaying it, you can use a slightly different call:

```
{% url 'some-url-name' arg arg2 as the_url %}

<a href="{{ the_url }}">I'm linking to {{ the_url }}</a>
```

The scope of the variable created by the `as var` syntax is the `{% block %}` in which the `{% url %}` tag appears.

This `{% url ... as var %}` syntax will *not* cause an error if the view is missing. In practice you'll use this to link to views that are optional:

```
{% url 'some-url-name' as the_url %}
{% if the_url %}
  <a href="{{ the_url }}">Link to optional stuff</a>
{% endif %}
```

If you'd like to retrieve a namespaced URL, specify the fully qualified name:

```
{% url 'myapp:view-name' %}
```

This will follow the normal *namespaced URL resolution strategy*, including using any hints provided by the context as to the current application.

: Don't forget to put quotes around the *url()* name, otherwise the value will be interpreted as a context variable!

verbatim

Stops the template engine from rendering the contents of this block tag.

A common use is to allow a JavaScript template layer that collides with Django's syntax. For example:

```
{% verbatim %}
    {{if dying}}Still alive.{{/if}}
{% endverbatim %}
```

You can also designate a specific closing tag, allowing the use of `{% endverbatim %}` as part of the unrendered contents:

```
{% verbatim myblock %}
    Avoid template rendering via the {% verbatim %}{% endverbatim %} block.
{% endverbatim myblock %}
```

widthratio

For creating bar charts and such, this tag calculates the ratio of a given value to a maximum value, and then applies that ratio to a constant.

For example:

```
<img src="bar.png" alt="Bar"
     height="10" width="{% widthratio this_value max_value max_width %}" />
```

If `this_value` is 175, `max_value` is 200, and `max_width` is 100, the image in the above example will be 88 pixels wide (because 175/200 = .875; .875 * 100 = 87.5 which is rounded up to 88).

In some cases you might want to capture the result of `widthratio` in a variable. It can be useful, for instance, in a *`blocktrans`* like this:

```
{% widthratio this_value max_value max_width as width %}
{% blocktrans %}The width is: {{ width }}{% endblocktrans %}
```

with

Caches a complex variable under a simpler name. This is useful when accessing an "expensive" method (e.g., one that hits the database) multiple times.

For example:

```
{% with total=business.employees.count %}
    {{ total }} employee{{ total|pluralize }}
{% endwith %}
```

The populated variable (in the example above, `total`) is only available between the `{% with %}` and `{% endwith %}` tags.

You can assign more than one context variable:

```
{% with alpha=1 beta=2 %}
    ...
{% endwith %}
```

: The previous more verbose format is still supported: `{% with business.employees.count as total %}`

Built-in filter reference

add

Adds the argument to the value.

For example:

```
{{ value|add:"2" }}
```

If `value` is `4`, then the output will be `6`.

This filter will first try to coerce both values to integers. If this fails, it'll attempt to add the values together anyway. This will work on some data types (strings, list, etc.) and fail on others. If it fails, the result will be an empty string.

For example, if we have:

```
{{ first|add:second }}
```

and `first` is `[1, 2, 3]` and `second` is `[4, 5, 6]`, then the output will be `[1, 2, 3, 4, 5, 6]`.

: Strings that can be coerced to integers will be **summed**, not concatenated, as in the first example above.

addslashes

Adds slashes before quotes. Useful for escaping strings in CSV, for example.

For example:

```
{{ value|addslashes }}
```

If `value` is `"I'm using Django"`, the output will be `"I\'m using Django"`.

capfirst

Capitalizes the first character of the value. If the first character is not a letter, this filter has no effect.

For example:

```
{{ value|capfirst }}
```

If `value` is `"django"`, the output will be `"Django"`.

center

Centers the value in a field of a given width.

For example:

```
"{{ value|center:"15" }}"
```

If `value` is `"Django"`, the output will be `" Django "`.

cut

Removes all values of arg from the given string.

For example:

```
{{ value|cut:" " }}
```

If `value` is `"String with spaces"`, the output will be `"Stringwithspaces"`.

date

Formats a date according to the given format.

Uses a similar format as PHP's `date()` function (https://php.net/date) with some differences.

: These format characters are not used in Django outside of templates. They were designed to be compatible with PHP to ease transitioning for designers.

Available format strings:

Format character	Description
a	`'a.m.'` or `'p.m.'` (Note that this is slightly different than PHP's output, because this includes periods to ma
A	`'AM'` or `'PM'`.
b	Month, textual, 3 letters, lowercase.
B	Not implemented.
c	ISO 8601 format. (Note: unlike others formatters, such as "Z", "O" or "r", the "c" formatter will not add timez
d	Day of the month, 2 digits with leading zeros.
D	Day of the week, textual, 3 letters.
e	Timezone name. Could be in any format, or might return an empty string, depending on the datetime.
E	Month, locale specific alternative representation usually used for long date representation.
f	Time, in 12-hour hours and minutes, with minutes left off if they're zero. Proprietary extension.
F	Month, textual, long.
g	Hour, 12-hour format without leading zeros.
G	Hour, 24-hour format without leading zeros.
h	Hour, 12-hour format.
H	Hour, 24-hour format.
i	Minutes.
I	Daylight Savings Time, whether it's in effect or not.
j	Day of the month without leading zeros.
l	Day of the week, textual, long.
L	Boolean for whether it's a leap year.
m	Month, 2 digits with leading zeros.
M	Month, textual, 3 letters.
n	Month without leading zeros.
N	Month abbreviation in Associated Press style. Proprietary extension.
o	ISO-8601 week-numbering year, corresponding to the ISO-8601 week number (W) which uses leap weeks. See
O	Difference to Greenwich time in hours.
P	Time, in 12-hour hours, minutes and 'a.m.'/'p.m.', with minutes left off if they're zero and the special-case strin
r	**RFC 5322** formatted date.
s	Seconds, 2 digits with leading zeros.
S	English ordinal suffix for day of the month, 2 characters.
t	Number of days in the given month.
T	Time zone of this machine.
u	Microseconds.
U	Seconds since the Unix Epoch (January 1 1970 00:00:00 UTC).
w	Day of the week, digits without leading zeros.
W	ISO-8601 week number of year, with weeks starting on Monday.
y	Year, 2 digits.
Y	Year, 4 digits.
z	Day of the year.

Format character	Description
Z	Time zone offset in seconds. The offset for timezones west of UTC is always negative, and for those east of UT

For example:

```
{{ value|date:"D d M Y" }}
```

If `value` is a `datetime` object (e.g., the result of `datetime.datetime.now()`), the output will be the string `'Wed 09 Jan 2008'`.

The format passed can be one of the predefined ones *DATE_FORMAT*, *DATETIME_FORMAT*, *SHORT_DATE_FORMAT* or *SHORT_DATETIME_FORMAT*, or a custom format that uses the format specifiers shown in the table above. Note that predefined formats may vary depending on the current locale.

Assuming that *USE_L10N* is `True` and *LANGUAGE_CODE* is, for example, `"es"`, then for:

```
{{ value|date:"SHORT_DATE_FORMAT" }}
```

the output would be the string `"09/01/2008"` (the `"SHORT_DATE_FORMAT"` format specifier for the `es` locale as shipped with Django is `"d/m/Y"`).

When used without a format string:

```
{{ value|date }}
```

...the formatting string defined in the *DATE_FORMAT* setting will be used, without applying any localization.

You can combine `date` with the *time* filter to render a full representation of a `datetime` value. E.g.:

```
{{ value|date:"D d M Y" }} {{ value|time:"H:i" }}
```

default

If value evaluates to `False`, uses the given default. Otherwise, uses the value.

For example:

```
{{ value|default:"nothing" }}
```

If `value` is `""` (the empty string), the output will be `nothing`.

default_if_none

If (and only if) value is `None`, uses the given default. Otherwise, uses the value.

Note that if an empty string is given, the default value will *not* be used. Use the *default* filter if you want to fallback for empty strings.

For example:

```
{{ value|default_if_none:"nothing" }}
```

If `value` is `None`, the output will be the string `"nothing"`.

dictsort

Takes a list of dictionaries and returns that list sorted by the key given in the argument.

For example:

```
{{ value|dictsort:"name" }}
```

If `value` is:

```
[
    {'name': 'zed', 'age': 19},
    {'name': 'amy', 'age': 22},
    {'name': 'joe', 'age': 31},
]
```

then the output would be:

```
[
    {'name': 'amy', 'age': 22},
    {'name': 'joe', 'age': 31},
    {'name': 'zed', 'age': 19},
]
```

You can also do more complicated things like:

```
{% for book in books|dictsort:"author.age" %}
    * {{ book.title }} ({{ book.author.name }})
{% endfor %}
```

If `books` is:

```
[
    {'title': '1984', 'author': {'name': 'George', 'age': 45}},
    {'title': 'Timequake', 'author': {'name': 'Kurt', 'age': 75}},
    {'title': 'Alice', 'author': {'name': 'Lewis', 'age': 33}},
]
```

then the output would be:

```
* Alice (Lewis)
* 1984 (George)
* Timequake (Kurt)
```

`dictsort` can also order a list of lists (or any other object implementing `__getitem__()`) by elements at specified index. For example:

```
{{ value|dictsort:0 }}
```

If `value` is:

```
[
    ('a', '42'),
    ('c', 'string'),
    ('b', 'foo'),
]
```

then the output would be:

```
[
    ('a', '42'),
    ('b', 'foo'),
    ('c', 'string'),
]
```

You must pass the index as an integer rather than a string. The following produce empty output:

```
{{ values|dictsort:"0" }}
```

The ability to order a list of lists was added.

dictsortreversed

Takes a list of dictionaries and returns that list sorted in reverse order by the key given in the argument. This works exactly the same as the above filter, but the returned value will be in reverse order.

divisibleby

Returns `True` if the value is divisible by the argument.

For example:

```
{{ value|divisibleby:"3" }}
```

If `value` is `21`, the output would be `True`.

escape

Escapes a string's HTML. Specifically, it makes these replacements:

- < is converted to `<`
- > is converted to `>`
- ' (single quote) is converted to `'`
- " (double quote) is converted to `"`
- & is converted to `&`

The escaping is only applied when the string is output, so it does not matter where in a chained sequence of filters you put `escape`: it will always be applied as though it were the last filter. If you want escaping to be applied immediately, use the *force_escape* filter.

Applying `escape` to a variable that would normally have auto-escaping applied to the result will only result in one round of escaping being done. So it is safe to use this function even in auto-escaping environments. If you want multiple escaping passes to be applied, use the *force_escape* filter.

For example, you can apply `escape` to fields when *autoescape* is off:

```
{% autoescape off %}
    {{ title|escape }}
{% endautoescape %}
```

1.10 : The "lazy" behavior of the `escape` filter is deprecated. It will change to immediately apply *conditional_escape()* in Django 2.0.

escapejs

Escapes characters for use in JavaScript strings. This does *not* make the string safe for use in HTML, but does protect you from syntax errors when using templates to generate JavaScript/JSON.

For example:

```
{{ value|escapejs }}
```

If `value` is `"testing\r\njavascript \'string" <b>escaping</b>"`, the output will be `"testing\\u000D\\u000Ajavascript \\u0027string\\u0022 \\u003Cb\\u003Eescaping\\u003C/b\\u003E"`.

filesizeformat

Formats the value like a 'human-readable' file size (i.e. `'13 KB'`, `'4.1 MB'`, `'102 bytes'`, etc.).

For example:

```
{{ value|filesizeformat }}
```

If `value` is 123456789, the output would be `117.7 MB`.

File sizes and SI units

Strictly speaking, `filesizeformat` does not conform to the International System of Units which recommends using KiB, MiB, GiB, etc. when byte sizes are calculated in powers of 1024 (which is the case here). Instead, Django uses traditional unit names (KB, MB, GB, etc.) corresponding to names that are more commonly used.

---

first

Returns the first item in a list.

For example:

```
{{ value|first }}
```

If `value` is the list `['a', 'b', 'c']`, the output will be `'a'`.

floatformat

When used without an argument, rounds a floating-point number to one decimal place – but only if there's a decimal part to be displayed. For example:

value	Template	Output
34.23234	{{ value|floatformat }}	34.2
34.00000	{{ value|floatformat }}	34
34.26000	{{ value|floatformat }}	34.3

If used with a numeric integer argument, `floatformat` rounds a number to that many decimal places. For example:

value	Template	Output
34.23234	{{ value|floatformat:3 }}	34.232
34.00000	{{ value|floatformat:3 }}	34.000
34.26000	{{ value|floatformat:3 }}	34.260

Particularly useful is passing 0 (zero) as the argument which will round the float to the nearest integer.

value	Template	Output
34.23234	{{ value|floatformat:"0" }}	34
34.00000	{{ value|floatformat:"0" }}	34
39.56000	{{ value|floatformat:"0" }}	40

If the argument passed to `floatformat` is negative, it will round a number to that many decimal places – but only if there's a decimal part to be displayed. For example:

value	Template	Output
34.23234	{{ value|floatformat:"-3" }}	34.232
34.00000	{{ value|floatformat:"-3" }}	34
34.26000	{{ value|floatformat:"-3" }}	34.260

Using `floatformat` with no argument is equivalent to using `floatformat` with an argument of `-1`.

force_escape

Applies HTML escaping to a string (see the *escape* filter for details). This filter is applied *immediately* and returns a new, escaped string. This is useful in the rare cases where you need multiple escaping or want to apply other filters to the escaped results. Normally, you want to use the *escape* filter.

For example, if you want to catch the <p> HTML elements created by the *linebreaks* filter:

```
{% autoescape off %}
    {{ body|linebreaks|force_escape }}
{% endautoescape %}
```

get_digit

Given a whole number, returns the requested digit, where 1 is the right-most digit, 2 is the second-right-most digit, etc. Returns the original value for invalid input (if input or argument is not an integer, or if argument is less than 1). Otherwise, output is always an integer.

For example:

```
{{ value|get_digit:"2" }}
```

If `value` is `123456789`, the output will be `8`.

iriencode

Converts an IRI (Internationalized Resource Identifier) to a string that is suitable for including in a URL. This is necessary if you're trying to use strings containing non-ASCII characters in a URL.

It's safe to use this filter on a string that has already gone through the *urlencode* filter.

For example:

```
{{ value|iriencode }}
```

If `value` is `"?test=1&me=2"`, the output will be `"?test=1&me=2"`.

join

Joins a list with a string, like Python's `str.join(list)`

For example:

```
{{ value|join:" // " }}
```

If `value` is the list `['a', 'b', 'c']`, the output will be the string `"a // b // c"`.

last

Returns the last item in a list.

For example:

```
{{ value|last }}
```

If `value` is the list `['a', 'b', 'c', 'd']`, the output will be the string `"d"`.

length

Returns the length of the value. This works for both strings and lists.

For example:

```
{{ value|length }}
```

If `value` is `['a', 'b', 'c', 'd']` or `"abcd"`, the output will be `4`.

The filter returns `0` for an undefined variable.

length_is

Returns `True` if the value's length is the argument, or `False` otherwise.

For example:

```
{{ value|length_is:"4" }}
```

If `value` is `['a', 'b', 'c', 'd']` or `"abcd"`, the output will be `True`.

linebreaks

Replaces line breaks in plain text with appropriate HTML; a single newline becomes an HTML line break (`<br />`) and a new line followed by a blank line becomes a paragraph break (`</p>`).

For example:

```
{{ value|linebreaks }}
```

If `value` is `Joel\nis a slug`, the output will be `<p>Joel<br />is a slug</p>`.

linebreaksbr

Converts all newlines in a piece of plain text to HTML line breaks (`<br />`).

For example:

```
{{ value|linebreaksbr }}
```

If `value` is `Joel\nis a slug`, the output will be `Joel<br />is a slug`.

linenumbers

Displays text with line numbers.

For example:

```
{{ value|linenumbers }}
```

If `value` is:

```
one
two
three
```

the output will be:

```
1. one
2. two
3. three
```

ljust

Left-aligns the value in a field of a given width.

Argument: field size

For example:

```
"{{ value|ljust:"10" }}"
```

If `value` is `Django`, the output will be `"Django    "`.

lower

Converts a string into all lowercase.

For example:

```
{{ value|lower }}
```

If `value` is `Totally LOVING this Album!`, the output will be `totally loving this album!`.

make_list

Returns the value turned into a list. For a string, it's a list of characters. For an integer, the argument is cast into an unicode string before creating a list.

For example:

```
{{ value|make_list }}
```

If `value` is the string `"Joel"`, the output would be the list `['J', 'o', 'e', 'l']`. If `value` is `123`, the output will be the list `['1', '2', '3']`.

phone2numeric

Converts a phone number (possibly containing letters) to its numerical equivalent.

The input doesn't have to be a valid phone number. This will happily convert any string.

For example:

```
{{ value|phone2numeric }}
```

If `value` is `800-COLLECT`, the output will be `800-2655328`.

pluralize

Returns a plural suffix if the value is not 1. By default, this suffix is `'s'`.

Example:

```
You have {{ num_messages }} message{{ num_messages|pluralize }}.
```

If `num_messages` is `1`, the output will be `You have 1 message.` If `num_messages` is `2` the output will be `You have 2 messages.`

For words that require a suffix other than `'s'`, you can provide an alternate suffix as a parameter to the filter.

Example:

```
You have {{ num_walruses }} walrus{{ num_walruses|pluralize:"es" }}.
```

For words that don't pluralize by simple suffix, you can specify both a singular and plural suffix, separated by a comma.

Example:

```
You have {{ num_cherries }} cherr{{ num_cherries|pluralize:"y,ies" }}.
```

: Use *blocktrans* to pluralize translated strings.

pprint

A wrapper around `pprint.pprint()` – for debugging, really.

random

Returns a random item from the given list.

For example:

```
{{ value|random }}
```

If `value` is the list `['a', 'b', 'c', 'd']`, the output could be `"b"`.

rjust

Right-aligns the value in a field of a given width.

Argument: field size

For example:

```
"{{ value|rjust:"10" }}"
```

If `value` is `Django`, the output will be `"  Django"`.

safe

Marks a string as not requiring further HTML escaping prior to output. When autoescaping is off, this filter has no effect.

: If you are chaining filters, a filter applied after `safe` can make the contents unsafe again. For example, the following code prints the variable as is, unescaped:

```
{{ var|safe|escape }}
```

---

safeseq

Applies the *`safe`* filter to each element of a sequence. Useful in conjunction with other filters that operate on sequences, such as *`join`*. For example:

```
{{ some_list|safeseq|join:", " }}
```

You couldn't use the *`safe`* filter directly in this case, as it would first convert the variable into a string, rather than working with the individual elements of the sequence.

slice

Returns a slice of the list.

Uses the same syntax as Python's list slicing. See http://www.diveintopython3.net/native-datatypes.html#slicinglists for an introduction.

Example:

```
{{ some_list|slice:":2" }}
```

If some_list is ['a', 'b', 'c'], the output will be ['a', 'b'].

slugify

Converts to ASCII. Converts spaces to hyphens. Removes characters that aren't alphanumerics, underscores, or hyphens. Converts to lowercase. Also strips leading and trailing whitespace.

For example:

```
{{ value|slugify }}
```

If value is "Joel is a slug", the output will be "joel-is-a-slug".

stringformat

Formats the variable according to the argument, a string formatting specifier. This specifier uses the printf-style String Formatting syntax, with the exception that the leading "%" is dropped.

For example:

```
{{ value|stringformat:"E" }}
```

If value is 10, the output will be 1.000000E+01.

striptags

Makes all possible efforts to strip all [X]HTML tags.

For example:

```
{{ value|striptags }}
```

If value is "Joel <button>is</button> a slug", the output will be "Joel is a slug".

No safety guarantee

Note that striptags doesn't give any guarantee about its output being HTML safe, particularly with non valid HTML input. So **NEVER** apply the safe filter to a striptags output. If you are looking for something more robust, you can use the bleach Python library, notably its clean method.

---

time

Formats a time according to the given format.

Given format can be the predefined one *TIME_FORMAT*, or a custom format, same as the *date* filter. Note that the predefined format is locale-dependent.

For example:

```
{{ value|time:"H:i" }}
```

If `value` is equivalent to `datetime.datetime.now()`, the output will be the string `"01:23"`.

Another example:

Assuming that *USE_L10N* is `True` and *LANGUAGE_CODE* is, for example, `"de"`, then for:

```
{{ value|time:"TIME_FORMAT" }}
```

the output will be the string `"01:23:00"` (The `"TIME_FORMAT"` format specifier for the `de` locale as shipped with Django is `"H:i:s"`).

The `time` filter will only accept parameters in the format string that relate to the time of day, not the date (for obvious reasons). If you need to format a `date` value, use the *date* filter instead (or along `time` if you need to render a full `datetime` value).

There is one exception the above rule: When passed a `datetime` value with attached timezone information (a *time-zone-aware* `datetime` instance) the `time` filter will accept the timezone-related *format specifiers* `'e'`, `'O'` , `'T'` and `'Z'`.

When used without a format string:

```
{{ value|time }}
```

...the formatting string defined in the *TIME_FORMAT* setting will be used, without applying any localization.

timesince

Formats a date as the time since that date (e.g., “4 days, 6 hours”).

Takes an optional argument that is a variable containing the date to use as the comparison point (without the argument, the comparison point is *now*). For example, if `blog_date` is a date instance representing midnight on 1 June 2006, and `comment_date` is a date instance for 08:00 on 1 June 2006, then the following would return “8 hours”:

```
{{ blog_date|timesince:comment_date }}
```

Comparing offset-naive and offset-aware datetimes will return an empty string.

Minutes is the smallest unit used, and “0 minutes” will be returned for any date that is in the future relative to the comparison point.

timeuntil

Similar to `timesince`, except that it measures the time from now until the given date or datetime. For example, if today is 1 June 2006 and `conference_date` is a date instance holding 29 June 2006, then `{{ conference_date|timeuntil }}` will return “4 weeks”.

Takes an optional argument that is a variable containing the date to use as the comparison point (instead of *now*). If `from_date` contains 22 June 2006, then the following will return “1 week”:

```
{{ conference_date|timeuntil:from_date }}
```

Comparing offset-naive and offset-aware datetimes will return an empty string.

Minutes is the smallest unit used, and “0 minutes” will be returned for any date that is in the past relative to the comparison point.

title

Converts a string into titlecase by making words start with an uppercase character and the remaining characters lowercase. This tag makes no effort to keep "trivial words" in lowercase.

For example:

```
{{ value|title }}
```

If `value` is `"my FIRST post"`, the output will be `"My First Post"`.

truncatechars

Truncates a string if it is longer than the specified number of characters. Truncated strings will end with a translatable ellipsis sequence ("...").

Argument: Number of characters to truncate to

For example:

```
{{ value|truncatechars:9 }}
```

If `value` is `"Joel is a slug"`, the output will be `"Joel i..."`.

truncatechars_html

Similar to *truncatechars*, except that it is aware of HTML tags. Any tags that are opened in the string and not closed before the truncation point are closed immediately after the truncation.

For example:

```
{{ value|truncatechars_html:9 }}
```

If `value` is `"<p>Joel is a slug</p>"`, the output will be `"<p>Joel i...</p>"`.

Newlines in the HTML content will be preserved.

truncatewords

Truncates a string after a certain number of words.

Argument: Number of words to truncate after

For example:

```
{{ value|truncatewords:2 }}
```

If `value` is `"Joel is a slug"`, the output will be `"Joel is ..."`.

Newlines within the string will be removed.

truncatewords_html

Similar to *truncatewords*, except that it is aware of HTML tags. Any tags that are opened in the string and not closed before the truncation point, are closed immediately after the truncation.

This is less efficient than *truncatewords*, so should only be used when it is being passed HTML text.

For example:

```
{{ value|truncatewords_html:2 }}
```

If value is " Joel is a slug ", the output will be " Joel is ... ".

Newlines in the HTML content will be preserved.

unordered_list

Recursively takes a self-nested list and returns an HTML unordered list – WITHOUT opening and closing <ul> tags.

The list is assumed to be in the proper format. For example, if var contains ['States', ['Kansas', ['Lawrence', 'Topeka'], 'Illinois']], then {{ var|unordered_list }} would return:

```
<li>States
<ul>
        <li>Kansas
        <ul>
                <li>Lawrence</li>
                <li>Topeka</li>
        </ul>
        </li>
        <li>Illinois</li>
</ul>
</li>
```

upper

Converts a string into all uppercase.

For example:

```
{{ value|upper }}
```

If value is "Joel is a slug", the output will be "JOEL IS A SLUG".

urlencode

Escapes a value for use in a URL.

For example:

```
{{ value|urlencode }}
```

If value is "https://www.example.org/foo?a=b&c=d", the output will be "https%3A//www.example.org/foo%3Fa%3Db%26c%3Dd".

An optional argument containing the characters which should not be escaped can be provided.

If not provided, the '/' character is assumed safe. An empty string can be provided when *all* characters should be escaped. For example:

```
{{ value|urlencode:"" }}
```

If `value` is `"https://www.example.org/"`, the output will be `"https%3A%2F%2Fwww.example.org%2F"`.

urlize

Converts URLs and email addresses in text into clickable links.

This template tag works on links prefixed with `http://`, `https://`, or `www.`. For example, `https://goo.gl/aia1t` will get converted but `goo.gl/aia1t` won't.

It also supports domain-only links ending in one of the original top level domains (`.com`, `.edu`, `.gov`, `.int`, `.mil`, `.net`, and `.org`). For example, `djangoproject.com` gets converted.

Links can have trailing punctuation (periods, commas, close-parens) and leading punctuation (opening parens), and `urlize` will still do the right thing.

Links generated by `urlize` have a `rel="nofollow"` attribute added to them.

For example:

```
{{ value|urlize }}
```

If `value` is `"Check out www.djangoproject.com"`, the output will be `"Check out <a href="http://www.djangoproject.com" rel="nofollow">www.djangoproject.com</a>"`.

In addition to web links, `urlize` also converts email addresses into `mailto:` links. If `value` is `"Send questions to foo@example.com"`, the output will be `"Send questions to <a href="mailto:foo@example.com">foo@example.com</a>"`.

The `urlize` filter also takes an optional parameter `autoescape`. If `autoescape` is `True`, the link text and URLs will be escaped using Django's built-in *escape* filter. The default value for `autoescape` is `True`.

: If `urlize` is applied to text that already contains HTML markup, things won't work as expected. Apply this filter only to plain text.

---

urlizetrunc

Converts URLs and email addresses into clickable links just like *urlize*, but truncates URLs longer than the given character limit.

Argument: Number of characters that link text should be truncated to, including the ellipsis that's added if truncation is necessary.

For example:

```
{{ value|urlizetrunc:15 }}
```

If `value` is `"Check out www.djangoproject.com"`, the output would be `'Check out <a href="http://www.djangoproject.com" rel="nofollow">www.djangopr...</a>'`.

As with *urlize*, this filter should only be applied to plain text.

wordcount

Returns the number of words.

For example:

```
{{ value|wordcount }}
```

If `value` is `"Joel is a slug"`, the output will be `4`.

wordwrap

Wraps words at specified line length.

Argument: number of characters at which to wrap the text

For example:

```
{{ value|wordwrap:5 }}
```

If `value` is `Joel is a slug`, the output would be:

```
Joel
is a
slug
```

yesno

Maps values for `True`, `False`, and (optionally) `None`, to the strings "yes", "no", "maybe", or a custom mapping passed as a comma-separated list, and returns one of those strings according to the value:

For example:

```
{{ value|yesno:"yeah,no,maybe" }}
```

Value	Argument	Outputs
`True`		`yes`
`True`	`"yeah,no,maybe"`	`yeah`
`False`	`"yeah,no,maybe"`	`no`
`None`	`"yeah,no,maybe"`	`maybe`
`None`	`"yeah,no"`	`no` (converts `None` to `False` if no mapping for `None` is given)

Internationalization tags and filters

Django provides template tags and filters to control each aspect of internationalization in templates. They allow for granular control of translations, formatting, and time zone conversions.

i18n

This library allows specifying translatable text in templates. To enable it, set *`USE_I18N`* to `True`, then load it with `{% load i18n %}`.

See *Internationalization: in template code*.

l10n

This library provides control over the localization of values in templates. You only need to load the library using `{% load l10n %}`, but you'll often set *USE_L10N* to `True` so that localization is active by default.

See *Controlling localization in templates*.

tz

This library provides control over time zone conversions in templates. Like `l10n`, you only need to load the library using `{% load tz %}`, but you'll usually also set *USE_TZ* to `True` so that conversion to local time happens by default.

See *Time zone aware output in templates*.

Other tags and filters libraries

Django comes with a couple of other template-tag libraries that you have to enable explicitly in your *INSTALLED_APPS* setting and enable in your template with the *{% load %}* tag.

django.contrib.humanize

A set of Django template filters useful for adding a "human touch" to data. See django.contrib.humanize.

static

static To link to static files that are saved in *STATIC_ROOT* Django ships with a *static* template tag. If the *django.contrib.staticfiles* app is installed, the tag will serve files using `url()` method of the storage specified by *STATICFILES_STORAGE*. For example:

```
{% load static %}
<img src="{% static "images/hi.jpg" %}" alt="Hi!" />
```

It is also able to consume standard context variables, e.g. assuming a `user_stylesheet` variable is passed to the template:

```
{% load static %}
<link rel="stylesheet" href="{% static user_stylesheet %}" type="text/css" media="screen" />
```

If you'd like to retrieve a static URL without displaying it, you can use a slightly different call:

```
{% load static %}
{% static "images/hi.jpg" as myphoto %}
<img src="{{ myphoto }}"></img>
```

Using Jinja2 templates?

See *Jinja2* for information on using the `static` tag with Jinja2.

In older versions, you had to use `{% load static from staticfiles %}` in your template to serve files from the storage defined in *STATICFILES_STORAGE*. This is no longer required.

get_static_prefix You should prefer the *static* template tag, but if you need more control over exactly where and how *STATIC_URL* is injected into the template, you can use the *get_static_prefix* template tag:

```
{% load static %}
<img src="{% get_static_prefix %}images/hi.jpg" alt="Hi!" />
```

There's also a second form you can use to avoid extra processing if you need the value multiple times:

```
{% load static %}
{% get_static_prefix as STATIC_PREFIX %}

<img src="{{ STATIC_PREFIX }}images/hi.jpg" alt="Hi!" />
<img src="{{ STATIC_PREFIX }}images/hi2.jpg" alt="Hello!" />
```

get_media_prefix Similar to the *get_static_prefix*, `get_media_prefix` populates a template variable with the media prefix *MEDIA_URL*, e.g.:

```
{% load static %}
<body data-media-url="{% get_media_prefix %}">
```

By storing the value in a data attribute, we ensure it's escaped appropriately if we want to use it in a JavaScript context.

6.20.3 The Django template language: for Python programmers

This document explains the Django template system from a technical perspective – how it works and how to extend it. If you're just looking for reference on the language syntax, see The Django template language.

It assumes an understanding of templates, contexts, variables, tags, and rendering. Start with the *introduction to the Django template language* if you aren't familiar with these concepts.

Overview

Using the template system in Python is a three-step process:

1. You configure an *Engine*.
2. You compile template code into a *Template*.
3. You render the template with a *Context*.

Django projects generally rely on the *high level, backend agnostic APIs* for each of these steps instead of the template system's lower level APIs:

1. For each *DjangoTemplates* backend in the *TEMPLATES* setting, Django instantiates an *Engine*. *DjangoTemplates* wraps *Engine* and adapts it to the common template backend API.
2. The *django.template.loader* module provides functions such as *get_template()* for loading templates. They return a `django.template.backends.django.Template` which wraps the actual *django.template.Template*.
3. The `Template` obtained in the previous step has a *render()* method which marshals a context and possibly a request into a *Context* and delegates the rendering to the underlying *Template*.

Configuring an engine

If you are simply using the *DjangoTemplates* backend, this probably isn't the documentation you're looking for. An instance of the `Engine` class described below is accessible using the `engine` attribute of that backend and any attribute defaults mentioned below are overridden by what's passed by *DjangoTemplates*.

class **Engine**(*dirs=None*, *app_dirs=False*, *context_processors=None*, *debug=False*, *loaders=None*, *string_if_invalid=''*, *file_charset='utf-8'*, *libraries=None*, *builtins=None*, *autoescape=True*)

When instantiating an `Engine` all arguments must be passed as keyword arguments:

- `dirs` is a list of directories where the engine should look for template source files. It is used to configure *filesystem.Loader*.

  It defaults to an empty list.

- `app_dirs` only affects the default value of `loaders`. See below.

  It defaults to `False`.

- `autoescape` controls whether HTML autoescaping is enabled.

  It defaults to `True`.

  > **:** Only set it to `False` if you're rendering non-HTML templates!

  The `autoescape` option was added.

- `context_processors` is a list of dotted Python paths to callables that are used to populate the context when a template is rendered with a request. These callables take a request object as their argument and return a `dict` of items to be merged into the context.

  It defaults to an empty list.

  See *RequestContext* for more information.

- `debug` is a boolean that turns on/off template debug mode. If it is `True`, the template engine will store additional debug information which can be used to display a detailed report for any exception raised during template rendering.

  It defaults to `False`.

- `loaders` is a list of template loader classes, specified as strings. Each `Loader` class knows how to import templates from a particular source. Optionally, a tuple can be used instead of a string. The first item in the tuple should be the `Loader` class name, subsequent items are passed to the `Loader` during initialization.

  It defaults to a list containing:

  - `'django.template.loaders.filesystem.Loader'`
  - `'django.template.loaders.app_directories.Loader'` if and only if `app_dirs` is `True`.

  See *Loader types* for details.

- `string_if_invalid` is the output, as a string, that the template system should use for invalid (e.g. misspelled) variables.

  It defaults to the empty string.

  See *How invalid variables are handled* for details.

- `file_charset` is the charset used to read template files on disk.

  It defaults to `'utf-8'`.

- `'libraries'`: A dictionary of labels and dotted Python paths of template tag modules to register with the template engine. This is used to add new libraries or provide alternate labels for existing ones. For example:

```
Engine(
    libraries={
        'myapp_tags': 'path.to.myapp.tags',
        'admin.urls': 'django.contrib.admin.templatetags.admin_urls',
    },
)
```

Libraries can be loaded by passing the corresponding dictionary key to the *{% load %}* tag.

- `'builtins'`: A list of dotted Python paths of template tag modules to add to built-ins. For example:

```
Engine(
    builtins=['myapp.builtins'],
)
```

Tags and filters from built-in libraries can be used without first calling the *{% load %}* tag.

The `libraries` and `builtins` arguments were added.

static `Engine.`**`get_default`**`()`
When a Django project configures one and only one *DjangoTemplates* engine, this method returns the underlying *Engine*. In other circumstances it will raise *ImproperlyConfigured*.

It's required for preserving APIs that rely on a globally available, implicitly configured engine. Any other use is strongly discouraged.

`Engine.`**`from_string`**`(`*template_code*`)`
Compiles the given template code and returns a *Template* object.

`Engine.`**`get_template`**`(`*template_name*`)`
Loads a template with the given name, compiles it and returns a *Template* object.

`Engine.`**`select_template`**`(`*self*, *template_name_list*`)`
Like *get_template()*, except it takes a list of names and returns the first template that was found.

Loading a template

The recommended way to create a *Template* is by calling the factory methods of the *Engine*: *get_template()*, *select_template()* and *from_string()*.

In a Django project where the *TEMPLATES* setting defines exactly one *DjangoTemplates* engine, it's possible to instantiate a *Template* directly.

class `Template`
This class lives at `django.template.Template`. The constructor takes one argument — the raw template code:

```
from django.template import Template

template = Template("My name is {{ my_name }}.")
```

Behind the scenes

The system only parses your raw template code once – when you create the `Template` object. From then on, it's stored internally as a tree structure for performance.

Even the parsing itself is quite fast. Most of the parsing happens via a single call to a single, short, regular expression.

Rendering a context

Once you have a compiled *Template* object, you can render a context with it. You can reuse the same template to render it several times with different contexts.

class Context(*dict_=None*)

This class lives at `django.template.Context`. The constructor takes two optional arguments:

- A dictionary mapping variable names to variable values.
- The name of the current application. This application name is used to help *resolve namespaced URLs*. If you're not using namespaced URLs, you can ignore this argument.

For details, see *Playing with Context objects* below.

`Template.`**render**(*context*)

Call the *Template* object's `render()` method with a *Context* to "fill" the template:

```
>>> from django.template import Context, Template
>>> template = Template("My name is {{ my_name }}.")

>>> context = Context({"my_name": "Adrian"})
>>> template.render(context)
"My name is Adrian."

>>> context = Context({"my_name": "Dolores"})
>>> template.render(context)
"My name is Dolores."
```

Variables and lookups

Variable names must consist of any letter (A-Z), any digit (0-9), an underscore (but they must not start with an underscore) or a dot.

Dots have a special meaning in template rendering. A dot in a variable name signifies a **lookup**. Specifically, when the template system encounters a dot in a variable name, it tries the following lookups, in this order:

- Dictionary lookup. Example: `foo["bar"]`
- Attribute lookup. Example: `foo.bar`
- List-index lookup. Example: `foo[bar]`

Note that "bar" in a template expression like `{{ foo.bar }}` will be interpreted as a literal string and not using the value of the variable "bar", if one exists in the template context.

The template system uses the first lookup type that works. It's short-circuit logic. Here are a few examples:

```
>>> from django.template import Context, Template
>>> t = Template("My name is {{ person.first_name }}.")
>>> d = {"person": {"first_name": "Joe", "last_name": "Johnson"}}
>>> t.render(Context(d))
"My name is Joe."

>>> class PersonClass: pass
>>> p = PersonClass()
```

```
>>> p.first_name = "Ron"
>>> p.last_name = "Nasty"
>>> t.render(Context({"person": p}))
"My name is Ron."

>>> t = Template("The first stooge in the list is {{ stooges.0 }}.")
>>> c = Context({"stooges": ["Larry", "Curly", "Moe"]})
>>> t.render(c)
"The first stooge in the list is Larry."
```

If any part of the variable is callable, the template system will try calling it. Example:

```
>>> class PersonClass2:
...     def name(self):
...         return "Samantha"
>>> t = Template("My name is {{ person.name }}.")
>>> t.render(Context({"person": PersonClass2}))
"My name is Samantha."
```

Callable variables are slightly more complex than variables which only require straight lookups. Here are some things to keep in mind:

- If the variable raises an exception when called, the exception will be propagated, unless the exception has an attribute `silent_variable_failure` whose value is `True`. If the exception *does* have a `silent_variable_failure` attribute whose value is `True`, the variable will render as the value of the engine's `string_if_invalid` configuration option (an empty string, by default). Example:

```
>>> t = Template("My name is {{ person.first_name }}.")
>>> class PersonClass3:
...     def first_name(self):
...         raise AssertionError("foo")
>>> p = PersonClass3()
>>> t.render(Context({"person": p}))
Traceback (most recent call last):
...
AssertionError: foo

>>> class SilentAssertionError(Exception):
...     silent_variable_failure = True
>>> class PersonClass4:
...     def first_name(self):
...         raise SilentAssertionError
>>> p = PersonClass4()
>>> t.render(Context({"person": p}))
"My name is ."
```

  Note that *django.core.exceptions.ObjectDoesNotExist*, which is the base class for all Django database API `DoesNotExist` exceptions, has `silent_variable_failure = True`. So if you're using Django templates with Django model objects, any `DoesNotExist` exception will fail silently.

- A variable can only be called if it has no required arguments. Otherwise, the system will return the value of the engine's `string_if_invalid` option.

- Obviously, there can be side effects when calling some variables, and it'd be either foolish or a security hole to allow the template system to access them.

  A good example is the *delete()* method on each Django model object. The template system shouldn't be allowed to do something like this:

```
I will now delete this valuable data. {{ data.delete }}
```

To prevent this, set an `alters_data` attribute on the callable variable. The template system won't call a variable if it has `alters_data=True` set, and will instead replace the variable with `string_if_invalid`, unconditionally. The dynamically-generated *`delete()`* and *`save()`* methods on Django model objects get `alters_data=True` automatically. Example:

```
def sensitive_function(self):
    self.database_record.delete()
sensitive_function.alters_data = True
```

- Occasionally you may want to turn off this feature for other reasons, and tell the template system to leave a variable uncalled no matter what. To do so, set a `do_not_call_in_templates` attribute on the callable with the value `True`. The template system then will act as if your variable is not callable (allowing you to access attributes of the callable, for example).

How invalid variables are handled

Generally, if a variable doesn't exist, the template system inserts the value of the engine's `string_if_invalid` configuration option, which is set to `''` (the empty string) by default.

Filters that are applied to an invalid variable will only be applied if `string_if_invalid` is set to `''` (the empty string). If `string_if_invalid` is set to any other value, variable filters will be ignored.

This behavior is slightly different for the `if`, `for` and `regroup` template tags. If an invalid variable is provided to one of these template tags, the variable will be interpreted as `None`. Filters are always applied to invalid variables within these template tags.

If `string_if_invalid` contains a `'%s'`, the format marker will be replaced with the name of the invalid variable.

For debug purposes only!

While `string_if_invalid` can be a useful debugging tool, it is a bad idea to turn it on as a 'development default'.

Many templates, including those in the Admin site, rely upon the silence of the template system when a non-existent variable is encountered. If you assign a value other than `''` to `string_if_invalid`, you will experience rendering problems with these templates and sites.

Generally, `string_if_invalid` should only be enabled in order to debug a specific template problem, then cleared once debugging is complete.

Built-in variables

Every context contains `True`, `False` and `None`. As you would expect, these variables resolve to the corresponding Python objects.

Limitations with string literals

Django's template language has no way to escape the characters used for its own syntax. For example, the *`templatetag`* tag is required if you need to output character sequences like `{%` and `%}`.

A similar issue exists if you want to include these sequences in template filter or tag arguments. For example, when parsing a block tag, Django's template parser looks for the first occurrence of `%}` after a `{%`. This prevents the use of `"%}"` as a string literal. For example, a `TemplateSyntaxError` will be raised for the following expressions:

```
{% include "template.html" tvar="Some string literal with %} in it." %}

{% with tvar="Some string literal with %} in it." %}{% endwith %}
```

The same issue can be triggered by using a reserved sequence in filter arguments:

```
{{ some.variable|default:"}}" }}
```

If you need to use strings with these sequences, store them in template variables or use a custom template tag or filter to workaround the limitation.

Playing with `Context` objects

Most of the time, you'll instantiate *`Context`* objects by passing in a fully-populated dictionary to `Context()`. But you can add and delete items from a `Context` object once it's been instantiated, too, using standard dictionary syntax:

```
>>> from django.template import Context
>>> c = Context({"foo": "bar"})
>>> c['foo']
'bar'
>>> del c['foo']
>>> c['foo']
Traceback (most recent call last):
...
KeyError: 'foo'
>>> c['newvariable'] = 'hello'
>>> c['newvariable']
'hello'
```

`Context.`**`get`**(*key*, *otherwise=None*)
: Returns the value for `key` if `key` is in the context, else returns `otherwise`.

`Context.`**`setdefault`**(*key*, *default=None*)
: If `key` is in the context, returns its value. Otherwise inserts `key` with a value of `default` and returns `default`.

`Context.`**`pop`**`()`

`Context.`**`push`**`()`

exception `ContextPopException`

A `Context` object is a stack. That is, you can `push()` and `pop()` it. If you `pop()` too much, it'll raise `django.template.ContextPopException`:

```
>>> c = Context()
>>> c['foo'] = 'first level'
>>> c.push()
{}
>>> c['foo'] = 'second level'
>>> c['foo']
'second level'
>>> c.pop()
{'foo': 'second level'}
>>> c['foo']
'first level'
>>> c['foo'] = 'overwritten'
>>> c['foo']
'overwritten'
```

```
>>> c.pop()
Traceback (most recent call last):
...
ContextPopException
```

You can also use `push()` as a context manager to ensure a matching `pop()` is called.

```
>>> c = Context()
>>> c['foo'] = 'first level'
>>> with c.push():
...     c['foo'] = 'second level'
...     c['foo']
'second level'
>>> c['foo']
'first level'
```

All arguments passed to `push()` will be passed to the `dict` constructor used to build the new context level.

```
>>> c = Context()
>>> c['foo'] = 'first level'
>>> with c.push(foo='second level'):
...     c['foo']
'second level'
>>> c['foo']
'first level'
```

`Context.`**`update`**(*other_dict*)

In addition to `push()` and `pop()`, the `Context` object also defines an `update()` method. This works like `push()` but takes a dictionary as an argument and pushes that dictionary onto the stack instead of an empty one.

```
>>> c = Context()
>>> c['foo'] = 'first level'
>>> c.update({'foo': 'updated'})
{'foo': 'updated'}
>>> c['foo']
'updated'
>>> c.pop()
{'foo': 'updated'}
>>> c['foo']
'first level'
```

Like `push()`, you can use `update()` as a context manager to ensure a matching `pop()` is called.

```
>>> c = Context()
>>> c['foo'] = 'first level'
>>> with c.update({'foo': 'second level'}):
...     c['foo']
'second level'
>>> c['foo']
'first level'
```

The ability to use `update()` as a context manager was added.

Using a `Context` as a stack comes in handy in *some custom template tags*.

`Context.`**`flatten`**`()`

Using `flatten()` method you can get whole `Context` stack as one dictionary including builtin variables.

```
>>> c = Context()
>>> c['foo'] = 'first level'
>>> c.update({'bar': 'second level'})
{'bar': 'second level'}
>>> c.flatten()
{'True': True, 'None': None, 'foo': 'first level', 'False': False, 'bar': 'second level'}
```

A `flatten()` method is also internally used to make `Context` objects comparable.

```
>>> c1 = Context()
>>> c1['foo'] = 'first level'
>>> c1['bar'] = 'second level'
>>> c2 = Context()
>>> c2.update({'bar': 'second level', 'foo': 'first level'})
{'foo': 'first level', 'bar': 'second level'}
>>> c1 == c2
True
```

Result from `flatten()` can be useful in unit tests to compare `Context` against `dict`:

```
class ContextTest(unittest.TestCase):
    def test_against_dictionary(self):
        c1 = Context()
        c1['update'] = 'value'
        self.assertEqual(c1.flatten(), {
            'True': True,
            'None': None,
            'False': False,
            'update': 'value',
        })
```

Using `RequestContext`

class `RequestContext`(*request*, *dict_=None*, *processors=None*)

Django comes with a special `Context` class, `django.template.RequestContext`, that acts slightly differently from the normal `django.template.Context`. The first difference is that it takes an *`HttpRequest`* as its first argument. For example:

```
c = RequestContext(request, {
    'foo': 'bar',
})
```

The second difference is that it automatically populates the context with a few variables, according to the engine's `context_processors` configuration option.

The `context_processors` option is a list of callables – called **context processors** – that take a request object as their argument and return a dictionary of items to be merged into the context. In the default generated settings file, the default template engine contains the following context processors:

```
[
    'django.template.context_processors.debug',
    'django.template.context_processors.request',
    'django.contrib.auth.context_processors.auth',
    'django.contrib.messages.context_processors.messages',
]
```

In addition to these, *RequestContext* always enables `'django.template.context_processors.csrf'`. This is a security related context processor required by the admin and other contrib apps, and, in case of accidental misconfiguration, it is deliberately hardcoded in and cannot be turned off in the `context_processors` option.

Each processor is applied in order. That means, if one processor adds a variable to the context and a second processor adds a variable with the same name, the second will override the first. The default processors are explained below.

When context processors are applied

Context processors are applied on top of context data. This means that a context processor may overwrite variables you've supplied to your *Context* or *RequestContext*, so take care to avoid variable names that overlap with those supplied by your context processors.

If you want context data to take priority over context processors, use the following pattern:

```
from django.template import RequestContext

request_context = RequestContext(request)
request_context.push({"my_name": "Adrian"})
```

Django does this to allow context data to override context processors in APIs such as *render()* and *TemplateResponse*.

Also, you can give *RequestContext* a list of additional processors, using the optional, third positional argument, `processors`. In this example, the *RequestContext* instance gets a `ip_address` variable:

```
from django.http import HttpResponse
from django.template import RequestContext, Template

def ip_address_processor(request):
    return {'ip_address': request.META['REMOTE_ADDR']}

def client_ip_view(request):
    template = Template('{{ title }}: {{ ip_address }}')
    context = RequestContext(request, {
        'title': 'Your IP Address',
    }, [ip_address_processor])
    return HttpResponse(template.render(context))
```

Built-in template context processors

Here's what each of the built-in processors does:

django.contrib.auth.context_processors.auth
auth()
If this processor is enabled, every `RequestContext` will contain these variables:

- `user` – An `auth.User` instance representing the currently logged-in user (or an `AnonymousUser` instance, if the client isn't logged in).
- `perms` – An instance of `django.contrib.auth.context_processors.PermWrapper`, representing the permissions that the currently logged-in user has.

django.template.context_processors.debug
debug()

If this processor is enabled, every `RequestContext` will contain these two variables – but only if your *DEBUG* setting is set to `True` and the request's IP address (`request.META['REMOTE_ADDR']`) is in the *INTERNAL_IPS* setting:

- `debug` – `True`. You can use this in templates to test whether you're in *DEBUG* mode.
- `sql_queries` – A list of `{'sql': ..., 'time': ...}` dictionaries, representing every SQL query that has happened so far during the request and how long it took. The list is in order by database alias and then by query. It's lazily generated on access.

In older versions, only the queries for the default database alias were included.

django.template.context_processors.i18n If this processor is enabled, every `RequestContext` will contain these two variables:

- `LANGUAGES` – The value of the *LANGUAGES* setting.
- `LANGUAGE_CODE` – `request.LANGUAGE_CODE`, if it exists. Otherwise, the value of the *LANGUAGE_CODE* setting.

See Internationalization and localization for more.

django.template.context_processors.media If this processor is enabled, every `RequestContext` will contain a variable `MEDIA_URL`, providing the value of the *MEDIA_URL* setting.

django.template.context_processors.static
static()
If this processor is enabled, every `RequestContext` will contain a variable `STATIC_URL`, providing the value of the *STATIC_URL* setting.

django.template.context_processors.csrf This processor adds a token that is needed by the *csrf_token* template tag for protection against Cross Site Request Forgeries.

django.template.context_processors.request If this processor is enabled, every `RequestContext` will contain a variable `request`, which is the current *HttpRequest*.

django.template.context_processors.tz
tz()
If this processor is enabled, every `RequestContext` will contain a variable `TIME_ZONE`, providing the name of the currently active time zone.

django.contrib.messages.context_processors.messages If this processor is enabled, every `RequestContext` will contain these two variables:

- `messages` – A list of messages (as strings) that have been set via the messages framework.
- `DEFAULT_MESSAGE_LEVELS` – A mapping of the message level names to *their numeric value*.

Writing your own context processors

A context processor has a very simple interface: It's a Python function that takes one argument, an *HttpRequest* object, and returns a dictionary that gets added to the template context. Each context processor *must* return a dictionary.

Custom context processors can live anywhere in your code base. All Django cares about is that your custom context processors are pointed to by the `'context_processors'` option in your *TEMPLATES* setting — or the `context_processors` argument of *Engine* if you're using it directly.

Loading templates

Generally, you'll store templates in files on your filesystem rather than using the low-level *Template* API yourself. Save templates in a directory specified as a **template directory**.

Django searches for template directories in a number of places, depending on your template loading settings (see "Loader types" below), but the most basic way of specifying template directories is by using the *DIRS* option.

The `DIRS` option

Tell Django what your template directories are by using the *DIRS* option in the *TEMPLATES* setting in your settings file — or the `dirs` argument of *Engine*. This should be set to a list of strings that contain full paths to your template directories:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            '/home/html/templates/lawrence.com',
            '/home/html/templates/default',
        ],
    },
]
```

Your templates can go anywhere you want, as long as the directories and templates are readable by the Web server. They can have any extension you want, such as `.html` or `.txt`, or they can have no extension at all.

Note that these paths should use Unix-style forward slashes, even on Windows.

Loader types

By default, Django uses a filesystem-based template loader, but Django comes with a few other template loaders, which know how to load templates from other sources.

Some of these other loaders are disabled by default, but you can activate them by adding a `'loaders'` option to your `DjangoTemplates` backend in the *TEMPLATES* setting or passing a `loaders` argument to *Engine*. `loaders` should be a list of strings or tuples, where each represents a template loader class. Here are the template loaders that come with Django:

`django.template.loaders.filesystem.Loader`

class `filesystem.`**`Loader`**

Loads templates from the filesystem, according to *DIRS*.

This loader is enabled by default. However it won't find any templates until you set *DIRS* to a non-empty list:

```
    TEMPLATES = [{
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
    }]
```

`django.template.loaders.app_directories.Loader`

class `app_directories.`**Loader**

Loads templates from Django apps on the filesystem. For each app in *INSTALLED_APPS*, the loader looks for a `templates` subdirectory. If the directory exists, Django looks for templates in there.

This means you can store templates with your individual apps. This also makes it easy to distribute Django apps with default templates.

For example, for this setting:

```
    INSTALLED_APPS = ['myproject.polls', 'myproject.music']
```

...then `get_template('foo.html')` will look for `foo.html` in these directories, in this order:

- `/path/to/myproject/polls/templates/`
- `/path/to/myproject/music/templates/`

... and will use the one it finds first.

The order of *INSTALLED_APPS* is significant! For example, if you want to customize the Django admin, you might choose to override the standard `admin/base_site.html` template, from `django.contrib.admin`, with your own `admin/base_site.html` in `myproject.polls`. You must then make sure that your `myproject.polls` comes *before* `django.contrib.admin` in *INSTALLED_APPS*, otherwise `django.contrib.admin`'s will be loaded first and yours will be ignored.

Note that the loader performs an optimization when it first runs: it caches a list of which *INSTALLED_APPS* packages have a `templates` subdirectory.

You can enable this loader simply by setting *APP_DIRS* to `True`:

```
    TEMPLATES = [{
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'APP_DIRS': True,
    }]
```

`django.template.loaders.eggs.Loader`

class `eggs.`**Loader**

1.9 : Distributing applications as eggs is not recommended.

Just like `app_directories` above, but it loads templates from Python eggs rather than from the filesystem.

This loader is disabled by default.

`django.template.loaders.cached.Loader`

class `cached.`**Loader**

By default, the templating system will read and compile your templates every time they need to be rendered. While the Django templating system is quite fast, the overhead from reading and compiling templates can add up.

The cached template loader is a class-based loader that you configure with a list of other loaders that it should wrap. The wrapped loaders are used to locate unknown templates when they are first encountered. The cached loader then stores the compiled `Template` in memory. The cached `Template` instance is returned for subsequent requests to load the same template.

For example, to enable template caching with the `filesystem` and `app_directories` template loaders you might use the following settings:

```
TEMPLATES = [{
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'DIRS': [os.path.join(BASE_DIR, 'templates')],
    'OPTIONS': {
        'loaders': [
            ('django.template.loaders.cached.Loader', [
                'django.template.loaders.filesystem.Loader',
                'django.template.loaders.app_directories.Loader',
            ]),
        ],
    },
}]
```

: All of the built-in Django template tags are safe to use with the cached loader, but if you're using custom template tags that come from third party packages, or that you wrote yourself, you should ensure that the `Node` implementation for each tag is thread-safe. For more information, see *template tag thread safety considerations*.

This loader is disabled by default.

`django.template.loaders.locmem.Loader`

class `locmem.`**`Loader`**

Loads templates from a Python dictionary. This is useful for testing.

This loader takes a dictionary of templates as its first argument:

```
TEMPLATES = [{
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'OPTIONS': {
        'loaders': [
            ('django.template.loaders.locmem.Loader', {
                'index.html': 'content here',
            }),
        ],
    },
}]
```

This loader is disabled by default.

Django uses the template loaders in order according to the `'loaders'` option. It uses each loader until a loader finds a match.

Custom loaders

It's possible to load templates from additional sources using custom template loaders. Custom `Loader` classes should inherit from `django.template.loaders.base.Loader` and define the `get_contents()` and `get_template_sources()` methods.

In previous versions of Django, custom loaders defined a single method: `load_template_source()`.

Loader methods

class Loader

Loads templates from a given source, such as the filesystem or a database.

get_template_sources(*template_name*)

A method that takes a `template_name` and yields *Origin* instances for each possible source.

For example, the filesystem loader may receive `'index.html'` as a `template_name` argument. This method would yield origins for the full path of `index.html` as it appears in each template directory the loader looks at.

The method doesn't need to verify that the template exists at a given path, but it should ensure the path is valid. For instance, the filesystem loader makes sure the path lies under a valid template directory.

get_contents(*origin*)

Returns the contents for a template given a *Origin* instance.

This is where a filesystem loader would read contents from the filesystem, or a database loader would read from the database. If a matching template doesn't exist, this should raise a *TemplateDoesNotExist* error.

get_template(*template_name*, *skip=None*)

Returns a `Template` object for a given `template_name` by looping through results from *get_template_sources()* and calling *get_contents()*. This returns the first matching template. If no template is found, *TemplateDoesNotExist* is raised.

The optional `skip` argument is a list of origins to ignore when extending templates. This allow templates to extend other templates of the same name. It also used to avoid recursion errors.

In general, it is enough to define *get_template_sources()* and *get_contents()* for custom template loaders. `get_template()` will usually not need to be overridden.

load_template_source(*template_name*, *template_dirs=None*)

Returns a tuple of (`template_string`, `template_origin`), where `template_string` is a string containing the template contents, and `template_origin` is a string identifying the template source. A filesystem-based loader may return the full path to the file as the `template_origin`, for example.

`template_dirs` is an optional argument used to control which directories the loader will search.

This method is called automatically by *load_template()* and should be overridden when writing custom template loaders.

1.9 : Custom loaders should use *get_template()* and *get_contents()* instead.

load_template(*template_name*, *template_dirs=None*)

Returns a tuple of (`template`, `template_origin`), where `template` is a `Template` object and `template_origin` is a string identifying the template source. A filesystem-based loader may return the full path to the file as the `template_origin`, for example.

1.9 : Custom loaders should use *get_template()* and *get_contents()* instead.

Building your own

For examples, read the source code for Django's built-in loaders.

Template origin

Templates have an `origin` containing attributes depending on the source they are loaded from.

Django used to create an origin based on `django.template.loader.LoaderOrigin` or `django.template.base.StringOrigin`. These have been replaced by `django.template.base.Origin`.

class `Origin`

`name`
The path to the template as returned by the template loader. For loaders that read from the file system, this is the full path to the template.

If the template is instantiated directly rather than through a template loader, this is a string value of `<unknown_source>`.

`template_name`
The relative path to the template as passed into the template loader.

If the template is instantiated directly rather than through a template loader, this is `None`.

6.20.4 Upgrading templates to Django 1.8

Django's template system was overhauled in Django 1.8 when it gained support for multiple template engines. This document complements the release notes with detailed upgrade instructions on some topics.

The `TEMPLATES` settings

A new setting was introduced in Django 1.8: *`TEMPLATES`*. All existing template-related settings were deprecated.

During the deprecation period, Django will create a backwards-compatible *`TEMPLATES`* based on the `TEMPLATE_*` settings if you don't define it yourself.

Here's how to define *`TEMPLATES`* in your settings module.

If you're using the default value of `TEMPLATE_LOADERS`, that is, if it isn't defined in your settings file or if it's set to:

```
['django.template.loaders.filesystem.Loader',
 'django.template.loaders.app_directories.Loader']
```

then you should define *`TEMPLATES`* as follows:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            # insert your TEMPLATE_DIRS here
        ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                # Insert your TEMPLATE_CONTEXT_PROCESSORS here or use this
                # list if you haven't customized them:
                'django.contrib.auth.context_processors.auth',
                'django.template.context_processors.debug',
                'django.template.context_processors.i18n',
```

```
                'django.template.context_processors.media',
                'django.template.context_processors.static',
                'django.template.context_processors.tz',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

If you aren't using the default value of TEMPLATE_LOADERS, then you should define *TEMPLATES* as follows:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            # insert your TEMPLATE_DIRS here
        ],
        'OPTIONS': {
            'context_processors': [
                # Insert your TEMPLATE_CONTEXT_PROCESSORS here or use this
                # list if you haven't customized them:
                'django.contrib.auth.context_processors.auth',
                'django.template.context_processors.debug',
                'django.template.context_processors.i18n',
                'django.template.context_processors.media',
                'django.template.context_processors.static',
                'django.template.context_processors.tz',
                'django.contrib.messages.context_processors.messages',
            ],
            'loaders': [
                # insert your TEMPLATE_LOADERS here
            ]
        },
    },
]
```

Furthermore you should replace `django.core.context_processors` with `django.template.context_processors` in the names of context processors.

If your settings module defines ALLOWED_INCLUDE_ROOTS or TEMPLATE_STRING_IF_INVALID, include their values under the `'allowed_include_roots'` and `'string_if_invalid'` keys in the `'OPTIONS'` dictionary.

If it sets TEMPLATE_DEBUG to a value that differs from *DEBUG*, include that value under the `'debug'` key in `'OPTIONS'`.

Once you have defined *TEMPLATES*, you can safely remove ALLOWED_INCLUDE_ROOTS, TEMPLATE_CONTEXT_PROCESSORS, TEMPLATE_DEBUG, TEMPLATE_DIRS, TEMPLATE_LOADERS, and TEMPLATE_STRING_IF_INVALID.

If you are overriding some of these settings in tests, you should override the entire *TEMPLATES* setting instead.

django.template.loader

get_template() and select_template()

In Django 1.8 *get_template()* and *select_template()* return a backend-dependent `Template` instead of a *django.template.Template*.

For example, if *get_template()* loads a template with a *DjangoTemplates* backend, then it returns a `django.template.backends.django.Template`.

`Template` objects must provide a *render()* method whose signature differs slightly from the Django template language's *render()*.

Instead of:

```
from django.template import Context
from django.template.loader import get_template

template = get_template('hello.html')
html = template.render(Context({'name': 'world'}))
```

You should write:

```
from django.template.loader import get_template

template = get_template('hello.html')
html = template.render({'name': 'world'})
```

And instead of:

```
from django.template import RequestContext
from django.template.loader import get_template

template = get_template('hello.html')
html = template.render(RequestContext(request, {'name': 'world'}))
```

You should write:

```
from django.template.loader import get_template

template = get_template('hello.html')
html = template.render({'name': 'world'}, request)
```

Passing a *Context* or a *RequestContext* is still possible when the template is loaded by a *DjangoTemplates* backend but it's deprecated and won't be supported in Django 1.10.

If you're loading a template while you're rendering another template with the Django template language and you have access to the current context, for instance in the `render()` method of a template tag, you can use the current *Engine* directly. Instead of:

```
from django.template.loader import get_template
template = get_template('included.html')
```

You can write:

```
template = context.template.engine.get_template('included.html')
```

This will load the template with the current engine without triggering the multiple template engines machinery, which is usually the desired behavior. Unlike previous solutions, this returns a *django.template.Template*, like *get_template()* used to in Django 1.7 and earlier, avoiding all backwards-compatibility problems.

get_template_from_string()

Private API `get_template_from_string(template_code)` was removed in Django 1.8 because it had no way to choose an engine to compile the template.

Three alternatives are available.

If you control the project's setting, you can use one of the configured engines:

```
from django.template import engines

template = engines['django'].from_string(template_code)
```

This returns a backend-dependent `Template` object.

For trivial templates that don't need context processors nor anything else, you can create a bare-bones engine and use its `from_string()` method:

```
from django.template import Engine

template = Engine().from_string(template_code)
```

This returns a *`django.template.Template`* because *`Engine`* is part of the Django template language's APIs. The multiple template engines machinery isn't involved here.

Finally, if you have access to the current context, you can use the same trick as above:

```
template = context.template.engine.from_string(template_code)
```

`Template()`

To a lesser extent, instantiating a template with `Template(template_code)` suffers from the same issue as `get_template_from_string()`.

It still works when the *`TEMPLATES`* setting defines exactly one *`DjangoTemplates`* backend, but pluggable applications can't control this requirement.

The last two solutions described in the previous section are recommended in that case.

:

For information on writing your own custom tags and filters, see Custom template tags and filters.

6.21 `TemplateResponse` and `SimpleTemplateResponse`

Standard *`HttpResponse`* objects are static structures. They are provided with a block of pre-rendered content at time of construction, and while that content can be modified, it isn't in a form that makes it easy to perform modifications.

However, it can sometimes be beneficial to allow decorators or middleware to modify a response *after* it has been constructed by the view. For example, you may want to change the template that is used, or put additional data into the context.

TemplateResponse provides a way to do just that. Unlike basic *`HttpResponse`* objects, TemplateResponse objects retain the details of the template and context that was provided by the view to compute the response. The final output of the response is not computed until it is needed, later in the response process.

6.21.1 `SimpleTemplateResponse` objects

class `SimpleTemplateResponse`

Attributes

`SimpleTemplateResponse.`**`template_name`**
The name of the template to be rendered. Accepts a backend-dependent template object (such as those returned by *`get_template()`*), the name of a template, or a list of template names.

Example: `['foo.html', 'path/to/bar.html']`

`SimpleTemplateResponse.`**`context_data`**
The context data to be used when rendering the template. It must be a `dict`.

Example: `{'foo': 123}`

`SimpleTemplateResponse.`**`rendered_content`**
The current rendered value of the response content, using the current template and context data.

`SimpleTemplateResponse.`**`is_rendered`**
A boolean indicating whether the response content has been rendered.

Methods

`SimpleTemplateResponse.`**`__init__`**(*template*, *context=None*, *content_type=None*, *status=None*, *charset=None*, *using=None*)
Instantiates a *`SimpleTemplateResponse`* object with the given template, context, content type, HTTP status, and charset.

`template` A backend-dependent template object (such as those returned by *`get_template()`*), the name of a template, or a list of template names.

`context` A `dict` of values to add to the template context. By default, this is an empty dictionary.

`content_type` The value included in the HTTP `Content-Type` header, including the MIME type specification and the character set encoding. If `content_type` is specified, then its value is used. Otherwise, *`DEFAULT_CONTENT_TYPE`* is used.

`status` The HTTP status code for the response.

`charset` The charset in which the response will be encoded. If not given it will be extracted from `content_type`, and if that is unsuccessful, the *`DEFAULT_CHARSET`* setting will be used.

`using` The *`NAME`* of a template engine to use for loading the template.

`SimpleTemplateResponse.`**`resolve_context`**(*context*)
Preprocesses context data that will be used for rendering a template. Accepts a `dict` of context data. By default, returns the same `dict`.

Override this method in order to customize the context.

`SimpleTemplateResponse.`**`resolve_template`**(*template*)
Resolves the template instance to use for rendering. Accepts a backend-dependent template object (such as those returned by *`get_template()`*), the name of a template, or a list of template names.

Returns the backend-dependent template object instance to be rendered.

Override this method in order to customize template loading.

`SimpleTemplateResponse.`**`add_post_render_callback`**()
Add a callback that will be invoked after rendering has taken place. This hook can be used to defer certain processing operations (such as caching) until after rendering has occurred.

If the *`SimpleTemplateResponse`* has already been rendered, the callback will be invoked immediately.

When called, callbacks will be passed a single argument – the rendered *SimpleTemplateResponse* instance.

If the callback returns a value that is not `None`, this will be used as the response instead of the original response object (and will be passed to the next post rendering callback etc.)

`SimpleTemplateResponse.`**`render`**`()`
Sets `response.content` to the result obtained by *SimpleTemplateResponse.rendered_content*, runs all post-rendering callbacks, and returns the resulting response object.

`render()` will only have an effect the first time it is called. On subsequent calls, it will return the result obtained from the first call.

6.21.2 `TemplateResponse` objects

class `TemplateResponse`
`TemplateResponse` is a subclass of *SimpleTemplateResponse* that knows about the current *HttpRequest*.

Methods

`TemplateResponse.`**`__init__`**(*request*, *template*, *context=None*, *content_type=None*, *status=None*, *charset=None*, *using=None*)
Instantiates a *TemplateResponse* object with the given request, template, context, content type, HTTP status, and charset.

`request` An *HttpRequest* instance.

`template` A backend-dependent template object (such as those returned by *get_template()*), the name of a template, or a list of template names.

`context` A `dict` of values to add to the template context. By default, this is an empty dictionary.

`content_type` The value included in the HTTP `Content-Type` header, including the MIME type specification and the character set encoding. If `content_type` is specified, then its value is used. Otherwise, *DEFAULT_CONTENT_TYPE* is used.

`status` The HTTP status code for the response.

`charset` The charset in which the response will be encoded. If not given it will be extracted from `content_type`, and if that is unsuccessful, the *DEFAULT_CHARSET* setting will be used.

`using` The *NAME* of a template engine to use for loading the template.

6.21.3 The rendering process

Before a *TemplateResponse* instance can be returned to the client, it must be rendered. The rendering process takes the intermediate representation of template and context, and turns it into the final byte stream that can be served to the client.

There are three circumstances under which a `TemplateResponse` will be rendered:

- When the `TemplateResponse` instance is explicitly rendered, using the *SimpleTemplateResponse.render()* method.
- When the content of the response is explicitly set by assigning `response.content`.
- After passing through template response middleware, but before passing through response middleware.

A TemplateResponse can only be rendered once. The first call to *SimpleTemplateResponse.render()* sets the content of the response; subsequent rendering calls do not change the response content.

However, when response.content is explicitly assigned, the change is always applied. If you want to force the content to be re-rendered, you can re-evaluate the rendered content, and assign the content of the response manually:

```
# Set up a rendered TemplateResponse
>>> from django.template.response import TemplateResponse
>>> t = TemplateResponse(request, 'original.html', {})
>>> t.render()
>>> print(t.content)
Original content

# Re-rendering doesn't change content
>>> t.template_name = 'new.html'
>>> t.render()
>>> print(t.content)
Original content

# Assigning content does change, no render() call required
>>> t.content = t.rendered_content
>>> print(t.content)
New content
```

Post-render callbacks

Some operations – such as caching – cannot be performed on an unrendered template. They must be performed on a fully complete and rendered response.

If you're using middleware, the solution is easy. Middleware provides multiple opportunities to process a response on exit from a view. If you put behavior in the response middleware, it's guaranteed to execute after template rendering has taken place.

However, if you're using a decorator, the same opportunities do not exist. Any behavior defined in a decorator is handled immediately.

To compensate for this (and any other analogous use cases), *TemplateResponse* allows you to register callbacks that will be invoked when rendering has completed. Using this callback, you can defer critical processing until a point where you can guarantee that rendered content will be available.

To define a post-render callback, just define a function that takes a single argument – response – and register that function with the template response:

```
from django.template.response import TemplateResponse

def my_render_callback(response):
    # Do content-sensitive processing
    do_post_processing()

def my_view(request):
    # Create a response
    response = TemplateResponse(request, 'mytemplate.html', {})
    # Register the callback
    response.add_post_render_callback(my_render_callback)
    # Return the response
    return response
```

my_render_callback() will be invoked after the mytemplate.html has been rendered, and will be provided the fully rendered *TemplateResponse* instance as an argument.

If the template has already been rendered, the callback will be invoked immediately.

6.21.4 Using `TemplateResponse` and `SimpleTemplateResponse`

A *`TemplateResponse`* object can be used anywhere that a normal *`django.http.HttpResponse`* can be used. It can also be used as an alternative to calling *`render()`*.

For example, the following simple view returns a *`TemplateResponse`* with a simple template and a context containing a queryset:

```
from django.template.response import TemplateResponse

def blog_index(request):
    return TemplateResponse(request, 'entry_list.html', {'entries': Entry.objects.all()})
```

6.22 Unicode data

Django natively supports Unicode data everywhere. Providing your database can somehow store the data, you can safely pass around Unicode strings to templates, models and the database.

This document tells you what you need to know if you're writing applications that use data or templates that are encoded in something other than ASCII.

6.22.1 Creating the database

Make sure your database is configured to be able to store arbitrary string data. Normally, this means giving it an encoding of UTF-8 or UTF-16. If you use a more restrictive encoding – for example, latin1 (iso8859-1) – you won't be able to store certain characters in the database, and information will be lost.

- MySQL users, refer to the MySQL manual for details on how to set or alter the database character set encoding.
- PostgreSQL users, refer to the PostgreSQL manual (section 22.3.2 in PostgreSQL 9) for details on creating databases with the correct encoding.
- Oracle users, refer to the Oracle manual for details on how to set (section 2) or alter (section 11) the database character set encoding.
- SQLite users, there is nothing you need to do. SQLite always uses UTF-8 for internal encoding.

All of Django's database backends automatically convert Unicode strings into the appropriate encoding for talking to the database. They also automatically convert strings retrieved from the database into Python Unicode strings. You don't even need to tell Django what encoding your database uses: that is handled transparently.

For more, see the section "The database API" below.

6.22.2 General string handling

Whenever you use strings with Django – e.g., in database lookups, template rendering or anywhere else – you have two choices for encoding those strings. You can use Unicode strings, or you can use normal strings (sometimes called "bytestrings") that are encoded using UTF-8.

In Python 3, the logic is reversed, that is normal strings are Unicode, and when you want to specifically create a bytestring, you have to prefix the string with a 'b'. As we are doing in Django code from version 1.5, we recommend that you import `unicode_literals` from the __future__ library in your code. Then, when you specifically want to create a bytestring literal, prefix the string with 'b'.

Python 2 legacy:

```
my_string = "This is a bytestring"
my_unicode = u"This is an Unicode string"
```

Python 2 with unicode literals or Python 3:

```
from __future__ import unicode_literals

my_string = b"This is a bytestring"
my_unicode = "This is an Unicode string"
```

See also Python 3 compatibility.

: A bytestring does not carry any information with it about its encoding. For that reason, we have to make an assumption, and Django assumes that all bytestrings are in UTF-8.
If you pass a string to Django that has been encoded in some other format, things will go wrong in interesting ways. Usually, Django will raise a `UnicodeDecodeError` at some point.

If your code only uses ASCII data, it's safe to use your normal strings, passing them around at will, because ASCII is a subset of UTF-8.

Don't be fooled into thinking that if your *DEFAULT_CHARSET* setting is set to something other than `'utf-8'` you can use that other encoding in your bytestrings! *DEFAULT_CHARSET* only applies to the strings generated as the result of template rendering (and email). Django will always assume UTF-8 encoding for internal bytestrings. The reason for this is that the *DEFAULT_CHARSET* setting is not actually under your control (if you are the application developer). It's under the control of the person installing and using your application – and if that person chooses a different setting, your code must still continue to work. Ergo, it cannot rely on that setting.

In most cases when Django is dealing with strings, it will convert them to Unicode strings before doing anything else. So, as a general rule, if you pass in a bytestring, be prepared to receive a Unicode string back in the result.

Translated strings

Aside from Unicode strings and bytestrings, there's a third type of string-like object you may encounter when using Django. The framework's internationalization features introduce the concept of a "lazy translation" – a string that has been marked as translated but whose actual translation result isn't determined until the object is used in a string. This feature is useful in cases where the translation locale is unknown until the string is used, even though the string might have originally been created when the code was first imported.

Normally, you won't have to worry about lazy translations. Just be aware that if you examine an object and it claims to be a `django.utils.functional.__proxy__` object, it is a lazy translation. Calling `unicode()` with the lazy translation as the argument will generate a Unicode string in the current locale.

For more details about lazy translation objects, refer to the internationalization documentation.

Useful utility functions

Because some string operations come up again and again, Django ships with a few useful functions that should make working with Unicode and bytestring objects a bit easier.

Conversion functions

The `django.utils.encoding` module contains a few functions that are handy for converting back and forth between Unicode and bytestrings.

- `smart_text(s, encoding='utf-8', strings_only=False, errors='strict')` converts its input to a Unicode string. The `encoding` parameter specifies the input encoding. (For example, Django uses this internally when processing form input data, which might not be UTF-8 encoded.) The `strings_only` parameter, if set to True, will result in Python numbers, booleans and `None` not being converted to a string (they keep their original types). The `errors` parameter takes any of the values that are accepted by Python's `unicode()` function for its error handling.

  If you pass `smart_text()` an object that has a `__unicode__` method, it will use that method to do the conversion.

- `force_text(s, encoding='utf-8', strings_only=False, errors='strict')` is identical to `smart_text()` in almost all cases. The difference is when the first argument is a *lazy translation* instance. While `smart_text()` preserves lazy translations, `force_text()` forces those objects to a Unicode string (causing the translation to occur). Normally, you'll want to use `smart_text()`. However, `force_text()` is useful in template tags and filters that absolutely *must* have a string to work with, not just something that can be converted to a string.

- `smart_bytes(s, encoding='utf-8', strings_only=False, errors='strict')` is essentially the opposite of `smart_text()`. It forces the first argument to a bytestring. The `strings_only` parameter has the same behavior as for `smart_text()` and `force_text()`. This is slightly different semantics from Python's builtin `str()` function, but the difference is needed in a few places within Django's internals.

Normally, you'll only need to use `smart_text()`. Call it as early as possible on any input data that might be either Unicode or a bytestring, and from then on, you can treat the result as always being Unicode.

URI and IRI handling

Web frameworks have to deal with URLs (which are a type of IRI). One requirement of URLs is that they are encoded using only ASCII characters. However, in an international environment, you might need to construct a URL from an IRI – very loosely speaking, a URI that can contain Unicode characters. Quoting and converting an IRI to URI can be a little tricky, so Django provides some assistance.

- The function *`django.utils.encoding.iri_to_uri()`* implements the conversion from IRI to URI as required by the specification (**RFC 3987#section-3.1**).

- The functions *`django.utils.http.urlquote()`* and *`django.utils.http.urlquote_plus()`* are versions of Python's standard `urllib.quote()` and `urllib.quote_plus()` that work with non-ASCII characters. (The data is converted to UTF-8 prior to encoding.)

These two groups of functions have slightly different purposes, and it's important to keep them straight. Normally, you would use `urlquote()` on the individual portions of the IRI or URI path so that any reserved characters such as '&' or '%' are correctly encoded. Then, you apply `iri_to_uri()` to the full IRI and it converts any non-ASCII characters to the correct encoded values.

: Technically, it isn't correct to say that `iri_to_uri()` implements the full algorithm in the IRI specification. It doesn't (yet) perform the international domain name encoding portion of the algorithm.

The `iri_to_uri()` function will not change ASCII characters that are otherwise permitted in a URL. So, for example, the character '%' is not further encoded when passed to `iri_to_uri()`. This means you can pass a full URL to this function and it will not mess up the query string or anything like that.

An example might clarify things here:

```
>>> urlquote('Paris & Orléans')
'Paris%20%26%20Orl%C3%A9ans'
```

```
>>> iri_to_uri('/favorites/François/%s' % urlquote('Paris & Orléans'))
'/favorites/Fran%C3%A7ois/Paris%20%26%20Orl%C3%A9ans'
```

If you look carefully, you can see that the portion that was generated by `urlquote()` in the second example was not double-quoted when passed to `iri_to_uri()`. This is a very important and useful feature. It means that you can construct your IRI without worrying about whether it contains non-ASCII characters and then, right at the end, call `iri_to_uri()` on the result.

Similarly, Django provides *`django.utils.encoding.uri_to_iri()`* which implements the conversion from URI to IRI as per **RFC 3987#section-3.2**. It decodes all percent-encodings except those that don't represent a valid UTF-8 sequence.

An example to demonstrate:

```
>>> uri_to_iri('/%E2%99%A5%E2%99%A5/?utf8=%E2%9C%93')
'/[unicode-heart][unicode-heart]/?utf8=[unicode-checkmark]'
>>> uri_to_iri('%A9helloworld')
'%A9helloworld'
```

In the first example, the UTF-8 characters and reserved characters are unquoted. In the second, the percent-encoding remains unchanged because it lies outside the valid UTF-8 range.

Both `iri_to_uri()` and `uri_to_iri()` functions are idempotent, which means the following is always true:

```
iri_to_uri(iri_to_uri(some_string)) == iri_to_uri(some_string)
uri_to_iri(uri_to_iri(some_string)) == uri_to_iri(some_string)
```

So you can safely call it multiple times on the same URI/IRI without risking double-quoting problems.

6.22.3 Models

Because all strings are returned from the database as Unicode strings, model fields that are character based (CharField, TextField, URLField, etc.) will contain Unicode values when Django retrieves data from the database. This is *always* the case, even if the data could fit into an ASCII bytestring.

You can pass in bytestrings when creating a model or populating a field, and Django will convert it to Unicode when it needs to.

Choosing between `__str__()` and `__unicode__()`

: If you are on Python 3, you can skip this section because you'll always create `__str__()` rather than `__unicode__()`. If you'd like compatibility with Python 2, you can decorate your model class with *`python_2_unicode_compatible()`*.

One consequence of using Unicode by default is that you have to take some care when printing data from the model.

In particular, rather than giving your model a `__str__()` method, we recommended you implement a `__unicode__()` method. In the `__unicode__()` method, you can quite safely return the values of all your fields without having to worry about whether they fit into a bytestring or not. (The way Python works, the result of `__str__()` is *always* a bytestring, even if you accidentally try to return a Unicode object).

You can still create a `__str__()` method on your models if you want, of course, but you shouldn't need to do this unless you have a good reason. Django's `Model` base class automatically provides a `__str__()` implementation that calls `__unicode__()` and encodes the result into UTF-8. This means you'll normally only need to implement a `__unicode__()` method and let Django handle the coercion to a bytestring when required.

Taking care in `get_absolute_url()`

URLs can only contain ASCII characters. If you're constructing a URL from pieces of data that might be non-ASCII, be careful to encode the results in a way that is suitable for a URL. The *reverse()* function handles this for you automatically.

If you're constructing a URL manually (i.e., *not* using the `reverse()` function), you'll need to take care of the encoding yourself. In this case, use the `iri_to_uri()` and `urlquote()` functions that were documented *above*. For example:

```
from django.utils.encoding import iri_to_uri
from django.utils.http import urlquote

def get_absolute_url(self):
    url = '/person/%s/?x=0&y=0' % urlquote(self.location)
    return iri_to_uri(url)
```

This function returns a correctly encoded URL even if `self.location` is something like "Jack visited Paris & Orléans". (In fact, the `iri_to_uri()` call isn't strictly necessary in the above example, because all the non-ASCII characters would have been removed in quoting in the first line.)

6.22.4 The database API

You can pass either Unicode strings or UTF-8 bytestrings as arguments to `filter()` methods and the like in the database API. The following two querysets are identical:

```
from __future__ import unicode_literals

qs = People.objects.filter(name__contains='Å')
qs = People.objects.filter(name__contains=b'\xc3\x85') # UTF-8 encoding of Å
```

6.22.5 Templates

You can use either Unicode or bytestrings when creating templates manually:

```
from __future__ import unicode_literals
from django.template import Template
t1 = Template(b'This is a bytestring template.')
t2 = Template('This is a Unicode template.')
```

But the common case is to read templates from the filesystem, and this creates a slight complication: not all filesystems store their data encoded as UTF-8. If your template files are not stored with a UTF-8 encoding, set the *FILE_CHARSET* setting to the encoding of the files on disk. When Django reads in a template file, it will convert the data from this encoding to Unicode. (*FILE_CHARSET* is set to `'utf-8'` by default.)

The *DEFAULT_CHARSET* setting controls the encoding of rendered templates. This is set to UTF-8 by default.

Template tags and filters

A couple of tips to remember when writing your own template tags and filters:

- Always return Unicode strings from a template tag's `render()` method and from template filters.
- Use `force_text()` in preference to `smart_text()` in these places. Tag rendering and filter calls occur as the template is being rendered, so there is no advantage to postponing the conversion of lazy translation objects into strings. It's easier to work solely with Unicode strings at that point.

6.22.6 Files

If you intend to allow users to upload files, you must ensure that the environment used to run Django is configured to work with non-ASCII file names. If your environment isn't configured correctly, you'll encounter `UnicodeEncodeError` exceptions when saving files with file names that contain non-ASCII characters.

Filesystem support for UTF-8 file names varies and might depend on the environment. Check your current configuration in an interactive Python shell by running:

```
import sys
sys.getfilesystemencoding()
```

This should output "UTF-8".

The `LANG` environment variable is responsible for setting the expected encoding on Unix platforms. Consult the documentation for your operating system and application server for the appropriate syntax and location to set this variable.

In your development environment, you might need to add a setting to your `~.bashrc` analogous to::

```
export LANG="en_US.UTF-8"
```

6.22.7 Email

Django's email framework (in `django.core.mail`) supports Unicode transparently. You can use Unicode data in the message bodies and any headers. However, you're still obligated to respect the requirements of the email specifications, so, for example, email addresses should use only ASCII characters.

The following code example demonstrates that everything except email addresses can be non-ASCII:

```
from __future__ import unicode_literals
from django.core.mail import EmailMessage

subject = 'My visit to Sør-Trøndelag'
sender = 'Arnbjörg Ráðormsdóttir <arnbjorg@example.com>'
recipients = ['Fred <fred@example.com']
body = '...'
msg = EmailMessage(subject, body, sender, recipients)
msg.attach("Une pièce jointe.pdf", "%PDF-1.4.%...", mimetype="application/pdf")
msg.send()
```

6.22.8 Form submission

HTML form submission is a tricky area. There's no guarantee that the submission will include encoding information, which means the framework might have to guess at the encoding of submitted data.

Django adopts a "lazy" approach to decoding form data. The data in an `HttpRequest` object is only decoded when you access it. In fact, most of the data is not decoded at all. Only the `HttpRequest.GET` and `HttpRequest.POST` data structures have any decoding applied to them. Those two fields will return their members as Unicode data. All other attributes and methods of `HttpRequest` return data exactly as it was submitted by the client.

By default, the *`DEFAULT_CHARSET`* setting is used as the assumed encoding for form data. If you need to change this for a particular form, you can set the `encoding` attribute on an `HttpRequest` instance. For example:

```
def some_view(request):
    # We know that the data must be encoded as KOI8-R (for some reason).
    request.encoding = 'koi8-r'
    ...
```

You can even change the encoding after having accessed `request.GET` or `request.POST`, and all subsequent accesses will use the new encoding.

Most developers won't need to worry about changing form encoding, but this is a useful feature for applications that talk to legacy systems whose encoding you cannot control.

Django does not decode the data of file uploads, because that data is normally treated as collections of bytes, rather than strings. Any automatic decoding there would alter the meaning of the stream of bytes.

6.23 `django.urls` utility functions

1.10 : In older versions, these functions are located in `django.core.urlresolvers`. Importing from the old location will continue to work until Django 2.0.

6.23.1 `reverse()`

If you need to use something similar to the *`url`* template tag in your code, Django provides the following function:

`reverse`(*viewname*, *urlconf=None*, *args=None*, *kwargs=None*, *current_app=None*)

`viewname` can be a *URL pattern name* or the callable view object. For example, given the following `url`:

```
from news import views

url(r'^archive/$', views.archive, name='news-archive')
```

you can use any of the following to reverse the URL:

```
# using the named URL
reverse('news-archive')

# passing a callable object
# (This is discouraged because you can't reverse namespaced views this way.)
from news import views
reverse(views.archive)
```

If the URL accepts arguments, you may pass them in `args`. For example:

```
from django.urls import reverse

def myview(request):
    return HttpResponseRedirect(reverse('arch-summary', args=[1945]))
```

You can also pass `kwargs` instead of `args`. For example:

```
>>> reverse('admin:app_list', kwargs={'app_label': 'auth'})
'/admin/auth/'
```

`args` and `kwargs` cannot be passed to `reverse()` at the same time.

If no match can be made, `reverse()` raises a *`NoReverseMatch`* exception.

The `reverse()` function can reverse a large variety of regular expression patterns for URLs, but not every possible one. The main restriction at the moment is that the pattern cannot contain alternative choices using the vertical bar ("|") character. You can quite happily use such patterns for matching against incoming URLs and sending them off to views, but you cannot reverse such patterns.

The `current_app` argument allows you to provide a hint to the resolver indicating the application to which the currently executing view belongs. This `current_app` argument is used as a hint to resolve application namespaces into URLs on specific application instances, according to the *namespaced URL resolution strategy*.

The `urlconf` argument is the URLconf module containing the URL patterns to use for reversing. By default, the root URLconf for the current thread is used.

: The string returned by `reverse()` is already *urlquoted*. For example:

```
>>> reverse('cities', args=['Orléans'])
'.../Orl%C3%A9ans/'
```

Applying further encoding (such as *`urlquote()`* or `urllib.quote`) to the output of `reverse()` may produce undesirable results.

6.23.2 `reverse_lazy()`

A lazily evaluated version of *reverse()*.

`reverse_lazy`(*viewname*, *urlconf=None*, *args=None*, *kwargs=None*, *current_app=None*)

It is useful for when you need to use a URL reversal before your project's URLConf is loaded. Some common cases where this function is necessary are:

- providing a reversed URL as the `url` attribute of a generic class-based view.
- providing a reversed URL to a decorator (such as the `login_url` argument for the *`django.contrib.auth.decorators.permission_required()`* decorator).
- providing a reversed URL as a default value for a parameter in a function's signature.

6.23.3 `resolve()`

The `resolve()` function can be used for resolving URL paths to the corresponding view functions. It has the following signature:

`resolve`(*path*, *urlconf=None*)

`path` is the URL path you want to resolve. As with *`reverse()`*, you don't need to worry about the `urlconf` parameter. The function returns a *`ResolverMatch`* object that allows you to access various metadata about the resolved URL.

If the URL does not resolve, the function raises a *`Resolver404`* exception (a subclass of *`Http404`*) .

class `ResolverMatch`

`func`
The view function that would be used to serve the URL

`args`
The arguments that would be passed to the view function, as parsed from the URL.

kwargs
The keyword arguments that would be passed to the view function, as parsed from the URL.

url_name
The name of the URL pattern that matches the URL.

app_name
The application namespace for the URL pattern that matches the URL.

app_names
The list of individual namespace components in the full application namespace for the URL pattern that matches the URL. For example, if the `app_name` is `'foo:bar'`, then `app_names` will be `['foo', 'bar']`.

namespace
The instance namespace for the URL pattern that matches the URL.

namespaces
The list of individual namespace components in the full instance namespace for the URL pattern that matches the URL. i.e., if the namespace is `foo:bar`, then namespaces will be `['foo', 'bar']`.

view_name
The name of the view that matches the URL, including the namespace if there is one.

A *ResolverMatch* object can then be interrogated to provide information about the URL pattern that matches a URL:

```
# Resolve a URL
match = resolve('/some/path/')
# Print the URL pattern that matches the URL
print(match.url_name)
```

A *ResolverMatch* object can also be assigned to a triple:

```
func, args, kwargs = resolve('/some/path/')
```

One possible use of *resolve()* would be to test whether a view would raise a `Http404` error before redirecting to it:

```
from django.urls import resolve
from django.http import HttpResponseRedirect, Http404
from django.utils.six.moves.urllib.parse import urlparse

def myview(request):
    next = request.META.get('HTTP_REFERER', None) or '/'
    response = HttpResponseRedirect(next)

    # modify the request and response as required, e.g. change locale
    # and set corresponding locale cookie

    view, args, kwargs = resolve(urlparse(next)[2])
    kwargs['request'] = request
    try:
        view(*args, **kwargs)
    except Http404:
        return HttpResponseRedirect('/')
    return response
```

6.23.4 get_script_prefix()

get_script_prefix()

Normally, you should always use *reverse()* to define URLs within your application. However, if your application constructs part of the URL hierarchy itself, you may occasionally need to generate URLs. In that case, you need to be able to find the base URL of the Django project within its Web server (normally, *reverse()* takes care of this for you). In that case, you can call `get_script_prefix()`, which will return the script prefix portion of the URL for your Django project. If your Django project is at the root of its web server, this is always `"/"`.

6.24 django.conf.urls utility functions

6.24.1 static()

`static.`**static**(*prefix*, *view=django.views.static.serve*, ***kwargs*)

Helper function to return a URL pattern for serving files in debug mode:

```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    # ... the rest of your URLconf goes here ...
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

6.24.2 url()

url(*regex*, *view*, *kwargs=None*, *name=None*)

`urlpatterns` should be a list of `url()` instances. For example:

```
from django.conf.urls import include, url

urlpatterns = [
    url(r'^index/$', index_view, name='main-view'),
    url(r'^weblog/', include('blog.urls')),
    ...
]
```

The `view` parameter is a view function or the result of *as_view()* for class-based views. It can also be an *include()*.

The `kwargs` parameter allows you to pass additional arguments to the view function or method. See *Passing extra options to view functions* for an example.

See *Naming URL patterns* for why the `name` parameter is useful.

6.24.3 include()

include(*module*, *namespace=None*, *app_name=None*)
include(*pattern_list*)
include((*pattern_list*, *app_namespace*), *namespace=None*)

include(*(pattern_list, app_namespace, instance_namespace)*)

A function that takes a full Python import path to another URLconf module that should be "included" in this place. Optionally, the *application namespace* and *instance namespace* where the entries will be included into can also be specified.

Usually, the application namespace should be specified by the included module. If an application namespace is set, the `namespace` argument can be used to set a different instance namespace.

`include()` also accepts as an argument either an iterable that returns URL patterns, a 2-tuple containing such iterable plus the names of the application namespaces, or a 3-tuple containing the iterable and the names of both the application and instance namespace.

- **module** – URLconf module (or module name)
- **namespace** (*string*) – Instance namespace for the URL entries being included
- **app_name** (*string*) – Application namespace for the URL entries being included
- **pattern_list** – Iterable of *django.conf.urls.url()* instances
- **app_namespace** (*string*) – Application namespace for the URL entries being included
- **instance_namespace** (*string*) – Instance namespace for the URL entries being included

See *Including other URLconfs* and *URL namespaces and included URLconfs*.

1.9 : Support for the `app_name` argument is deprecated and will be removed in Django 2.0. Specify the `app_name` as explained in *URL namespaces and included URLconfs* instead.

Support for passing a 3-tuple is also deprecated and will be removed in Django 2.0. Pass a 2-tuple containing the pattern list and application namespace, and use the `namespace` argument instead.

Lastly, support for an instance namespace without an application namespace has been deprecated and will be removed in Django 2.0. Specify the application namespace or remove the instance namespace.

6.24.4 handler400

handler400

A callable, or a string representing the full Python import path to the view that should be called if the HTTP client has sent a request that caused an error condition and a response with a status code of 400.

By default, this is `'django.views.defaults.bad_request'`. If you implement a custom view, be sure it returns an *HttpResponseBadRequest*.

See the documentation about *the 400 (bad request) view* for more information.

6.24.5 handler403

handler403

A callable, or a string representing the full Python import path to the view that should be called if the user doesn't have the permissions required to access a resource.

By default, this is `'django.views.defaults.permission_denied'`. If you implement a custom view, be sure it returns an *HttpResponseForbidden*.

See the documentation about *the 403 (HTTP Forbidden) view* for more information.

6.24.6 handler404

handler404

A callable, or a string representing the full Python import path to the view that should be called if none of the URL patterns match.

By default, this is `'django.views.defaults.page_not_found'`. If you implement a custom view, be sure it returns an *HttpResponseNotFound*.

See the documentation about *the 404 (HTTP Not Found) view* for more information.

6.24.7 handler500

handler500

A callable, or a string representing the full Python import path to the view that should be called in case of server errors. Server errors happen when you have runtime errors in view code.

By default, this is `'django.views.defaults.server_error'`. If you implement a custom view, be sure it returns an *HttpResponseServerError*.

See the documentation about *the 500 (HTTP Internal Server Error) view* for more information.

6.25 Django Utils

This document covers all stable modules in `django.utils`. Most of the modules in `django.utils` are designed for internal use and only the following parts can be considered stable and thus backwards compatible as per the *internal release deprecation policy*.

6.25.1 django.utils.cache

This module contains helper functions for controlling caching. It does so by managing the `Vary` header of responses. It includes functions to patch the header of response objects directly and decorators that change functions to do that header-patching themselves.

For information on the `Vary` header, see **RFC 7231#section-7.1.4**.

Essentially, the `Vary` HTTP header defines which headers a cache should take into account when building its cache key. Requests with the same path but different header content for headers named in `Vary` need to get different cache keys to prevent delivery of wrong content.

For example, internationalization middleware would need to distinguish caches by the `Accept-language` header.

patch_cache_control(*response*, ***kwargs*)

This function patches the `Cache-Control` header by adding all keyword arguments to it. The transformation is as follows:

- All keyword parameter names are turned to lowercase, and underscores are converted to hyphens.
- If the value of a parameter is `True` (exactly `True`, not just a true value), only the parameter name is added to the header.
- All other parameters are added with their value, after applying `str()` to it.

get_max_age(*response*)

Returns the max-age from the response Cache-Control header as an integer (or `None` if it wasn't found or wasn't an integer).

patch_response_headers(*response*, *cache_timeout=None*)
Adds some useful headers to the given `HttpResponse` object:

- `ETag`
- `Last-Modified`
- `Expires`
- `Cache-Control`

Each header is only added if it isn't already set.

`cache_timeout` is in seconds. The *CACHE_MIDDLEWARE_SECONDS* setting is used by default.

add_never_cache_headers(*response*)
Adds a `Cache-Control: max-age=0, no-cache, no-store, must-revalidate` header to a response to indicate that a page should never be cached.

In older versions, `Cache-Control: max-age=0` was sent. This didn't reliably prevent caching in all browsers.

patch_vary_headers(*response*, *newheaders*)
Adds (or updates) the `Vary` header in the given `HttpResponse` object. `newheaders` is a list of header names that should be in `Vary`. Existing headers in `Vary` aren't removed.

get_cache_key(*request*, *key_prefix=None*)
Returns a cache key based on the request path. It can be used in the request phase because it pulls the list of headers to take into account from the global path registry and uses those to build a cache key to check against.

If there is no headerlist stored, the page needs to be rebuilt, so this function returns `None`.

learn_cache_key(*request*, *response*, *cache_timeout=None*, *key_prefix=None*)
Learns what headers to take into account for some request path from the response object. It stores those headers in a global path registry so that later access to that path will know what headers to take into account without building the response object itself. The headers are named in the `Vary` header of the response, but we want to prevent response generation.

The list of headers to use for cache key generation is stored in the same cache as the pages themselves. If the cache ages some data out of the cache, this just means that we have to build the response once to get at the Vary header and so at the list of headers to use for the cache key.

6.25.2 django.utils.dateparse

The functions defined in this module share the following properties:

- They raise `ValueError` if their input is well formatted but isn't a valid date or time.
- They return `None` if it isn't well formatted at all.
- They accept up to picosecond resolution in input, but they truncate it to microseconds, since that's what Python supports.

parse_date(*value*)
Parses a string and returns a `datetime.date`.

parse_time(*value*)
Parses a string and returns a `datetime.time`.

UTC offsets aren't supported; if `value` describes one, the result is `None`.

parse_datetime(*value*)
Parses a string and returns a `datetime.datetime`.

UTC offsets are supported; if `value` describes one, the result's `tzinfo` attribute is a *`FixedOffset`* instance.

parse_duration(*value*)
Parses a string and returns a `datetime.timedelta`.

Expects data in the format `"DD HH:MM:SS.uuuuuu"` or as specified by ISO 8601 (e.g. `P4DT1H15M20S` which is equivalent to `4 1:15:20`).

6.25.3 django.utils.decorators

method_decorator(*decorator*, *name=''*)
Converts a function decorator into a method decorator. It can be used to decorate methods or classes; in the latter case, `name` is the name of the method to be decorated and is required.

`decorator` may also be a list or tuple of functions. They are wrapped in reverse order so that the call order is the order in which the functions appear in the list/tuple.

See *decorating class based views* for example usage.

The ability to decorate classes, the `name` parameter, and the ability for `decorator` to accept a list/tuple of decorator functions were added.

decorator_from_middleware(*middleware_class*)
Given a middleware class, returns a view decorator. This lets you use middleware functionality on a per-view basis. The middleware is created with no params passed.

It assumes middleware that's compatible with the old style of Django 1.9 and earlier (having methods like `process_request()`, `process_exception()`, and `process_response()`).

decorator_from_middleware_with_args(*middleware_class*)
Like `decorator_from_middleware`, but returns a function that accepts the arguments to be passed to the middleware_class. For example, the *`cache_page()`* decorator is created from the `CacheMiddleware` like this:

```
cache_page = decorator_from_middleware_with_args(CacheMiddleware)

@cache_page(3600)
def my_view(request):
    pass
```

6.25.4 django.utils.encoding

python_2_unicode_compatible()
A decorator that defines `__unicode__` and `__str__` methods under Python 2. Under Python 3 it does nothing.

To support Python 2 and 3 with a single code base, define a `__str__` method returning text and apply this decorator to the class.

smart_text(*s*, *encoding='utf-8'*, *strings_only=False*, *errors='strict'*)
Returns a text object representing `s` – `unicode` on Python 2 and `str` on Python 3. Treats bytestrings using the `encoding` codec.

If `strings_only` is `True`, don't convert (some) non-string-like objects.

smart_unicode(*s*, *encoding='utf-8'*, *strings_only=False*, *errors='strict'*)
Historical name of *smart_text()*. Only available under Python 2.

is_protected_type(*obj*)
Determine if the object instance is of a protected type.

Objects of protected types are preserved as-is when passed to `force_text(strings_only=True)`.

force_text(*s*, *encoding='utf-8'*, *strings_only=False*, *errors='strict'*)
Similar to `smart_text`, except that lazy instances are resolved to strings, rather than kept as lazy objects.

If `strings_only` is `True`, don't convert (some) non-string-like objects.

force_unicode(*s*, *encoding='utf-8'*, *strings_only=False*, *errors='strict'*)
Historical name of *force_text()*. Only available under Python 2.

smart_bytes(*s*, *encoding='utf-8'*, *strings_only=False*, *errors='strict'*)
Returns a bytestring version of `s`, encoded as specified in `encoding`.

If `strings_only` is `True`, don't convert (some) non-string-like objects.

force_bytes(*s*, *encoding='utf-8'*, *strings_only=False*, *errors='strict'*)
Similar to `smart_bytes`, except that lazy instances are resolved to bytestrings, rather than kept as lazy objects.

If `strings_only` is `True`, don't convert (some) non-string-like objects.

smart_str(*s*, *encoding='utf-8'*, *strings_only=False*, *errors='strict'*)
Alias of *smart_bytes()* on Python 2 and *smart_text()* on Python 3. This function returns a `str` or a lazy string.

For instance, this is suitable for writing to `sys.stdout` on Python 2 and 3.

force_str(*s*, *encoding='utf-8'*, *strings_only=False*, *errors='strict'*)
Alias of *force_bytes()* on Python 2 and *force_text()* on Python 3. This function always returns a `str`.

iri_to_uri(*iri*)
Convert an Internationalized Resource Identifier (IRI) portion to a URI portion that is suitable for inclusion in a URL.

This is the algorithm from section 3.1 of **RFC 3987#section-3.1**. However, since we are assuming input is either UTF-8 or unicode already, we can simplify things a little from the full method.

Takes an IRI in UTF-8 bytes and returns ASCII bytes containing the encoded result.

uri_to_iri(*uri*)
Converts a Uniform Resource Identifier into an Internationalized Resource Identifier.

This is an algorithm from section 3.2 of **RFC 3987#section-3.2**.

Takes a URI in ASCII bytes and returns a unicode string containing the encoded result.

filepath_to_uri(*path*)
Convert a file system path to a URI portion that is suitable for inclusion in a URL. The path is assumed to be either UTF-8 or unicode.

This method will encode certain characters that would normally be recognized as special characters for URIs. Note that this method does not encode the ' character, as it is a valid character within URIs. See `encodeURIComponent()` JavaScript function for more details.

Returns an ASCII string containing the encoded result.

escape_uri_path(*path*)
Escapes the unsafe characters from the path portion of a Uniform Resource Identifier (URI).

6.25.5 django.utils.feedgenerator

Sample usage:

```
>>> from django.utils import feedgenerator
>>> feed = feedgenerator.Rss201rev2Feed(
...     title="Poynter E-Media Tidbits",
...     link="http://www.poynter.org/column.asp?id=31",
...     description="A group Weblog by the sharpest minds in online media/journalism/publishing.",
...     language="en",
... )
>>> feed.add_item(
...     title="Hello",
...     link="http://www.holovaty.com/test/",
...     description="Testing.",
... )
>>> with open('test.rss', 'w') as fp:
...     feed.write(fp, 'utf-8')
```

For simplifying the selection of a generator use `feedgenerator.DefaultFeed` which is currently `Rss201rev2Feed`

For definitions of the different versions of RSS, see: https://web.archive.org/web/20110718035220/http://diveintomark.org/archives/2004/06/02/atom-and-rss

get_tag_uri(*url*, *date*)
Creates a TagURI.

See https://web.archive.org/web/20110514113830/http://diveintomark.org/archives/2004/05/28/howto-atom-id

SyndicationFeed

class SyndicationFeed
Base class for all syndication feeds. Subclasses should provide write().

__init__(*title*, *link*, *description*, *language=None*, *author_email=None*, *author_name=None*, *author_link=None*, *subtitle=None*, *categories=None*, *feed_url=None*, *feed_copyright=None*, *feed_guid=None*, *ttl=None*, ***kwargs*)
Initialize the feed with the given dictionary of metadata, which applies to the entire feed.

Any extra keyword arguments you pass to `__init__` will be stored in `self.feed`.

All parameters should be Unicode objects, except `categories`, which should be a sequence of Unicode objects.

add_item(*title*, *link*, *description*, *author_email=None*, *author_name=None*, *author_link=None*, *pubdate=None*, *comments=None*, *unique_id=None*, *enclosure=None*, *categories=()*, *item_copyright=None*, *ttl=None*, *updateddate=None*, *enclosures=None*, ***kwargs*)
Adds an item to the feed. All args are expected to be Python `unicode` objects except `pubdate` and `updateddate`, which are `datetime.datetime` objects, `enclosure`, which is an `Enclosure` instance, and `enclosures`, which is a list of `Enclosure` instances.

1.9 : The `enclosure` keyword argument is deprecated in favor of the new `enclosures` keyword argument which accepts a list of `Enclosure` objects.

num_items()

root_attributes()
Return extra attributes to place on the root (i.e. feed/channel) element. Called from `write()`.

add_root_elements(*handler*)
Add elements in the root (i.e. feed/channel) element. Called from `write()`.

item_attributes(*item*)
Return extra attributes to place on each item (i.e. item/entry) element.

add_item_elements(*handler*, *item*)
Add elements on each item (i.e. item/entry) element.

write(*outfile*, *encoding*)
Outputs the feed in the given encoding to `outfile`, which is a file-like object. Subclasses should override this.

writeString(*encoding*)
Returns the feed in the given encoding as a string.

latest_post_date()
Returns the latest `pubdate` or `updateddate` for all items in the feed. If no items have either of these attributes this returns the current date/time.

Enclosure

class Enclosure
Represents an RSS enclosure

RssFeed

class RssFeed(*SyndicationFeed*)

Rss201rev2Feed

class Rss201rev2Feed(*RssFeed*)
Spec: https://cyber.law.harvard.edu/rss/rss.html

RssUserland091Feed

class RssUserland091Feed(*RssFeed*)
Spec: http://backend.userland.com/rss091

Atom1Feed

class Atom1Feed(*SyndicationFeed*)
Spec: https://tools.ietf.org/html/rfc4287

6.25.6 django.utils.functional

class cached_property(*object*, *name*)
The `@cached_property` decorator caches the result of a method with a single `self` argument as a property. The cached result will persist as long as the instance does, so if the instance is passed around and the function subsequently invoked, the cached result will be returned.

Consider a typical case, where a view might need to call a model's method to perform some computation, before placing the model instance into the context, where the template might invoke the method once more:

```
    # the model
    class Person(models.Model):

        def friends(self):
            # expensive computation
            ...
            return friends

    # in the view:
    if person.friends():
        ...
```

And in the template you would have:

```
    {% for friend in person.friends %}
```

Here, `friends()` will be called twice. Since the instance `person` in the view and the template are the same, `@cached_property` can avoid that:

```
    from django.utils.functional import cached_property

    @cached_property
    def friends(self):
        # expensive computation
        ...
        return friends
```

Note that as the method is now a property, in Python code it will need to be invoked appropriately:

```
    # in the view:
    if person.friends:
        ...
```

The cached value can be treated like an ordinary attribute of the instance:

```
    # clear it, requiring re-computation next time it's called
    del person.friends # or delattr(person, "friends")

    # set a value manually, that will persist on the instance until cleared
    person.friends = ["Huckleberry Finn", "Tom Sawyer"]
```

As well as offering potential performance advantages, `@cached_property` can ensure that an attribute's value does not change unexpectedly over the life of an instance. This could occur with a method whose computation is based on `datetime.now()`, or simply if a change were saved to the database by some other process in the brief interval between subsequent invocations of a method on the same instance.

You can use the `name` argument to make cached properties of other methods. For example, if you had an expensive `get_friends()` method and wanted to allow calling it without retrieving the cached value, you could write:

```
    friends = cached_property(get_friends, name='friends')
```

While `person.get_friends()` will recompute the friends on each call, the value of the cached property will persist until you delete it as described above:

```
    x = person.friends          # calls first time
    y = person.get_friends()    # calls again
    z = person.friends          # does not call
    x is z                      # is True
```

allow_lazy (*func*, **resultclasses*)
1.10 .

Works like *keep_lazy()* except that it can't be used as a decorator.

keep_lazy (*func*, **resultclasses*)
Django offers many utility functions (particularly in `django.utils`) that take a string as their first argument and do something to that string. These functions are used by template filters as well as directly in other code.

If you write your own similar functions and deal with translations, you'll face the problem of what to do when the first argument is a lazy translation object. You don't want to convert it to a string immediately, because you might be using this function outside of a view (and hence the current thread's locale setting will not be correct).

For cases like this, use the `django.utils.functional.keep_lazy()` decorator. It modifies the function so that *if* it's called with a lazy translation as one of its arguments, the function evaluation is delayed until it needs to be converted to a string.

For example:

```
from django.utils import six
from django.utils.functional import keep_lazy, keep_lazy_text

def fancy_utility_function(s, ...):
    # Do some conversion on string 's'
    ...
fancy_utility_function = keep_lazy(six.text_type)(fancy_utility_function)

# Or more succinctly:
@keep_lazy(six.text_type)
def fancy_utility_function(s, ...):
    ...
```

The `keep_lazy()` decorator takes a number of extra arguments (`*args`) specifying the type(s) that the original function can return. A common use case is to have functions that return text. For these, you can just pass the `six.text_type` type to `keep_lazy` (or even simpler, use the *keep_lazy_text()* decorator described in the next section).

Using this decorator means you can write your function and assume that the input is a proper string, then add support for lazy translation objects at the end.

keep_lazy_text (*func*)
A shortcut for `keep_lazy(six.text_type)(func)`.

If you have a function that returns text and you want to be able to take lazy arguments while delaying their evaluation, simply use this decorator:

```
from django.utils import six
from django.utils.functional import keep_lazy, keep_lazy_text

# Our previous example was:
@keep_lazy(six.text_type)
def fancy_utility_function(s, ...):
    ...

# Which can be rewritten as:
@keep_lazy_text
def fancy_utility_function(s, ...):
    ...
```

6.25.7 django.utils.html

Usually you should build up HTML using Django's templates to make use of its autoescape mechanism, using the utilities in *django.utils.safestring* where appropriate. This module provides some additional low level utilities for escaping HTML.

escape(*text*)
Returns the given text with ampersands, quotes and angle brackets encoded for use in HTML. The input is first passed through *force_text()* and the output has *mark_safe()* applied.

conditional_escape(*text*)
Similar to `escape()`, except that it doesn't operate on pre-escaped strings, so it will not double escape.

format_html(*format_string*, **args*, ***kwargs*)
This is similar to `str.format()`, except that it is appropriate for building up HTML fragments. All args and kwargs are passed through *conditional_escape()* before being passed to `str.format()`.

For the case of building up small HTML fragments, this function is to be preferred over string interpolation using `%` or `str.format()` directly, because it applies escaping to all arguments - just like the template system applies escaping by default.

So, instead of writing:

```
mark_safe("%s <b>%s</b> %s" % (
    some_html,
    escape(some_text),
    escape(some_other_text),
))
```

You should instead use:

```
format_html("{} <b>{}</b> {}",
    mark_safe(some_html),
    some_text,
    some_other_text,
)
```

This has the advantage that you don't need to apply *escape()* to each argument and risk a bug and an XSS vulnerability if you forget one.

Note that although this function uses `str.format()` to do the interpolation, some of the formatting options provided by `str.format()` (e.g. number formatting) will not work, since all arguments are passed through *conditional_escape()* which (ultimately) calls *force_text()* on the values.

format_html_join(*sep*, *format_string*, *args_generator*)
A wrapper of *format_html()*, for the common case of a group of arguments that need to be formatted using the same format string, and then joined using `sep`. `sep` is also passed through *conditional_escape()*.

`args_generator` should be an iterator that returns the sequence of `args` that will be passed to *format_html()*. For example:

```
format_html_join(
    '\n', "<li>{} {}</li>",
    ((u.first_name, u.last_name) for u in users)
)
```

strip_tags(*value*)
Tries to remove anything that looks like an HTML tag from the string, that is anything contained within <>.

Absolutely NO guarantee is provided about the resulting string being HTML safe. So NEVER mark safe the result of a `strip_tag` call without escaping it first, for example with *escape()*.

For example:

```
strip_tags(value)
```

If `value` is `"<b>Joel</b> <button>is</button> a <span>slug</span>"` the return value will be `"Joel is a slug"`.

If you are looking for a more robust solution, take a look at the bleach Python library.

html_safe()
The `__html__()` method on a class helps non-Django templates detect classes whose output doesn't require HTML escaping.

This decorator defines the `__html__()` method on the decorated class by wrapping the `__unicode__()` (Python 2) or `__str__()` (Python 3) in *mark_safe()*. Ensure the `__unicode__()` or `__str__()` method does indeed return text that doesn't require HTML escaping.

6.25.8 django.utils.http

urlquote(*url*, *safe='/'*)
A version of Python's `urllib.quote()` function that can operate on unicode strings. The url is first UTF-8 encoded before quoting. The returned string can safely be used as part of an argument to a subsequent `iri_to_uri()` call without double-quoting occurring. Employs lazy execution.

urlquote_plus(*url*, *safe=''*)
A version of Python's urllib.quote_plus() function that can operate on unicode strings. The url is first UTF-8 encoded before quoting. The returned string can safely be used as part of an argument to a subsequent `iri_to_uri()` call without double-quoting occurring. Employs lazy execution.

urlencode(*query*, *doseq=0*)
A version of Python's urllib.urlencode() function that can operate on unicode strings. The parameters are first cast to UTF-8 encoded strings and then encoded as per normal.

cookie_date(*epoch_seconds=None*)
Formats the time to ensure compatibility with Netscape's cookie standard.

Accepts a floating point number expressed in seconds since the epoch in UTC–such as that outputted by `time.time()`. If set to `None`, defaults to the current time.

Outputs a string in the format `Wdy, DD-Mon-YYYY HH:MM:SS GMT`.

http_date(*epoch_seconds=None*)
Formats the time to match the **RFC 1123** date format as specified by HTTP **RFC 7231#section-7.1.1.1**.

Accepts a floating point number expressed in seconds since the epoch in UTC–such as that outputted by `time.time()`. If set to `None`, defaults to the current time.

Outputs a string in the format `Wdy, DD Mon YYYY HH:MM:SS GMT`.

base36_to_int(*s*)
Converts a base 36 string to an integer. On Python 2 the output is guaranteed to be an `int` and not a `long`.

int_to_base36(*i*)
Converts a positive integer to a base 36 string. On Python 2 `i` must be smaller than sys.maxint.

urlsafe_base64_encode(*s*)
Encodes a bytestring in base64 for use in URLs, stripping any trailing equal signs.

urlsafe_base64_decode(*s*)
Decodes a base64 encoded string, adding back any trailing equal signs that might have been stripped.

6.25.9 django.utils.module_loading

Functions for working with Python modules.

import_string(*dotted_path*)

Imports a dotted module path and returns the attribute/class designated by the last name in the path. Raises `ImportError` if the import failed. For example:

```
from django.utils.module_loading import import_string
ValidationError = import_string('django.core.exceptions.ValidationError')
```

is equivalent to:

```
from django.core.exceptions import ValidationError
```

6.25.10 django.utils.safestring

Functions and classes for working with "safe strings": strings that can be displayed safely without further escaping in HTML. Marking something as a "safe string" means that the producer of the string has already turned characters that should not be interpreted by the HTML engine (e.g. '<') into the appropriate entities.

class SafeBytes

A `bytes` subclass that has been specifically marked as "safe" (requires no further escaping) for HTML output purposes.

class SafeString

A `str` subclass that has been specifically marked as "safe" (requires no further escaping) for HTML output purposes. This is *SafeBytes* on Python 2 and *SafeText* on Python 3.

class SafeText

A `str` (in Python 3) or `unicode` (in Python 2) subclass that has been specifically marked as "safe" for HTML output purposes.

class SafeUnicode

Historical name of *SafeText*. Only available under Python 2.

mark_safe(*s*)

Explicitly mark a string as safe for (HTML) output purposes. The returned object can be used everywhere a string or unicode object is appropriate.

Can be called multiple times on a single string.

For building up fragments of HTML, you should normally be using *django.utils.html.format_html()* instead.

String marked safe will become unsafe again if modified. For example:

```
>>> mystr = '<b>Hello World</b>   '
>>> mystr = mark_safe(mystr)
>>> type(mystr)
<class 'django.utils.safestring.SafeBytes'>

>>> mystr = mystr.strip()  # removing whitespace
>>> type(mystr)
<type 'str'>
```

mark_for_escaping(*s*)

1.10 .

Explicitly mark a string as requiring HTML escaping upon output. Has no effect on `SafeData` subclasses.

Can be called multiple times on a single string (the resulting escaping is only applied once).

6.25.11 django.utils.text

slugify(*allow_unicode=False*)
Converts to ASCII if `allow_unicode` is `False` (default). Converts spaces to hyphens. Removes characters that aren't alphanumerics, underscores, or hyphens. Converts to lowercase. Also strips leading and trailing whitespace.

For example:

```
slugify(value)
```

If `value` is `"Joel is a slug"`, the output will be `"joel-is-a-slug"`.

You can set the `allow_unicode` parameter to `True`, if you want to allow Unicode characters:

```
slugify(value, allow_unicode=True)
```

If `value` is `" World"`, the output will be `"-world"`.

The `allow_unicode` parameter was added.

6.25.12 django.utils.timezone

utc
`tzinfo` instance that represents UTC.

class FixedOffset(*offset=None*, *name=None*)
A `tzinfo` subclass modeling a fixed offset from UTC. `offset` is an integer number of minutes east of UTC.

get_fixed_timezone(*offset*)
Returns a `tzinfo` instance that represents a time zone with a fixed offset from UTC.

`offset` is a `datetime.timedelta` or an integer number of minutes. Use positive values for time zones east of UTC and negative values for west of UTC.

get_default_timezone()
Returns a `tzinfo` instance that represents the *default time zone*.

get_default_timezone_name()
Returns the name of the *default time zone*.

get_current_timezone()
Returns a `tzinfo` instance that represents the *current time zone*.

get_current_timezone_name()
Returns the name of the *current time zone*.

activate(*timezone*)
Sets the *current time zone*. The `timezone` argument must be an instance of a `tzinfo` subclass or, if pytz is available, a time zone name.

deactivate()
Unsets the *current time zone*.

override(*timezone*)
This is a Python context manager that sets the *current time zone* on entry with *`activate()`*, and restores the previously active time zone on exit. If the `timezone` argument is `None`, the *current time zone* is unset on entry with *`deactivate()`* instead.

`override` is also usable as a function decorator.

localtime(*value*, *timezone=None*)
Converts an aware `datetime` to a different time zone, by default the *current time zone*.

This function doesn't work on naive datetimes; use *`make_aware()`* instead.

now()
Returns a `datetime` that represents the current point in time. Exactly what's returned depends on the value of *`USE_TZ`*:

- If *`USE_TZ`* is `False`, this will be a *naive* datetime (i.e. a datetime without an associated timezone) that represents the current time in the system's local timezone.
- If *`USE_TZ`* is `True`, this will be an *aware* datetime representing the current time in UTC. Note that *`now()`* will always return times in UTC regardless of the value of *`TIME_ZONE`*; you can use *`localtime()`* to convert to a time in the current time zone.

is_aware(*value*)
Returns `True` if `value` is aware, `False` if it is naive. This function assumes that `value` is a `datetime`.

is_naive(*value*)
Returns `True` if `value` is naive, `False` if it is aware. This function assumes that `value` is a `datetime`.

make_aware(*value*, *timezone=None*, *is_dst=None*)
Returns an aware `datetime` that represents the same point in time as `value` in `timezone`, `value` being a naive `datetime`. If `timezone` is set to `None`, it defaults to the *current time zone*.

When pytz is installed, the exception `pytz.AmbiguousTimeError` will be raised if you try to make `value` aware during a DST transition where the same time occurs twice (when reverting from DST). Setting `is_dst` to `True` or `False` will avoid the exception by choosing if the time is pre-transition or post-transition respectively.

When pytz is installed, the exception `pytz.NonExistentTimeError` will be raised if you try to make `value` aware during a DST transition such that the time never occurred (when entering into DST). Setting `is_dst` to `True` or `False` will avoid the exception by moving the hour backwards or forwards by 1 respectively. For example, `is_dst=True` would change a non-existent time of 2:30 to 1:30 and `is_dst=False` would change the time to 3:30.

`is_dst` has no effect when `pytz` is not installed.

The `is_dst` argument was added.

make_naive(*value*, *timezone=None*)
Returns an naive `datetime` that represents in `timezone` the same point in time as `value`, `value` being an aware `datetime`. If `timezone` is set to `None`, it defaults to the *current time zone*.

6.25.13 django.utils.translation

For a complete discussion on the usage of the following see the translation documentation.

gettext(*message*)
Translates `message` and returns it in a UTF-8 bytestring

ugettext(*message*)
Translates `message` and returns it in a unicode string

pgettext(*context*, *message*)
Translates `message` given the `context` and returns it in a unicode string.

For more information, see *Contextual markers*.

gettext_lazy(*message*)

ugettext_lazy(*message*)

pgettext_lazy(*context*, *message*)
Same as the non-lazy versions above, but using lazy execution.

See *lazy translations documentation*.

gettext_noop(*message*)

ugettext_noop(*message*)
Marks strings for translation but doesn't translate them now. This can be used to store strings in global variables that should stay in the base language (because they might be used externally) and will be translated later.

ngettext(*singular*, *plural*, *number*)
Translates `singular` and `plural` and returns the appropriate string based on `number` in a UTF-8 bytestring.

ungettext(*singular*, *plural*, *number*)
Translates `singular` and `plural` and returns the appropriate string based on `number` in a unicode string.

npgettext(*context*, *singular*, *plural*, *number*)
Translates `singular` and `plural` and returns the appropriate string based on `number` and the `context` in a unicode string.

ngettext_lazy(*singular*, *plural*, *number*)

ungettext_lazy(*singular*, *plural*, *number*)

npgettext_lazy(*context*, *singular*, *plural*, *number*)
Same as the non-lazy versions above, but using lazy execution.

See *lazy translations documentation*.

string_concat(**strings*)
Lazy variant of string concatenation, needed for translations that are constructed from multiple parts.

activate(*language*)
Fetches the translation object for a given language and activates it as the current translation object for the current thread.

deactivate()
Deactivates the currently active translation object so that further _ calls will resolve against the default translation object, again.

deactivate_all()
Makes the active translation object a `NullTranslations()` instance. This is useful when we want delayed translations to appear as the original string for some reason.

override(*language*, *deactivate=False*)
A Python context manager that uses *`django.utils.translation.activate()`* to fetch the translation object for a given language, activates it as the translation object for the current thread and reactivates the previous active language on exit. Optionally, it can simply deactivate the temporary translation on exit with *`django.utils.translation.deactivate()`* if the `deactivate` argument is `True`. If you pass `None` as the language argument, a `NullTranslations()` instance is activated within the context.

`override` is also usable as a function decorator.

check_for_language(*lang_code*)
Checks whether there is a global language file for the given language code (e.g. 'fr', 'pt_BR'). This is used to decide whether a user-provided language is available.

get_language()
Returns the currently selected language code. Returns `None` if translations are temporarily deactivated (by *`deactivate_all()`* or when `None` is passed to *`override()`*).

get_language_bidi()
Returns selected language's BiDi layout:

- `False` = left-to-right layout
- `True` = right-to-left layout

get_language_from_request(*request*, *check_path=False*)
Analyzes the request to find what language the user wants the system to show. Only languages listed in settings.LANGUAGES are taken into account. If the user requests a sublanguage where we have a main language, we send out the main language.

If `check_path` is `True`, the function first checks the requested URL for whether its path begins with a language code listed in the *LANGUAGES* setting.

to_locale(*language*)
Turns a language name (en-us) into a locale name (en_US).

templatize(*src*)
Turns a Django template into something that is understood by `xgettext`. It does so by translating the Django translation tags into standard `gettext` function invocations.

LANGUAGE_SESSION_KEY
Session key under which the active language for the current session is stored.

6.26 Validators

6.26.1 Writing validators

A validator is a callable that takes a value and raises a *ValidationError* if it doesn't meet some criteria. Validators can be useful for re-using validation logic between different types of fields.

For example, here's a validator that only allows even numbers:

```
from django.core.exceptions import ValidationError
from django.utils.translation import ugettext_lazy as _

def validate_even(value):
    if value % 2 != 0:
        raise ValidationError(
            _('%(value)s is not an even number'),
            params={'value': value},
        )
```

You can add this to a model field via the field's *validators* argument:

```
from django.db import models

class MyModel(models.Model):
    even_field = models.IntegerField(validators=[validate_even])
```

Because values are converted to Python before validators are run, you can even use the same validator with forms:

```
from django import forms

class MyForm(forms.Form):
    even_field = forms.IntegerField(validators=[validate_even])
```

You can also use a class with a `__call__()` method for more complex or configurable validators. *RegexValidator*, for example, uses this technique. If a class-based validator is used in the *validators* model field option, you should make sure it is *serializable by the migration framework* by adding *deconstruct()* and `__eq__()` methods.

6.26.2 How validators are run

See the form validation for more information on how validators are run in forms, and *Validating objects* for how they're run in models. Note that validators will not be run automatically when you save a model, but if you are using a *ModelForm*, it will run your validators on any fields that are included in your form. See the ModelForm documentation for information on how model validation interacts with forms.

6.26.3 Built-in validators

The *django.core.validators* module contains a collection of callable validators for use with model and form fields. They're used internally but are available for use with your own fields, too. They can be used in addition to, or in lieu of custom `field.clean()` methods.

RegexValidator

class RegexValidator(*regex=None*, *message=None*, *code=None*, *inverse_match=None*, *flags=0*)

- **regex** – If not `None`, overrides *regex*. Can be a regular expression string or a pre-compiled regular expression.
- **message** – If not `None`, overrides *message*.
- **code** – If not `None`, overrides *code*.
- **inverse_match** – If not `None`, overrides *inverse_match*.
- **flags** – If not `None`, overrides *flags*. In that case, *regex* must be a regular expression string, or `TypeError` is raised.

regex
The regular expression pattern to search for the provided `value`, or a pre-compiled regular expression. By default, raises a *ValidationError* with *message* and *code* if a match is not found. That standard behavior can be reversed by setting *inverse_match* to `True`, in which case the *ValidationError* is raised when a match **is** found. By default, matches any string (including an empty string).

message
The error message used by *ValidationError* if validation fails. Defaults to `"Enter a valid value"`.

code
The error code used by *ValidationError* if validation fails. Defaults to `"invalid"`.

inverse_match
The match mode for *regex*. Defaults to `False`.

flags
The flags used when compiling the regular expression string *regex*. If *regex* is a pre-compiled regular expression, and *flags* is overridden, `TypeError` is raised. Defaults to `0`.

EmailValidator

class EmailValidator(*message=None*, *code=None*, *whitelist=None*)

- **message** – If not `None`, overrides *message*.
- **code** – If not `None`, overrides *code*.
- **whitelist** – If not `None`, overrides *whitelist*.

message
The error message used by *ValidationError* if validation fails. Defaults to `"Enter a valid email address"`.

code
The error code used by *ValidationError* if validation fails. Defaults to `"invalid"`.

whitelist
Whitelist of email domains to allow. By default, a regular expression (the `domain_regex` attribute) is used to validate whatever appears after the @ sign. However, if that string appears in the whitelist, this validation is bypassed. If not provided, the default whitelist is `['localhost']`. Other domains that don't contain a dot won't pass validation, so you'd need to whitelist them as necessary.

URLValidator

class URLValidator(*schemes=None*, *regex=None*, *message=None*, *code=None*)
A *RegexValidator* that ensures a value looks like a URL, and raises an error code of `'invalid'` if it doesn't.

Loopback addresses and reserved IP spaces are considered valid. Literal IPv6 addresses (**RFC 2732**) and unicode domains are both supported.

In addition to the optional arguments of its parent *RegexValidator* class, `URLValidator` accepts an extra optional attribute:

schemes
URL/URI scheme list to validate against. If not provided, the default list is `['http', 'https', 'ftp', 'ftps']`. As a reference, the IANA website provides a full list of valid URI schemes.

validate_email

validate_email
An *EmailValidator* instance without any customizations.

validate_slug

validate_slug
A *RegexValidator* instance that ensures a value consists of only letters, numbers, underscores or hyphens.

validate_unicode_slug

validate_unicode_slug
A *RegexValidator* instance that ensures a value consists of only Unicode letters, numbers, underscores, or hyphens.

validate_ipv4_address

validate_ipv4_address
A *RegexValidator* instance that ensures a value looks like an IPv4 address.

validate_ipv6_address

validate_ipv6_address
Uses `django.utils.ipv6` to check the validity of an IPv6 address.

validate_ipv46_address

validate_ipv46_address
Uses both `validate_ipv4_address` and `validate_ipv6_address` to ensure a value is either a valid IPv4 or IPv6 address.

validate_comma_separated_integer_list

validate_comma_separated_integer_list
A *RegexValidator* instance that ensures a value is a comma-separated list of integers.

int_list_validator

int_list_validator(*sep='*, *'*, *message=None*, *code='invalid'*, *allow_negative=False*)
Returns a *RegexValidator* instance that ensures a string consists of integers separated by `sep`. It allows negative integers when `allow_negative` is `True`.

The `allow_negative` parameter was added.

MaxValueValidator

class MaxValueValidator(*max_value*, *message=None*)
Raises a *ValidationError* with a code of `'max_value'` if `value` is greater than `max_value`.

MinValueValidator

class MinValueValidator(*min_value*, *message=None*)
Raises a *ValidationError* with a code of `'min_value'` if `value` is less than `min_value`.

MaxLengthValidator

class MaxLengthValidator(*max_length*, *message=None*)
Raises a *ValidationError* with a code of `'max_length'` if the length of `value` is greater than `max_length`.

MinLengthValidator

class MinLengthValidator(*min_length*, *message=None*)
Raises a *ValidationError* with a code of `'min_length'` if the length of `value` is less than `min_length`.

DecimalValidator

class DecimalValidator(*max_digits*, *decimal_places*)
Raises *ValidationError* with the following codes:

- `'max_digits'` if the number of digits is larger than `max_digits`.
- `'max_decimal_places'` if the number of decimals is larger than `decimal_places`.
- `'max_whole_digits'` if the number of whole digits is larger than the difference between `max_digits` and `decimal_places`.

6.27 Built-in Views

Several of Django's built-in views are documented in Writing views as well as elsewhere in the documentation.

6.27.1 Serving files in development

`static.`**serve**(*request*, *path*, *document_root*, *show_indexes=False*)

There may be files other than your project's static assets that, for convenience, you'd like to have Django serve for you in local development. The *serve()* view can be used to serve any directory you give it. (This view is **not** hardened for production use and should be used only as a development aid; you should serve these files in production using a real front-end web server).

The most likely example is user-uploaded content in *MEDIA_ROOT*. `django.contrib.staticfiles` is intended for static assets and has no built-in handling for user-uploaded files, but you can have Django serve your *MEDIA_ROOT* by appending something like this to your URLconf:

```
from django.conf import settings
from django.views.static import serve

# ... the rest of your URLconf goes here ...

if settings.DEBUG:
    urlpatterns += [
        url(r'^media/(?P<path>.*)$', serve, {
            'document_root': settings.MEDIA_ROOT,
        }),
    ]
```

Note, the snippet assumes your *MEDIA_URL* has a value of `'/media/'`. This will call the *serve()* view, passing in the path from the URLconf and the (required) `document_root` parameter.

Since it can become a bit cumbersome to define this URL pattern, Django ships with a small URL helper function *static()* that takes as parameters the prefix such as *MEDIA_URL* and a dotted path to a view, such as `'django.views.static.serve'`. Any other function parameter will be transparently passed to the view.

6.27.2 Error views

Django comes with a few views by default for handling HTTP errors. To override these with your own custom views, see *Customizing error views*.

The 404 (page not found) view

`defaults.`**`page_not_found`**(*request*, *exception*, *template_name='404.html'*)

When you raise *`Http404`* from within a view, Django loads a special view devoted to handling 404 errors. By default, it's the view *`django.views.defaults.page_not_found()`*, which either produces a very simple "Not Found" message or loads and renders the template `404.html` if you created it in your root template directory.

The default 404 view will pass two variables to the template: `request_path`, which is the URL that resulted in the error, and `exception`, which is a useful representation of the exception that triggered the view (e.g. containing any message passed to a specific `Http404` instance).

Three things to note about 404 views:

- The 404 view is also called if Django doesn't find a match after checking every regular expression in the URLconf.
- The 404 view is passed a *`RequestContext`* and will have access to variables supplied by your template context processors (e.g. `MEDIA_URL`).
- If *`DEBUG`* is set to `True` (in your settings module), then your 404 view will never be used, and your URLconf will be displayed instead, with some debug information.

The signature of `page_not_found()` changed. The function now accepts a second parameter, the exception that triggered the error. A useful representation of the exception is also passed in the template context.

Passing a nonexistent `template_name` will raise `TemplateDoesNotExist`.

The 500 (server error) view

`defaults.`**`server_error`**(*request*, *template_name='500.html'*)

Similarly, Django executes special-case behavior in the case of runtime errors in view code. If a view results in an exception, Django will, by default, call the view `django.views.defaults.server_error`, which either produces a very simple "Server Error" message or loads and renders the template `500.html` if you created it in your root template directory.

The default 500 view passes no variables to the `500.html` template and is rendered with an empty `Context` to lessen the chance of additional errors.

If *`DEBUG`* is set to `True` (in your settings module), then your 500 view will never be used, and the traceback will be displayed instead, with some debug information.

Passing a nonexistent `template_name` will raise `TemplateDoesNotExist`.

The 403 (HTTP Forbidden) view

`defaults.`**`permission_denied`**(*request*, *exception*, *template_name='403.html'*)

In the same vein as the 404 and 500 views, Django has a view to handle 403 Forbidden errors. If a view results in a 403 exception then Django will, by default, call the view `django.views.defaults.permission_denied`.

This view loads and renders the template `403.html` in your root template directory, or if this file does not exist, instead serves the text "403 Forbidden", as per **RFC 7231#section-6.5.3** (the HTTP 1.1 Specification). The template context contains `exception`, which is the unicode representation of the exception that triggered the view.

`django.views.defaults.permission_denied` is triggered by a *`PermissionDenied`* exception. To deny access in a view you can use code like this:

```
from django.core.exceptions import PermissionDenied

def edit(request, pk):
    if not request.user.is_staff:
        raise PermissionDenied
    # ...
```

The signature of `permission_denied()` changed in Django 1.9. The function now accepts a second parameter, the exception that triggered the error. The unicode representation of the exception is also passed in the template context.

Passing a nonexistent `template_name` will raise `TemplateDoesNotExist`.

The 400 (bad request) view

`defaults.`**`bad_request`**(*request*, *exception*, *template_name='400.html'*)

When a *`SuspiciousOperation`* is raised in Django, it may be handled by a component of Django (for example resetting the session data). If not specifically handled, Django will consider the current request a 'bad request' instead of a server error.

`django.views.defaults.bad_request`, is otherwise very similar to the `server_error` view, but returns with the status code 400 indicating that the error condition was the result of a client operation. By default, nothing related to the exception that triggered the view is passed to the template context, as the exception message might contain sensitive information like filesystem paths.

`bad_request` views are also only used when *`DEBUG`* is `False`.

The signature of `bad_request()` changed in Django 1.9. The function now accepts a second parameter, the exception that triggered the error.

Passing a nonexistent `template_name` will raise `TemplateDoesNotExist`.

CHAPTER 7

Meta-documentation and miscellany

Documentation that we can't find a more organized place for. Like that drawer in your kitchen with the scissors, batteries, duct tape, and other junk.

7.1 API stability

Django promises API stability and forwards-compatibility since version 1.0. In a nutshell, this means that code you develop against a version of Django will continue to work with future releases. You may need to make minor changes when upgrading the version of Django your project uses: see the "Backwards incompatible changes" section of the release note for the version or versions to which you are upgrading.

7.1.1 What "stable" means

In this context, stable means:

- All the public APIs (everything in this documentation) will not be moved or renamed without providing backwards-compatible aliases.
- If new features are added to these APIs – which is quite possible – they will not break or change the meaning of existing methods. In other words, "stable" does not (necessarily) mean "complete."
- If, for some reason, an API declared stable must be removed or replaced, it will be declared deprecated but will remain in the API for at least two feature releases. Warnings will be issued when the deprecated method is called.

  See *Official releases* for more details on how Django's version numbering scheme works, and how features will be deprecated.
- We'll only break backwards compatibility of these APIs if a bug or security hole makes it completely unavoidable.

7.1.2 Stable APIs

In general, everything covered in the documentation – with the exception of anything in the internals area is considered stable.

7.1.3 Exceptions

There are a few exceptions to this stability and backwards-compatibility promise.

Security fixes

If we become aware of a security problem – hopefully by someone following our *security reporting policy* – we'll do everything necessary to fix it. This might mean breaking backwards compatibility; security trumps the compatibility guarantee.

APIs marked as internal

Certain APIs are explicitly marked as "internal" in a couple of ways:

- Some documentation refers to internals and mentions them as such. If the documentation says that something is internal, we reserve the right to change it.
- Functions, methods, and other objects prefixed by a leading underscore (_). This is the standard Python way of indicating that something is private; if any method starts with a single _, it's an internal API.

7.2 Design philosophies

This document explains some of the fundamental philosophies Django's developers have used in creating the framework. Its goal is to explain the past and guide the future.

7.2.1 Overall

Loose coupling

A fundamental goal of Django's stack is loose coupling and tight cohesion. The various layers of the framework shouldn't "know" about each other unless absolutely necessary.

For example, the template system knows nothing about Web requests, the database layer knows nothing about data display and the view system doesn't care which template system a programmer uses.

Although Django comes with a full stack for convenience, the pieces of the stack are independent of another wherever possible.

Less code

Django apps should use as little code as possible; they should lack boilerplate. Django should take full advantage of Python's dynamic capabilities, such as introspection.

Quick development

The point of a Web framework in the 21st century is to make the tedious aspects of Web development fast. Django should allow for incredibly quick Web development.

Don't repeat yourself (DRY)

Every distinct concept and/or piece of data should live in one, and only one, place. Redundancy is bad. Normalization is good.

The framework, within reason, should deduce as much as possible from as little as possible.

:

The discussion of DRY on the Portland Pattern Repository

Explicit is better than implicit

This is a core Python principle listed in **PEP 20**, and it means Django shouldn't do too much "magic." Magic shouldn't happen unless there's a really good reason for it. Magic is worth using only if it creates a huge convenience unattainable in other ways, and it isn't implemented in a way that confuses developers who are trying to learn how to use the feature.

Consistency

The framework should be consistent at all levels. Consistency applies to everything from low-level (the Python coding style used) to high-level (the "experience" of using Django).

7.2.2 Models

Explicit is better than implicit

Fields shouldn't assume certain behaviors based solely on the name of the field. This requires too much knowledge of the system and is prone to errors. Instead, behaviors should be based on keyword arguments and, in some cases, on the type of the field.

Include all relevant domain logic

Models should encapsulate every aspect of an "object," following Martin Fowler's Active Record design pattern.

This is why both the data represented by a model and information about it (its human-readable name, options like default ordering, etc.) are defined in the model class; all the information needed to understand a given model should be stored *in* the model.

7.2.3 Database API

The core goals of the database API are:

SQL efficiency

It should execute SQL statements as few times as possible, and it should optimize statements internally.

This is why developers need to call `save()` explicitly, rather than the framework saving things behind the scenes silently.

This is also why the `select_related()` `QuerySet` method exists. It's an optional performance booster for the common case of selecting "every related object."

Terse, powerful syntax

The database API should allow rich, expressive statements in as little syntax as possible. It should not rely on importing other modules or helper objects.

Joins should be performed automatically, behind the scenes, when necessary.

Every object should be able to access every related object, systemwide. This access should work both ways.

Option to drop into raw SQL easily, when needed

The database API should realize it's a shortcut but not necessarily an end-all-be-all. The framework should make it easy to write custom SQL – entire statements, or just custom `WHERE` clauses as custom parameters to API calls.

7.2.4 URL design

Loose coupling

URLs in a Django app should not be coupled to the underlying Python code. Tying URLs to Python function names is a Bad And Ugly Thing.

Along these lines, the Django URL system should allow URLs for the same app to be different in different contexts. For example, one site may put stories at `/stories/`, while another may use `/news/`.

Infinite flexibility

URLs should be as flexible as possible. Any conceivable URL design should be allowed.

Encourage best practices

The framework should make it just as easy (or even easier) for a developer to design pretty URLs than ugly ones.

File extensions in Web-page URLs should be avoided.

Vignette-style commas in URLs deserve severe punishment.

Definitive URLs

Technically, `foo.com/bar` and `foo.com/bar/` are two different URLs, and search-engine robots (and some Web traffic-analyzing tools) would treat them as separate pages. Django should make an effort to "normalize" URLs so that search-engine robots don't get confused.

This is the reasoning behind the *APPEND_SLASH* setting.

7.2.5 Template system

Separate logic from presentation

We see a template system as a tool that controls presentation and presentation-related logic – and that's it. The template system shouldn't support functionality that goes beyond this basic goal.

Discourage redundancy

The majority of dynamic websites use some sort of common sitewide design – a common header, footer, navigation bar, etc. The Django template system should make it easy to store those elements in a single place, eliminating duplicate code.

This is the philosophy behind *template inheritance*.

Be decoupled from HTML

The template system shouldn't be designed so that it only outputs HTML. It should be equally good at generating other text-based formats, or just plain text.

XML should not be used for template languages

Using an XML engine to parse templates introduces a whole new world of human error in editing templates – and incurs an unacceptable level of overhead in template processing.

Assume designer competence

The template system shouldn't be designed so that templates necessarily are displayed nicely in WYSIWYG editors such as Dreamweaver. That is too severe of a limitation and wouldn't allow the syntax to be as nice as it is. Django expects template authors are comfortable editing HTML directly.

Treat whitespace obviously

The template system shouldn't do magic things with whitespace. If a template includes whitespace, the system should treat the whitespace as it treats text – just display it. Any whitespace that's not in a template tag should be displayed.

Don't invent a programming language

The goal is not to invent a programming language. The goal is to offer just enough programming-esque functionality, such as branching and looping, that is essential for making presentation-related decisions. The *Django Template Language (DTL)* aims to avoid advanced logic.

The Django template system recognizes that templates are most often written by *designers*, not *programmers*, and therefore should not assume Python knowledge.

Safety and security

The template system, out of the box, should forbid the inclusion of malicious code – such as commands that delete database records.

This is another reason the template system doesn't allow arbitrary Python code.

Extensibility

The template system should recognize that advanced template authors may want to extend its technology.

This is the philosophy behind custom template tags and filters.

7.2.6 Views

Simplicity

Writing a view should be as simple as writing a Python function. Developers shouldn't have to instantiate a class when a function will do.

Use request objects

Views should have access to a request object – an object that stores metadata about the current request. The object should be passed directly to a view function, rather than the view function having to access the request data from a global variable. This makes it light, clean and easy to test views by passing in "fake" request objects.

Loose coupling

A view shouldn't care about which template system the developer uses – or even whether a template system is used at all.

Differentiate between GET and POST

GET and POST are distinct; developers should explicitly use one or the other. The framework should make it easy to distinguish between GET and POST data.

7.2.7 Cache Framework

The core goals of Django's cache framework are:

Less code

A cache should be as fast as possible. Hence, all framework code surrounding the cache backend should be kept to the absolute minimum, especially for `get()` operations.

Consistency

The cache API should provide a consistent interface across the different cache backends.

Extensibility

The cache API should be extensible at the application level based on the developer's needs (for example, see *Cache key transformation*).

7.3 Third-party distributions of Django

Many third-party distributors are now providing versions of Django integrated with their package-management systems. These can make installation and upgrading much easier for users of Django since the integration includes the ability to automatically install dependencies (like database adapters) that Django requires.

Typically, these packages are based on the latest stable release of Django, so if you want to use the development version of Django you'll need to follow the instructions for *installing the development version* from our Git repository.

If you're using Linux or a Unix installation, such as OpenSolaris, check with your distributor to see if they already package Django. If you're using a Linux distro and don't know how to find out if a package is available, then now is a good time to learn. The Django Wiki contains a list of Third Party Distributions to help you out.

7.3.1 For distributors

If you'd like to package Django for distribution, we'd be happy to help out! Please join the *django-developers* mailing list and introduce yourself.

We also encourage all distributors to subscribe to the *django-announce* mailing list, which is a (very) low-traffic list for announcing new releases of Django and important bugfixes.

CHAPTER 8

Glossary

concrete model A non-abstract (`abstract=False`) model.

field An attribute on a *model*; a given field usually maps directly to a single database column.

See Models.

generic view A higher-order *view* function that provides an abstract/generic implementation of a common idiom or pattern found in view development.

See Class-based views.

model Models store your application's data.

See Models.

MTV "Model-template-view"; a software pattern, similar in style to MVC, but a better description of the way Django does things.

See *the FAQ entry*.

MVC Model-view-controller; a software pattern. Django *follows MVC to some extent*.

project A Python package – i.e. a directory of code – that contains all the settings for an instance of Django. This would include database configuration, Django-specific options and application-specific settings.

property Also known as "managed attributes", and a feature of Python since version 2.2. This is a neat way to implement attributes whose usage resembles attribute access, but whose implementation uses method calls.

See `property`.

queryset An object representing some set of rows to be fetched from the database.

See Making queries.

slug A short label for something, containing only letters, numbers, underscores or hyphens. They're generally used in URLs. For example, in a typical blog entry URL:

`https://www.djangoproject.com/weblog/2008/apr/12/spring/`

the last bit (`spring`) is the slug.

template A chunk of text that acts as formatting for representing data. A template helps to abstract the presentation of data from the data itself.

See Templates.

view A function responsible for rendering a page.

CHAPTER 9

Release notes

Release notes for the official Django releases. Each release note will tell you what's new in each version, and will also describe any backwards-incompatible changes made in that version.

For those upgrading to a new version of Django, you will need to check all the backwards-incompatible changes and deprecated features for each 'final' release from the one after your current Django version, up to and including the new version.

9.1 Final releases

Below are release notes through Django 1.10 and its patch releases. Newer versions of the documentation contain the release notes for any later releases.

9.1.1 1.10 release

Django 1.10.3 release notes

November 1, 2016

Django 1.10.3 fixes two security issues and several bugs in 1.10.2.

User with hardcoded password created when running tests on Oracle

When running tests with an Oracle database, Django creates a temporary database user. In older versions, if a password isn't manually specified in the database settings `TEST` dictionary, a hardcoded password is used. This could allow an attacker with network access to the database server to connect.

This user is usually dropped after the test suite completes, but not when using the `manage.py test --keepdb` option or if the user has an active session (such as an attacker's connection).

A randomly generated password is now used for each test run.

DNS rebinding vulnerability when `DEBUG=True`

Older versions of Django don't validate the `Host` header against `settings.ALLOWED_HOSTS` when `settings.DEBUG=True`. This makes them vulnerable to a DNS rebinding attack.

While Django doesn't ship a module that allows remote code execution, this is at least a cross-site scripting vector, which could be quite serious if developers load a copy of the production database in development or connect to some production services for which there's no development instance, for example. If a project uses a package like the `django-debug-toolbar`, then the attacker could execute arbitrary SQL, which could be especially bad if the developers connect to the database with a superuser account.

`settings.ALLOWED_HOSTS` is now validated regardless of `DEBUG`. For convenience, if `ALLOWED_HOSTS` is empty and `DEBUG=True`, the following variations of localhost are allowed `['localhost', '127.0.0.1', '::1']`. If your local settings file has your production `ALLOWED_HOSTS` value, you must now omit it to get those fallback values.

Bugfixes

- Allowed `User.is_authenticated` and `User.is_anonymous` properties to be tested for `set` membership (#27309).
- Fixed a performance regression when running `migrate` in projects with `RenameModel` operations (#27279).
- Added `model_name` to the `allow_migrate()` calls in `makemigrations` (#27200).
- Made the `JavaScriptCatalog` view respect the `packages` argument; previously it was ignored (#27374).
- Fixed `QuerySet.bulk_create()` on PostgreSQL when the number of objects is a multiple plus one of `batch_size` (#27385).
- Prevented `i18n_patterns()` from using too much of the URL as the language to fix a use case for `prefix_default_language=False` (#27063).
- Replaced a possibly incorrect redirect from `SessionMiddleware` when a session is destroyed in a concurrent request with a `SuspiciousOperation` to indicate that the request can't be completed (#27363).

Django 1.10.2 release notes

October 1, 2016

Django 1.10.2 fixes several bugs in 1.10.1.

Bugfixes

- Fixed a crash in MySQL database validation where `SELECT @@sql_mode` doesn't return a result (#27180).
- Allowed combining `contrib.postgres.search.SearchQuery` with more than one `&` or `|` operators (#27143).
- Disabled system check for URL patterns beginning with a '/' when `APPEND_SLASH=False` (#27238).
- Fixed model form `default` fallback for `CheckboxSelectMultiple`, `MultiWidget`, `FileInput`, `SplitDateTimeWidget`, `SelectDateWidget`, and `SplitArrayWidget` (#27186). Custom widgets affected by this issue should implement *`value_omitted_from_data()`*.
- Fixed a crash in `runserver` logging during a "Broken pipe" error (#27271).
- Fixed a regression where unchanged localized date/time fields were listed as changed in the admin's model history messages (#27302).

Django 1.10.1 release notes

September 1, 2016

Django 1.10.1 fixes several bugs in 1.10.

Bugfixes

- Fixed a crash in MySQL connections where `SELECT @@SQL_AUTO_IS_NULL` doesn't return a result (#26991).
- Allowed `User.is_authenticated` and `User.is_anonymous` properties to be compared using `==`, `!=`, and `|` (#26988, #27154).
- Removed the broken `BaseCommand.usage()` method which was for `optparse` support (#27000).
- Fixed a checks framework crash with an empty `Meta.default_permissions` (#26997).
- Fixed a regression in the number of queries when using `RadioSelect` with a `ModelChoiceField` form field (#27001).
- Fixed a crash if `request.META['CONTENT_LENGTH']` is an empty string (#27005).
- Fixed the `isnull` lookup on a `ForeignKey` with its `to_field` pointing to a `CharField` or pointing to a `CharField` defined with `primary_key=True` (#26983).
- Prevented the `migrate` command from raising `InconsistentMigrationHistory` in the presence of unapplied squashed migrations (#27004).
- Fixed a regression in `Client.force_login()` which required specifying a `backend` rather than automatically using the first one if multiple backends are configured (#27027).
- Made `QuerySet.bulk_create()` properly initialize model instances on backends, such as PostgreSQL, that support returning the IDs of the created records so that many-to-many relationships can be used on the new objects (#27026).
- Fixed crash of `django.views.static.serve()` with `show_indexes` enabled (#26973).
- Fixed `ClearableFileInput` to avoid the `required` HTML attribute when initial data exists (#27037).
- Fixed annotations with database functions when combined with lookups on PostGIS (#27014).
- Reallowed the `{% for %}` tag to unpack any iterable (#27058).
- Made `makemigrations` skip inconsistent history checks on non-default databases if database routers aren't in use or if no apps can be migrated to the database (#27054, #27110, #27142).
- Removed duplicated managers in `Model._meta.managers` (#27073).
- Fixed `contrib.admindocs` crash when a view is in a class, such as some of the admin views (#27018).
- Reverted a few admin checks that checked `field.many_to_many` back to `isinstance(field, models.ManyToManyField)` since it turned out the checks weren't suitable to be generalized like that (#26998).
- Added the database alias to the `InconsistentMigrationHistory` message raised by `makemigrations` and `migrate` (#27089).
- Fixed the creation of `ContentType` and `Permission` objects for models of applications without migrations when calling the `migrate` command with no migrations to apply (#27044).
- Included the already applied migration state changes in the `Apps` instance provided to the `pre_migrate` signal receivers to allow `ContentType` renaming to be performed on model rename (#27100).

- Reallowed subclassing `UserCreationForm` without `USERNAME_FIELD` in `Meta.fields` (#27111).
- Fixed a regression in model forms where model fields with a `default` that didn't appear in POST data no longer used the `default` (#27039).

Django 1.10 release notes

August 1, 2016

Welcome to Django 1.10!

These release notes cover the *new features*, as well as some *backwards incompatible changes* you'll want to be aware of when upgrading from Django 1.9 or older versions. We've *dropped some features* that have reached the end of their deprecation cycle, and we've *begun the deprecation process for some features*.

See the Upgrading Django to a newer version guide if you're updating an existing project.

Python compatibility

Like Django 1.9, Django 1.10 requires Python 2.7, 3.4, or 3.5. We **highly recommend** and only officially support the latest release of each series.

What's new in Django 1.10

Full text search for PostgreSQL `django.contrib.postgres` now includes a collection of database functions to allow the use of the full text search engine. You can search across multiple fields in your relational database, combine the searches with other lookups, use different language configurations and weightings, and rank the results by relevance.

It also now includes trigram support, using the *`trigram_similar`* lookup, and the *`TrigramSimilarity`* and *`TrigramDistance`* expressions.

New-style middleware A new style of middleware is introduced to solve the lack of strict request/response layering of the old-style of middleware described in DEP 0005. You'll need to *adapt old, custom middleware* and switch from the `MIDDLEWARE_CLASSES` setting to the new *`MIDDLEWARE`* setting to take advantage of the improvements.

Official support for Unicode usernames The *`User`* model in `django.contrib.auth` originally only accepted ASCII letters in usernames. Although it wasn't a deliberate choice, Unicode characters have always been accepted when using Python 3.

The username validator now explicitly accepts Unicode letters by default on Python 3 only. This default behavior can be overridden by changing the *`username_validator`* attribute of the `User` model, or to any proxy of that model, using either *`ASCIIUsernameValidator`* or *`UnicodeUsernameValidator`*. Custom user models may also use those validators.

Minor features

`django.contrib.admin`

- For sites running on a subpath, the default *`URL for the "View site" link`* at the top of each admin page will now point to `request.META['SCRIPT_NAME']` if set, instead of `/`.

- The success message that appears after adding or editing an object now contains a link to the object's change form.
- All inline JavaScript is removed so you can enable the `Content-Security-Policy` HTTP header if you wish.
- The new *`InlineModelAdmin.classes`* attribute allows specifying classes on inline fieldsets. Inlines with a `collapse` class will be initially collapsed and their header will have a small "show" link.
- If a user doesn't have the add permission, the `object-tools` block on a model's changelist will now be rendered (without the add button, of course). This makes it easier to add custom tools in this case.
- The *`LogEntry`* model now stores change messages in a JSON structure so that the message can be dynamically translated using the current active language. A new `LogEntry.get_change_message()` method is now the preferred way of retrieving the change message.
- Selected objects for fields in `ModelAdmin.raw_id_fields` now have a link to object's change form.
- Added "No date" and "Has date" choices for `DateFieldListFilter` if the field is nullable.
- The jQuery library embedded in the admin is upgraded from version 2.1.4 to 2.2.3.

`django.contrib.auth`

- Added support for the *Argon2 password hash*. It's recommended over PBKDF2, however, it's not the default as it requires a third-party library.
- The default iteration count for the PBKDF2 password hasher has been increased by 25%. This backwards compatible change will not affect users who have subclassed `django.contrib.auth.hashers.PBKDF2PasswordHasher` to change the default value.
- The *`logout()`* view sends "no-cache" headers to prevent an issue where Safari caches redirects and prevents a user from being able to log out.
- Added the optional `backend` argument to *`login()`* to allow using it without credentials.
- The new *`LOGOUT_REDIRECT_URL`* setting controls the redirect of the *`logout()`* view, if the view doesn't get a `next_page` argument.
- The new `redirect_authenticated_user` parameter for the *`login()`* view allows redirecting authenticated users visiting the login page.
- The new *`AllowAllUsersModelBackend`* and *`AllowAllUsersRemoteUserBackend`* ignore the value of `User.is_active`, while *`ModelBackend`* and *`RemoteUserBackend`* now reject inactive users.

`django.contrib.gis`

- *Distance lookups* now accept expressions as the distance value parameter.
- The new *`GEOSGeometry.unary_union`* property computes the union of all the elements of this geometry.
- Added the *`GEOSGeometry.covers()`* binary predicate.
- Added the *`GDALBand.statistics()`* method and *`mean`* and *`std`* attributes.
- Added support for the *`MakeLine`* aggregate and *`GeoHash`* function on SpatiaLite.
- Added support for the *`Difference`*, *`Intersection`*, and *`SymDifference`* functions on MySQL.
- Added support for instantiating empty GEOS geometries.
- The new *`trim`* and *`precision`* properties of *`WKTWriter`* allow controlling output of the fractional part of the coordinates in WKT.

- Added the *LineString.closed* and *MultiLineString.closed* properties.
- The GeoJSON serializer now outputs the primary key of objects in the `properties` dictionary if specific fields aren't specified.
- The ability to replicate input data on the *GDALBand.data()* method was added. Band data can now be updated with repeated values efficiently.
- Added database functions *IsValid* and *MakeValid*, as well as the *isvalid* lookup, all for PostGIS. This allows filtering and repairing invalid geometries on the database side.
- Added raster support for all spatial lookups.

django.contrib.postgres

- For convenience, *HStoreField* now casts its keys and values to strings.

django.contrib.sessions

- The *clearsessions* management command now removes file-based sessions.

django.contrib.sites

- The *Site* model now supports *natural keys*.

django.contrib.staticfiles

- The *static* template tag now uses `django.contrib.staticfiles` if it's in `INSTALLED_APPS`. This is especially useful for third-party apps which can now always use `{% load static %}` (instead of `{% load staticfiles %}` or `{% load static from staticfiles %}`) and not worry about whether or not the `staticfiles` app is installed.
- You can *more easily customize* the `collectstatic --ignore_patterns` option with a custom `AppConfig`.

Cache

- The file-based cache backend now uses the highest pickling protocol.

CSRF

- The default *CSRF_FAILURE_VIEW*, `views.csrf.csrf_failure()` now accepts an optional `template_name` parameter, defaulting to `'403_csrf.html'`, to control the template used to render the page.
- To protect against BREACH attacks, the CSRF protection mechanism now changes the form token value on every request (while keeping an invariant secret which can be used to validate the different tokens).

Database backends

- Temporal data subtraction was unified on all backends.
- If the database supports it, backends can set `DatabaseFeatures.can_return_ids_from_bulk_insert=True` and implement `DatabaseOperations.fetch_returned_insert_ids()` to set primary keys on objects created using `QuerySet.bulk_create()`.

- Added keyword arguments to the `as_sql()` methods of various expressions (`Func`, `When`, `Case`, and `OrderBy`) to allow database backends to customize them without mutating `self`, which isn't safe when using different database backends. See the `arg_joiner` and `**extra_context` parameters of *`Func.as_sql()`* for an example.

File Storage

- Storage backends now present a timezone-aware API with new methods *`get_accessed_time()`*, *`get_created_time()`*, and *`get_modified_time()`*. They return a timezone-aware `datetime` if *`USE_TZ`* is `True` and a naive `datetime` in the local timezone otherwise.
- The new *`Storage.generate_filename()`* method makes it easier to implement custom storages that don't use the `os.path` calls previously in *`FileField`*.

Forms

- Form and widget `Media` is now served using *`django.contrib.staticfiles`* if installed.
- The `<input>` tag rendered by *`CharField`* now includes a `minlength` attribute if the field has a `min_length`.
- Required form fields now have the `required` HTML attribute. Set the new *`Form.use_required_attribute`* attribute to `False` to disable it. The `required` attribute isn't included on forms of formsets because the browser validation may not be correct when adding and deleting formsets.

Generic Views

- The *`View`* class can now be imported from `django.views`.

Internationalization

- The *`i18n_patterns()`* helper function can now be used in a root URLConf specified using *`request.urlconf`*.
- By setting the new `prefix_default_language` parameter for *`i18n_patterns()`* to `False`, you can allow accessing the default language without a URL prefix.
- *`set_language()`* now returns a 204 status code (No Content) for AJAX requests when there is no `next` parameter in `POST` or `GET`.
- The *`JavaScriptCatalog`* and *`JSONCatalog`* class-based views supersede the deprecated `javascript_catalog()` and `json_catalog()` function-based views. The new views are almost equivalent to the old ones except that by default the new views collect all JavaScript strings in the `djangojs` translation domain from all installed apps rather than only the JavaScript strings from *`LOCALE_PATHS`*.

Management Commands

- *`call_command()`* now returns the value returned from the `command.handle()` method.
- The new *`check --fail-level`* option allows specifying the message level that will cause the command to exit with a non-zero status.
- The new *`makemigrations --check`* option makes the command exit with a non-zero status when model changes without migrations are detected.
- *`makemigrations`* now displays the path to the migration files that it generates.

- The *shell --interface* option now accepts `python` to force use of the "plain" Python interpreter.
- The new *shell --command* option lets you run a command as Django and exit, instead of opening the interactive shell.
- Added a warning to *dumpdata* if a proxy model is specified (which results in no output) without its concrete parent.
- The new *BaseCommand.requires_migrations_checks* attribute may be set to `True` if you want your command to print a warning, like *runserver* does, if the set of migrations on disk don't match the migrations in the database.
- To assist with testing, *call_command()* now accepts a command object as the first argument.
- The *shell* command supports tab completion on systems using `libedit`, e.g. Mac OSX.
- The *inspectdb* command lets you choose what tables should be inspected by specifying their names as arguments.

Migrations

- Added support for serialization of `enum.Enum` objects.
- Added the `elidable` argument to the *RunSQL* and *RunPython* operations to allow them to be removed when squashing migrations.
- Added support for *non-atomic migrations* by setting the `atomic` attribute on a `Migration`.
- The `migrate` and `makemigrations` commands now *check for a consistent migration history*. If they find some unapplied dependencies of an applied migration, `InconsistentMigrationHistory` is raised.
- The *pre_migrate()* and *post_migrate()* signals now dispatch their migration `plan` and `apps`.

Models

- Reverse foreign keys from proxy models are now propagated to their concrete class. The reverse relation attached by a *ForeignKey* pointing to a proxy model is now accessible as a descriptor on the proxied model class and may be referenced in queryset filtering.
- The new *Field.rel_db_type()* method returns the database column data type for fields such as `ForeignKey` and `OneToOneField` that point to another field.
- The *arity* class attribute is added to *Func*. This attribute can be used to set the number of arguments the function accepts.
- Added *BigAutoField* which acts much like an *AutoField* except that it is guaranteed to fit numbers from `1` to `9223372036854775807`.
- *QuerySet.in_bulk()* may be called without any arguments to return all objects in the queryset.
- *related_query_name* now supports app label and class interpolation using the `'%(app_label)s'` and `'%(class)s'` strings.
- Allowed overriding model fields inherited from abstract base classes.
- The *prefetch_related_objects()* function is now a public API.
- *QuerySet.bulk_create()* sets the primary key on objects when using PostgreSQL.
- Added the *Cast* database function.
- A proxy model may now inherit multiple proxy models that share a common non-abstract parent class.
- Added *Extract* functions to extract datetime components as integers, such as year and hour.

- Added *Trunc* functions to truncate a date or datetime to a significant component. They enable queries like sales-per-day or sales-per-hour.
- `Model.__init__()` now sets values of virtual fields from its keyword arguments.
- The new *Meta.base_manager_name* and *Meta.default_manager_name* options allow controlling the *_base_manager* and *_default_manager*, respectively.

Requests and Responses

- Added `request.user` to the debug view.
- Added *HttpResponse* methods *readable()* and *seekable()* to make an instance a stream-like object and allow wrapping it with `io.TextIOWrapper`.
- Added the *HttpResponse.content_type* and *content_params* attributes which are parsed from the `CONTENT_TYPE` header.
- The parser for `request.COOKIES` is simplified to better match the behavior of browsers. `request.COOKIES` may now contain cookies that are invalid according to **RFC 6265** but are possible to set via `document.cookie`.

Serialization

- The `django.core.serializers.json.DjangoJSONEncoder` now knows how to serialize lazy strings, typically used for translatable content.

Templates

- Added the `autoescape` option to the *DjangoTemplates* backend and the *Engine* class.
- Added the `is` and `is not` comparison operators to the *if* tag.
- Allowed *dictsort* to order a list of lists by an element at a specified index.
- The *debug()* context processor contains queries for all database aliases instead of only the default alias.
- Added relative path support for string arguments of the *extends* and *include* template tags.

Tests

- To better catch bugs, *TestCase* now checks deferrable database constraints at the end of each test.
- Tests and test cases can be *marked with tags* and run selectively with the new *test --tag* and *test --exclude-tag* options.
- You can now login and use sessions with the test client even if *django.contrib.sessions* is not in *INSTALLED_APPS*.

URLs

- An addition in *django.setup()* allows URL resolving that happens outside of the request/response cycle (e.g. in management commands and standalone scripts) to take *FORCE_SCRIPT_NAME* into account when it is set.

Validators

- *URLValidator* now limits the length of domain name labels to 63 characters and the total length of domain names to 253 characters per **RFC 1034**.
- *int_list_validator()* now accepts an optional `allow_negative` boolean parameter, defaulting to `False`, to allow negative integers.

Backwards incompatible changes in 1.10

: In addition to the changes outlined in this section, be sure to review the *Features removed in 1.10* for the features that have reached the end of their deprecation cycle and therefore been removed. If you haven't updated your code within the deprecation timeline for a given feature, its removal may appear as a backwards incompatible change.

Database backend API

- GIS's `AreaField` uses an unspecified underlying numeric type that could in practice be any numeric Python type. `decimal.Decimal` values retrieved from the database are now converted to `float` to make it easier to combine them with values used by the GIS libraries.
- In order to enable temporal subtraction you must set the `supports_temporal_subtraction` database feature flag to `True` and implement the `DatabaseOperations.subtract_temporals()` method. This method should return the SQL and parameters required to compute the difference in microseconds between the `lhs` and `rhs` arguments in the datatype used to store *DurationField*.

`select_related()` prohibits non-relational fields for nested relations Django 1.8 added validation for non-relational fields in `select_related()`:

```
>>> Book.objects.select_related('title')
Traceback (most recent call last):
...
FieldError: Non-relational field given in select_related: 'title'
```

But it didn't prohibit nested non-relation fields as it does now:

```
>>> Book.objects.select_related('author__name')
Traceback (most recent call last):
...
FieldError: Non-relational field given in select_related: 'name'
```

`_meta.get_fields()` returns consistent reverse fields for proxy models Before Django 1.10, the *get_fields()* method returned different reverse fields when called on a proxy model compared to its proxied concrete class. This inconsistency was fixed by returning the full set of fields pointing to a concrete class or one of its proxies in both cases.

`AbstractUser.username` `max_length` increased to 150 A migration for *django.contrib.auth.models.User.username* is included. If you have a custom user model inheriting from `AbstractUser`, you'll need to generate and apply a database migration for your user model.

We considered an increase to 254 characters to more easily allow the use of email addresses (which are limited to 254 characters) as usernames but rejected it due to a MySQL limitation. When using the `utf8mb4` encoding (recommended for proper Unicode support), MySQL can only create unique indexes with 191 characters by default. Therefore, if you need a longer length, please use a custom user model.

If you want to preserve the 30 character limit for usernames, use a custom form when creating a user or changing usernames:

```
from django.contrib.auth.forms import UserCreationForm

class MyUserCreationForm(UserCreationForm):
    username = forms.CharField(
        max_length=30,
        help_text='Required. 30 characters or fewer. Letters, digits and @/./+/-/_ only.',
    )
```

If you wish to keep this restriction in the admin, set `UserAdmin.add_form` to use this form:

```
from django.contrib.auth.admin import UserAdmin as BaseUserAdmin
from django.contrib.auth.models import User

class UserAdmin(BaseUserAdmin):
    add_form = MyUserCreationForm

admin.site.unregister(User)
admin.site.register(User, UserAdmin)
```

Dropped support for PostgreSQL 9.1 Upstream support for PostgreSQL 9.1 ends in September 2016. As a consequence, Django 1.10 sets PostgreSQL 9.2 as the minimum version it officially supports.

`runserver` output goes through logging Request and response handling of the `runserver` command is sent to the *django.server* logger instead of to `sys.stderr`. If you disable Django's logging configuration or override it with your own, you'll need to add the appropriate logging configuration if you want to see that output:

```
'formatters': {
    'django.server': {
        '()': 'django.utils.log.ServerFormatter',
        'format': '[%(server_time)s] %(message)s',
    }
},
'handlers': {
    'django.server': {
        'level': 'INFO',
        'class': 'logging.StreamHandler',
        'formatter': 'django.server',
    },
},
'loggers': {
    'django.server': {
        'handlers': ['django.server'],
        'level': 'INFO',
        'propagate': False,
    }
}
```

`auth.CustomUser` and `auth.ExtensionUser` test models were removed Since the introduction of migrations for the contrib apps in Django 1.8, the tables of these custom user test models were not created anymore making them unusable in a testing context.

Apps registry is no longer auto-populated when unpickling models outside of Django The apps registry is no longer auto-populated when unpickling models. This was added in Django 1.7.2 as an attempt to allow unpickling models outside of Django, such as in an RQ worker, without calling `django.setup()`, but it creates the possibility of a deadlock. To adapt your code in the case of RQ, you can provide your own worker script that calls `django.setup()`.

Removed null assignment check for non-null foreign key fields In older versions, assigning `None` to a non-nullable `ForeignKey` or `OneToOneField` raised `ValueError('Cannot assign None: "model.field" does not allow null values.')`. For consistency with other model fields which don't have a similar check, this check is removed.

Removed weak password hashers from the default `PASSWORD_HASHERS` setting Django 0.90 stored passwords as unsalted MD5. Django 0.91 added support for salted SHA1 with automatic upgrade of passwords when a user logs in. Django 1.4 added PBKDF2 as the default password hasher.

If you have an old Django project with MD5 or SHA1 (even salted) encoded passwords, be aware that these can be cracked fairly easily with today's hardware. To make Django users acknowledge continued use of weak hashers, the following hashers are removed from the default *`PASSWORD_HASHERS`* setting:

```
'django.contrib.auth.hashers.SHA1PasswordHasher'
'django.contrib.auth.hashers.MD5PasswordHasher'
'django.contrib.auth.hashers.UnsaltedSHA1PasswordHasher'
'django.contrib.auth.hashers.UnsaltedMD5PasswordHasher'
'django.contrib.auth.hashers.CryptPasswordHasher'
```

Consider using a *wrapped password hasher* to strengthen the hashes in your database. If that's not feasible, add the *`PASSWORD_HASHERS`* setting to your project and add back any hashers that you need.

You can check if your database has any of the removed hashers like this:

```
from django.contrib.auth import get_user_model
User = get_user_model()

# Unsalted MD5/SHA1:
User.objects.filter(password__startswith='md5$$')
User.objects.filter(password__startswith='sha1$$')
# Salted MD5/SHA1:
User.objects.filter(password__startswith='md5$').exclude(password__startswith='md5$$')
User.objects.filter(password__startswith='sha1$').exclude(password__startswith='sha1$$')
# Crypt hasher:
User.objects.filter(password__startswith='crypt$$')

from django.db.models import CharField
from django.db.models.functions import Length
CharField.register_lookup(Length)
# Unsalted MD5 passwords might not have an 'md5$$' prefix:
User.objects.filter(password__length=32)
```

`Field.get_prep_lookup()` and `Field.get_db_prep_lookup()` methods are removed If you have a custom field that implements either of these methods, register a custom lookup for it. For example:

```
from django.db.models import Field
from django.db.models.lookups import Exact

class MyField(Field):
    ...
```

```
class MyFieldExact(Exact):
    def get_prep_lookup(self):
        # do_custom_stuff_for_myfield
        ....

MyField.register_lookup(MyFieldExact)
```

django.contrib.gis

- Support for SpatiaLite < 3.0 and GEOS < 3.3 is dropped.
- The `add_postgis_srs()` backwards compatibility alias for `django.contrib.gis.utils.add_srs_entry()` is removed.
- On Oracle/GIS, the *`Area`* aggregate function now returns a `float` instead of `decimal.Decimal`. (It's still wrapped in a measure of square meters.)
- The default *`GEOSGeometry`* representation (WKT output) is trimmed by default. That is, instead of `POINT (23.0000000000000000 5.5000000000000000)`, you'll get `POINT (23 5.5)`.

Maximum size of a request body and the number of GET/POST parameters is limited Two new settings help mitigate denial-of-service attacks via large requests:

- *`DATA_UPLOAD_MAX_MEMORY_SIZE`* limits the size that a request body may be. File uploads don't count towards this limit.
- *`DATA_UPLOAD_MAX_NUMBER_FIELDS`* limits the number of GET/POST parameters that are parsed.

Applications that receive unusually large form posts may need to tune these settings.

Miscellaneous

- The `repr()` of a `QuerySet` is wrapped in `<QuerySet >` to disambiguate it from a plain list when debugging.
- `utils.version.get_version()` returns **PEP 440** compliant release candidate versions (e.g. '1.10rc1' instead of '1.10c1').
- CSRF token values are now required to be strings of 64 alphanumerics; values of 32 alphanumerics, as set by older versions of Django by default, are automatically replaced by strings of 64 characters. Other values are considered invalid. This should only affect developers or users who replace these tokens.
- The `LOGOUT_URL` setting is removed as Django hasn't made use of it since pre-1.0. If you use it in your project, you can add it to your project's settings. The default value was `'/accounts/logout/'`.
- Objects with a `close()` method such as files and generators passed to *`HttpResponse`* are now closed immediately instead of when the WSGI server calls `close()` on the response.
- A redundant `transaction.atomic()` call in `QuerySet.update_or_create()` is removed. This may affect query counts tested by `TransactionTestCase.assertNumQueries()`.
- Support for `skip_validation` in `BaseCommand.execute(**options)` is removed. Use `skip_checks` (added in Django 1.7) instead.
- *`loaddata`* now raises a `CommandError` instead of showing a warning when the specified fixture file is not found.
- Instead of directly accessing the `LogEntry.change_message` attribute, it's now better to call the `LogEntry.get_change_message()` method which will provide the message in the current language.

- The default error views now raise `TemplateDoesNotExist` if a nonexistent `template_name` is specified.
- The unused `choices` keyword argument of the `Select` and `SelectMultiple` widgets' `render()` method is removed. The `choices` argument of the `render_options()` method is also removed, making `selected_choices` the first argument.
- Tests that violate deferrable database constraints will now error when run on a database that supports deferrable constraints.
- Built-in management commands now use indexing of keys in `options`, e.g. `options['verbosity']`, instead of `options.get()` and no longer perform any type coercion. This could be a problem if you're calling commands using `Command.execute()` (which bypasses the argument parser that sets a default value) instead of *`call_command()`*. Instead of calling `Command.execute()`, pass the command object as the first argument to `call_command()`.
- *`ModelBackend`* and *`RemoteUserBackend`* now reject inactive users. This means that inactive users can't login and will be logged out if they are switched from `is_active=True` to `False`. If you need the previous behavior, use the new *`AllowAllUsersModelBackend`* or *`AllowAllUsersRemoteUserBackend`* in *`AUTHENTICATION_BACKENDS`* instead.
- In light of the previous change, the test client's *`login()`* method no longer always rejects inactive users but instead delegates this decision to the authentication backend. *`force_login()`* also delegates the decision to the authentication backend, so if you're using the default backends, you need to use an active user.
- *`django.views.i18n.set_language()`* may now return a 204 status code for AJAX requests.
- The `base_field` attribute of *`RangeField`* is now a type of field, not an instance of a field. If you have created a custom subclass of *`RangeField`*, you should change the `base_field` attribute.
- Middleware classes are now initialized when the server starts rather than during the first request.
- If you override `is_authenticated()` or `is_anonymous()` in a custom user model, you must convert them to attributes or properties as described in *the deprecation note*.
- When using *`ModelAdmin.save_as=True`*, the "Save as new" button now redirects to the change view for the new object instead of to the model's changelist. If you need the previous behavior, set the new *`ModelAdmin.save_as_continue`* attribute to `False`.
- Required form fields now have the `required` HTML attribute. Set the *`Form.use_required_attribute`* attribute to `False` to disable it. You could also add the `novalidate` attribute to `<form>` if you don't want browser validation. To disable the `required` attribute on custom widgets, override the *`Widget.use_required_attribute()`* method.
- The WSGI handler no longer removes content of responses from `HEAD` requests or responses with a `status_code` of 100-199, 204, or 304. Most Web servers already implement this behavior. Responses retrieved using the Django test client continue to have these "response fixes" applied.
- `Model.__init__()` now receives `django.db.models.DEFERRED` as the value of deferred fields.
- The `Model._deferred` attribute is removed as dynamic model classes when using `QuerySet.defer()` and `only()` is removed.
- *`Storage.save()`* no longer replaces `'\'` with `'/'`. This behavior is moved to *`FileSystemStorage`* since this is a storage specific implementation detail. Any Windows user with a custom storage implementation that relies on this behavior will need to implement it in the custom storage's `save()` method.
- Private *`FileField`* methods `get_directory_name()` and `get_filename()` are no longer called (and are now deprecated) which is a backwards incompatible change for users overriding those methods on custom fields. To adapt such code, override `FileField.generate_filename()` or *`Storage.generate_filename()`* instead. It might be possible to use *`upload_to`* also.

- The subject of mail sent by `AdminEmailHandler` is no longer truncated at 989 characters. If you were counting on a limited length, truncate the subject yourself.
- Private expressions `django.db.models.expressions.Date` and `DateTime` are removed. The new *`Trunc`* expressions provide the same functionality.
- The `_base_manager` and `_default_manager` attributes are removed from model instances. They remain accessible on the model class.
- Accessing a deleted field on a model instance, e.g. after `del obj.field`, reloads the field's value instead of raising `AttributeError`.
- If you subclass `AbstractBaseUser` and override `clean()`, be sure it calls `super()`. *`AbstractBaseUser.normalize_username()`* is called in a new *`AbstractBaseUser.clean()`* method.
- Private API `django.forms.models.model_to_dict()` returns a queryset rather than a list of primary keys for `ManyToManyFields` .
- If `django.contrib.staticfiles` is installed, the *`static`* template tag uses the `staticfiles` storage to construct the URL rather than simply joining the value with `STATIC_ROOT`. The new approach encodes the URL, which could be backwards-incompatible in cases such as including a fragment in a path, e.g. `{% static 'img.svg#fragment' %}`, since the `#` is encoded as `%23`. To adapt, move the fragment outside the template tag: `{% static 'img.svg' %}#fragment`.

Features deprecated in 1.10

Direct assignment to a reverse foreign key or many-to-many relation Instead of assigning related objects using direct assignment:

```
>>> new_list = [obj1, obj2, obj3]
>>> e.related_set = new_list
```

Use the *`set()`* method added in Django 1.9:

```
>>> e.related_set.set([obj1, obj2, obj3])
```

This prevents confusion about an assignment resulting in an implicit save.

Non-timezone-aware `Storage` API The old, non-timezone-aware methods `accessed_time()`, `created_time()`, and `modified_time()` are deprecated in favor of the new `get_*_time()` methods.

Third-party storage backends should implement the new methods and mark the old ones as deprecated. Until then, the new `get_*_time()` methods on the base *`Storage`* class convert `datetimes` from the old methods as required and emit a deprecation warning as they do so.

Third-party storage backends may retain the old methods as long as they wish to support earlier versions of Django.

`django.contrib.gis`

- The `get_srid()` and `set_srid()` methods of *`GEOSGeometry`* are deprecated in favor of the *`srid`* property.
- The `get_x()`, `set_x()`, `get_y()`, `set_y()`, `get_z()`, and `set_z()` methods of *`Point`* are deprecated in favor of the `x`, `y`, and `z` properties.
- The `get_coords()` and `set_coords()` methods of *`Point`* are deprecated in favor of the `tuple` property.
- The `cascaded_union` property of *`MultiPolygon`* is deprecated in favor of the *`unary_union`* property.

- The `django.contrib.gis.utils.precision_wkt()` function is deprecated in favor of *`WKTWriter`*.

`CommaSeparatedIntegerField` model field `CommaSeparatedIntegerField` is deprecated in favor of *`CharField`* with the *`validate_comma_separated_integer_list()`* validator:

```
from django.core.validators import validate_comma_separated_integer_list
from django.db import models

class MyModel(models.Model):
    numbers = models.CharField(..., validators=[validate_comma_separated_integer_list])
```

If you're using Oracle, `CharField` uses a different database field type (`NVARCHAR2`) than `CommaSeparatedIntegerField` (`VARCHAR2`). Depending on your database settings, this might imply a different encoding, and thus a different length (in bytes) for the same contents. If your stored values are longer than the 4000 byte limit of `NVARCHAR2`, you should use `TextField` (`NCLOB`) instead. In this case, if you have any queries that group by the field (e.g. annotating the model with an aggregation or using `distinct()`) you'll need to change them (to defer the field).

Using a model name as a query lookup when `default_related_name` is set Assume the following models:

```
from django.db import models

class Foo(models.Model):
    pass

class Bar(models.Model):
    foo = models.ForeignKey(Foo)

    class Meta:
        default_related_name = 'bars'
```

In older versions, *`default_related_name`* couldn't be used as a query lookup. This is fixed and support for the old lookup name is deprecated. For example, since `default_related_name` is set in model `Bar`, instead of using the model name `bar` as the lookup:

```
>>> bar = Bar.objects.get(pk=1)
>>> Foo.objects.get(bar=bar)
```

use the default_related_name `bars`:

```
>>> Foo.objects.get(bars=bar)
```

`__search` query lookup The `search` lookup, which supports MySQL only and is extremely limited in features, is deprecated. Replace it with a custom lookup:

```
from django.db import models

class Search(models.Lookup):
    lookup_name = 'search'

    def as_mysql(self, compiler, connection):
        lhs, lhs_params = self.process_lhs(compiler, connection)
        rhs, rhs_params = self.process_rhs(compiler, connection)
        params = lhs_params + rhs_params
        return 'MATCH (%s) AGAINST (%s IN BOOLEAN MODE)' % (lhs, rhs), params
```

```
models.CharField.register_lookup(Search)
models.TextField.register_lookup(Search)
```

Using `User.is_authenticated()` and `User.is_anonymous()` as methods The `is_authenticated()` and `is_anonymous()` methods of *AbstractBaseUser* and *AnonymousUser* classes are now properties. They will still work as methods until Django 2.0, but all usage in Django now uses attribute access.

For example, if you use *AuthenticationMiddleware* and want to know whether the user is currently logged-in you would use:

```
if request.user.is_authenticated:
    ... # Do something for logged-in users.
else:
    ... # Do something for anonymous users.
```

instead of `request.user.is_authenticated()`.

This change avoids accidental information leakage if you forget to call the method, e.g.:

```
if request.user.is_authenticated:
    return sensitive_information
```

If you override these methods in a custom user model, you must change them to properties or attributes.

Django uses a `CallableBool` object to allow these attributes to work as both a property and a method. Thus, until the deprecation period ends, you cannot compare these properties using the `is` operator. That is, the following won't work:

```
if request.user.is_authenticated is True:
    ...
```

Custom manager classes available through `prefetch_related` must define a `_apply_rel_filters()` method If you defined a custom manager class available through *prefetch_related()* you must make sure it defines a `_apply_rel_filters()` method.

This method must accept a *QuerySet* instance as its single argument and return a filtered version of the queryset for the model instance the manager is bound to.

The "escape" half of `django.utils.safestring` The `mark_for_escaping()` function and the classes it uses: `EscapeData`, `EscapeBytes`, `EscapeText`, `EscapeString`, and `EscapeUnicode` are deprecated.

As a result, the "lazy" behavior of the `escape` filter (where it would always be applied as the last filter no matter where in the filter chain it appeared) is deprecated. The filter will change to immediately apply *conditional_escape()* in Django 2.0.

`Manager.use_for_related_fields` and inheritance changes `Manager.use_for_related_fields` is deprecated in favor of setting *Meta.base_manager_name* on the model.

Model `Manager` inheritance will follow MRO inheritance rules in Django 2.0, changing the current behavior where managers defined on non-abstract base classes aren't inherited by child classes. A deprecating warning with instructions on how to adapt your code is raised if you have any affected managers. You'll either redeclare a manager from an abstract model on the child class to override the manager from the concrete model, or you'll set the model's `Meta.manager_inheritance_from_future=True` option to opt-in to the new inheritance behavior.

During the deprecation period, `use_for_related_fields` will be honored and raise a warning, even if a `base_manager_name` is set. This allows third-party code to preserve legacy behavior while transitioning to the new API. The warning can be silenced by setting `silence_use_for_related_fields_deprecation=True` on the manager.

Miscellaneous

- The `makemigrations --exit` option is deprecated in favor of the *`makemigrations --check`* option.
- `django.utils.functional.allow_lazy()` is deprecated in favor of the new *`keep_lazy()`* function which can be used with a more natural decorator syntax.
- The `shell --plain` option is deprecated in favor of `-i python` or `--interface python`.
- Importing from the `django.core.urlresolvers` module is deprecated in favor of its new location, *`django.urls`*.
- The template `Context.has_key()` method is deprecated in favor of `in`.
- The private attribute `virtual_fields` of `Model._meta` is deprecated in favor of `private_fields`.
- The private keyword arguments `virtual_only` in `Field.contribute_to_class()` and `virtual` in `Model._meta.add_field()` are deprecated in favor of `private_only` and `private`, respectively.
- The `javascript_catalog()` and `json_catalog()` views are deprecated in favor of class-based views *`JavaScriptCatalog`* and *`JSONCatalog`*.
- In multi-table inheritance, implicit promotion of a `OneToOneField` to a `parent_link` is deprecated. Add `parent_link=True` to such fields.
- The private API `Widget._format_value()` is made public and renamed to *`format_value()`*. The old name will work through a deprecation period.
- Private `FileField` methods `get_directory_name()` and `get_filename()` are deprecated in favor of performing this work in *`Storage.generate_filename()`*).
- Old-style middleware that uses `settings.MIDDLEWARE_CLASSES` are deprecated. *Adapt old, custom middleware* and use the new *`MIDDLEWARE`* setting.

Features removed in 1.10

These features have reached the end of their deprecation cycle and are removed in Django 1.10. See *Features deprecated in 1.8* for details, including how to remove usage of these features.

- Support for calling a `SQLCompiler` directly as an alias for calling its `quote_name_unless_alias` method is removed.
- The `cycle` and `firstof` template tags are removed from the `future` template tag library.
- `django.conf.urls.patterns()` is removed.
- Support for the `prefix` argument to `django.conf.urls.i18n.i18n_patterns()` is removed.
- `SimpleTestCase.urls` is removed.
- Using an incorrect count of unpacked values in the `for` template tag raises an exception rather than failing silently.
- The ability to *`reverse()`* URLs using a dotted Python path is removed.
- The ability to use a dotted Python path for the `LOGIN_URL` and `LOGIN_REDIRECT_URL` settings is removed.

- Support for `optparse` is dropped for custom management commands.
- The class `django.core.management.NoArgsCommand` is removed.
- `django.core.context_processors` module is removed.
- `django.db.models.sql.aggregates` module is removed.
- `django.contrib.gis.db.models.sql.aggregates` module is removed.
- The following methods and properties of `django.db.sql.query.Query` are removed:
  - Properties: `aggregates` and `aggregate_select`
  - Methods: `add_aggregate`, `set_aggregate_mask`, and `append_aggregate_mask`.
- `django.template.resolve_variable` is removed.
- The following private APIs are removed from *`django.db.models.options.Options`* (`Model._meta`):
  - `get_field_by_name()`
  - `get_all_field_names()`
  - `get_fields_with_model()`
  - `get_concrete_fields_with_model()`
  - `get_m2m_with_model()`
  - `get_all_related_objects()`
  - `get_all_related_objects_with_model()`
  - `get_all_related_many_to_many_objects()`
  - `get_all_related_m2m_objects_with_model()`
- The `error_message` argument of `django.forms.RegexField` is removed.
- The `unordered_list` filter no longer supports old style lists.
- Support for string `view` arguments to `url()` is removed.
- The backward compatible shim to rename `django.forms.Form._has_changed()` to `has_changed()` is removed.
- The `removetags` template filter is removed.
- The `remove_tags()` and `strip_entities()` functions in `django.utils.html` is removed.
- The `is_admin_site` argument to `django.contrib.auth.views.password_reset()` is removed.
- `django.db.models.field.subclassing.SubfieldBase` is removed.
- `django.utils.checksums` is removed.
- The `original_content_type_id` attribute on `django.contrib.admin.helpers.InlineAdminForm` is removed.
- The backwards compatibility shim to allow `FormMixin.get_form()` to be defined with no default value for its `form_class` argument is removed.
- The following settings are removed, and you must upgrade to the *`TEMPLATES`* setting:
  - `ALLOWED_INCLUDE_ROOTS`
  - `TEMPLATE_CONTEXT_PROCESSORS`

  - TEMPLATE_DEBUG
  - TEMPLATE_DIRS
  - TEMPLATE_LOADERS
  - TEMPLATE_STRING_IF_INVALID
- The backwards compatibility alias `django.template.loader.BaseLoader` is removed.
- Django template objects returned by *`get_template()`* and *`select_template()`* no longer accept a *`Context`* in their *`render()`* method.
- Template response APIs enforce the use of `dict` and backend-dependent template objects instead of *`Context`* and *`Template`* respectively.
- The `current_app` parameter for the following function and classes is removed:
  - `django.shortcuts.render()`
  - `django.template.Context()`
  - `django.template.RequestContext()`
  - `django.template.response.TemplateResponse()`
- The `dictionary` and `context_instance` parameters for the following functions are removed:
  - `django.shortcuts.render()`
  - `django.shortcuts.render_to_response()`
  - `django.template.loader.render_to_string()`
- The `dirs` parameter for the following functions is removed:
  - `django.template.loader.get_template()`
  - `django.template.loader.select_template()`
  - `django.shortcuts.render()`
  - `django.shortcuts.render_to_response()`
- Session verification is enabled regardless of whether or not `'django.contrib.auth.middleware.SessionAuthentic` is in `MIDDLEWARE_CLASSES`. `SessionAuthenticationMiddleware` no longer has any purpose and can be removed from `MIDDLEWARE_CLASSES`. It's kept as a stub until Django 2.0 as a courtesy for users who don't read this note.
- Private attribute `django.db.models.Field.related` is removed.
- The `--list` option of the `migrate` management command is removed.
- The `ssi` template tag is removed.
- Support for the = comparison operator in the `if` template tag is removed.
- The backwards compatibility shims to allow `Storage.get_available_name()` and `Storage.save()` to be defined without a `max_length` argument are removed.
- Support for the legacy `%(<foo>)s` syntax in `ModelFormMixin.success_url` is removed.
- `GeoQuerySet` aggregate methods `collect()`, `extent()`, `extent3d()`, `make_line()`, and `unionagg()` are removed.
- The ability to specify `ContentType.name` when creating a content type instance is removed.
- Support for the old signature of `allow_migrate` is removed.

- Support for the syntax of `{% cycle %}` that uses comma-separated arguments is removed.
- The warning that *`Signer`* issued when given an invalid separator is now a `ValueError`.

9.1.2 1.9 release

Django 1.9.11 release notes

November 1, 2016

Django 1.9.11 fixes two security issues in 1.9.10.

User with hardcoded password created when running tests on Oracle

When running tests with an Oracle database, Django creates a temporary database user. In older versions, if a password isn't manually specified in the database settings `TEST` dictionary, a hardcoded password is used. This could allow an attacker with network access to the database server to connect.

This user is usually dropped after the test suite completes, but not when using the `manage.py test --keepdb` option or if the user has an active session (such as an attacker's connection).

A randomly generated password is now used for each test run.

DNS rebinding vulnerability when `DEBUG=True`

Older versions of Django don't validate the `Host` header against `settings.ALLOWED_HOSTS` when `settings.DEBUG=True`. This makes them vulnerable to a DNS rebinding attack.

While Django doesn't ship a module that allows remote code execution, this is at least a cross-site scripting vector, which could be quite serious if developers load a copy of the production database in development or connect to some production services for which there's no development instance, for example. If a project uses a package like the `django-debug-toolbar`, then the attacker could execute arbitrary SQL, which could be especially bad if the developers connect to the database with a superuser account.

`settings.ALLOWED_HOSTS` is now validated regardless of `DEBUG`. For convenience, if `ALLOWED_HOSTS` is empty and `DEBUG=True`, the following variations of localhost are allowed `['localhost', '127.0.0.1', '::1']`. If your local settings file has your production `ALLOWED_HOSTS` value, you must now omit it to get those fallback values.

Django 1.9.10 release notes

September 26, 2016

Django 1.9.10 fixes a security issue in 1.9.9.

CSRF protection bypass on a site with Google Analytics

An interaction between Google Analytics and Django's cookie parsing could allow an attacker to set arbitrary cookies leading to a bypass of CSRF protection.

The parser for `request.COOKIES` is simplified to better match the behavior of browsers and to mitigate this attack. `request.COOKIES` may now contain cookies that are invalid according to **RFC 6265** but are possible to set via `document.cookie`.

Django 1.9.9 release notes

August 1, 2016

Django 1.9.9 fixes several bugs in 1.9.8.

Bugfixes

- Fixed invalid HTML in template postmortem on the debug page (#26938).
- Fixed some GIS database function crashes on MySQL 5.7 (#26657).

Django 1.9.8 release notes

July 18, 2016

Django 1.9.8 fixes a security issue and several bugs in 1.9.7.

XSS in admin's add/change related popup

Unsafe usage of JavaScript's `Element.innerHTML` could result in XSS in the admin's add/change related popup. `Element.textContent` is now used to prevent execution of the data.

The debug view also used `innerHTML`. Although a security issue wasn't identified there, out of an abundance of caution it's also updated to use `textContent`.

Bugfixes

- Fixed missing `varchar/text_pattern_ops` index on `CharField` and `TextField` respectively when using `AddField` on PostgreSQL (#26889).
- Fixed `makemessages` crash on Python 2 with non-ASCII file names (#26897).

Django 1.9.7 release notes

June 4, 2016

Django 1.9.7 fixes several bugs in 1.9.6.

Bugfixes

- Removed the need for the `request` context processor on the admin login page to fix a regression in 1.9 (#26558).
- Fixed translation of password validators' `help_text` in forms (#26544).
- Fixed a regression causing the cached template loader to crash when using lazy template names (#26603).
- Fixed `on_commit` callbacks execution order when callbacks make transactions (#26627).
- Fixed `HStoreField` to raise a `ValidationError` instead of crashing on non-dictionary JSON input (#26672).
- Fixed `dbshell` crash on PostgreSQL with an empty database name (#26698).

- Fixed a regression in queries on a `OneToOneField` that has `to_field` and `primary_key=True` (#26667).

Django 1.9.6 release notes

May 2, 2016

Django 1.9.6 fixes several bugs in 1.9.5.

Bugfixes

- Added support for relative path redirects to the test client and to `SimpleTestCase.assertRedirects()` because Django 1.9 no longer converts redirects to absolute URIs (#26428).
- Fixed `TimeField` microseconds round-tripping on MySQL and SQLite (#26498).
- Prevented `makemigrations` from generating infinite migrations for a model field that references a `functools.partial` (#26475).
- Fixed a regression where `SessionBase.pop()` returned `None` rather than raising a `KeyError` for nonexistent values (#26520).
- Fixed a regression causing the cached template loader to crash when using template names starting with a dash (#26536).
- Restored conversion of an empty string to null when saving values of `GenericIPAddressField` on SQLite and MySQL (#26557).
- Fixed a `makemessages` regression where temporary `.py` extensions were leaked in source file paths (#26341).

Django 1.9.5 release notes

April 1, 2016

Django 1.9.5 fixes several bugs in 1.9.4.

Bugfixes

- Made `MultiPartParser` ignore filenames that normalize to an empty string to fix crash in `MemoryFileUploadHandler` on specially crafted user input (#26325).
- Fixed a race condition in `BaseCache.get_or_set()` (#26332). It now returns the `default` value instead of `False` if there's an error when trying to add the value to the cache.
- Fixed data loss on SQLite where `DurationField` values with fractional seconds could be saved as `None` (#26324).
- The forms in `contrib.auth` no longer strip trailing and leading whitespace from the password fields (#26334). The change requires users who set their password to something with such whitespace after a site updated to Django 1.9 to reset their password. It provides backwards-compatibility for earlier versions of Django.
- Fixed a memory leak in the cached template loader (#26306).
- Fixed a regression that caused `collectstatic --clear` to fail if the storage doesn't implement `path()` (#26297).
- Fixed a crash when using a reverse lookup with a subquery when a `ForeignKey` has a `to_field` set to something other than the primary key (#26373).

- Fixed a regression in `CommonMiddleware` that caused spurious warnings in logs on requests missing a trailing slash (#26293).
- Restored the functionality of the admin's `raw_id_fields` in `list_editable` (#26387).
- Fixed a regression with abstract model inheritance and explicit parent links (#26413).
- Fixed a migrations crash on SQLite when renaming the primary key of a model containing a `ForeignKey` to `'self'` (#26384).
- Fixed `JSONField` inadvertently escaping its contents when displaying values after failed form validation (#25532).

Django 1.9.4 release notes

March 5, 2016

Django 1.9.4 fixes a regression on Python 2 in the 1.9.3 security release where `utils.http.is_safe_url()` crashes on bytestring URLs (#26308).

Django 1.9.3 release notes

March 1, 2016

Django 1.9.3 fixes two security issues and several bugs in 1.9.2.

CVE-2016-2512: Malicious redirect and possible XSS attack via user-supplied redirect URLs containing basic auth

Django relies on user input in some cases (e.g. *`django.contrib.auth.views.login()`* and i18n) to redirect the user to an "on success" URL. The security check for these redirects (namely `django.utils.http.is_safe_url()`) considered some URLs with basic authentication credentials "safe" when they shouldn't be.

For example, a URL like `http://mysite.example.com\@attacker.com` would be considered safe if the request's host is `http://mysite.example.com`, but redirecting to this URL sends the user to `attacker.com`.

Also, if a developer relies on `is_safe_url()` to provide safe redirect targets and puts such a URL into a link, they could suffer from an XSS attack.

CVE-2016-2513: User enumeration through timing difference on password hasher work factor upgrade

In each major version of Django since 1.6, the default number of iterations for the `PBKDF2PasswordHasher` and its subclasses has increased. This improves the security of the password as the speed of hardware increases, however, it also creates a timing difference between a login request for a user with a password encoded in an older number of iterations and login request for a nonexistent user (which runs the default hasher's default number of iterations since Django 1.6).

This only affects users who haven't logged in since the iterations were increased. The first time a user logs in after an iterations increase, their password is updated with the new iterations and there is no longer a timing difference.

The new `BasePasswordHasher.harden_runtime()` method allows hashers to bridge the runtime gap between the work factor (e.g. iterations) supplied in existing encoded passwords and the default work factor of the hasher. This method is implemented for `PBKDF2PasswordHasher` and `BCryptPasswordHasher`. The number of rounds for the latter hasher hasn't changed since Django 1.4, but some projects may subclass it and increase the work factor as needed.

A warning will be emitted for any *third-party password hashers that don't implement* a `harden_runtime()` method.

If you have different password hashes in your database (such as SHA1 hashes from users who haven't logged in since the default hasher switched to PBKDF2 in Django 1.4), the timing difference on a login request for these users may be even greater and this fix doesn't remedy that difference (or any difference when changing hashers). You may be able to *upgrade those hashes* to prevent a timing attack for that case.

Bugfixes

- Skipped URL checks (new in 1.9) if the `ROOT_URLCONF` setting isn't defined (#26155).
- Fixed a crash on PostgreSQL that prevented using `TIME_ZONE=None` and `USE_TZ=False` (#26177).
- Added system checks for query name clashes of hidden relationships (#26162).
- Fixed a regression for cases where `ForeignObject.get_extra_descriptor_filter()` returned a `Q` object (#26153).
- Fixed regression with an `__in=qs` lookup for a `ForeignKey` with `to_field` set (#26196).
- Made `forms.FileField` and `utils.translation.lazy_number()` picklable (#26212).
- Fixed *`RangeField`* and *`ArrayField`* serialization with `None` values (#26215).
- Fixed a crash when filtering by a `Decimal` in `RawQuery` (#26219).
- Reallowed dashes in top-level domain names of URLs checked by `URLValidator` to fix a regression in Django 1.8 (#26204).
- Fixed some crashing deprecation shims in `SimpleTemplateResponse` that regressed in Django 1.9 (#26253).
- Fixed `BoundField` to reallow slices of subwidgets (#26267).
- Changed the admin's "permission denied" message in the login template to use `get_username` instead of `username` to support custom user models (#26231).
- Fixed a crash when passing a nonexistent template name to the cached template loader's `load_template()` method (#26280).
- Prevented `ContentTypeManager` instances from sharing their cache (#26286).
- Reverted a change in Django 1.9.2 (#25858) that prevented relative lazy relationships defined on abstract models to be resolved according to their concrete model's `app_label` (#26186).

Django 1.9.2 release notes

February 1, 2016

Django 1.9.2 fixes a security regression in 1.9 and several bugs in 1.9.1. It also makes a small backwards incompatible change that hopefully doesn't affect any users.

Security issue: User with "change" but not "add" permission can create objects for `ModelAdmin`'s with `save_as=True`

If a `ModelAdmin` uses `save_as=True` (not the default), the admin provides an option when editing objects to "Save as new". A regression in Django 1.9 prevented that form submission from raising a "Permission Denied" error for users without the "add" permission.

Backwards incompatible change: `.py-tpl` files rewritten in project/app templates

The addition of some Django template language syntax to the default app template in Django 1.9 means those files now have some invalid Python syntax. This causes difficulties for packaging systems that unconditionally byte-compile `*.py` files.

To remedy this, a `.py-tpl` suffix is now used for the project and app template files included in Django. The `.py-tpl` suffix is replaced with `.py` by the `startproject` and `startapp` commands. For example, a template with the filename `manage.py-tpl` will be created as `manage.py`.

Please file a ticket if you have a custom project template containing `.py-tpl` files and find this behavior problematic.

Bugfixes

- Fixed a regression in `ConditionalGetMiddleware` causing `If-None-Match` checks to always return HTTP 200 (#26024).
- Fixed a regression that caused the "user-tools" items to display on the admin's logout page (#26035).
- Fixed a crash in the translations system when the current language has no translations (#26046).
- Fixed a regression that caused the incorrect day to be selected when opening the admin calendar widget for timezones from GMT+0100 to GMT+1200 (#24980).
- Fixed a regression in the admin's edit related model popup that caused an escaped value to be displayed in the select dropdown of the parent window (#25997).
- Fixed a regression in 1.8.8 causing incorrect index handling in migrations on PostgreSQL when adding `db_index=True` or `unique=True` to a `CharField` or `TextField` that already had the other specified, or when removing one of them from a field that had both, or when adding `unique=True` to a field already listed in `unique_together` (#26034).
- Fixed a regression where defining a relation on an abstract model's field using a string model name without an app_label no longer resolved that reference to the abstract model's app if using that model in another application (#25858).
- Fixed a crash when destroying an existing test database on MySQL or PostgreSQL (#26096).
- Fixed CSRF cookie check on POST requests when `USE_X_FORWARDED_PORT=True` (#26094).
- Fixed a `QuerySet.order_by()` crash when ordering by a relational field of a `ManyToManyField through` model (#26092).
- Fixed a regression that caused an exception when making database queries on SQLite with more than 2000 parameters when *`DEBUG`* is `True` on distributions that increase the `SQLITE_MAX_VARIABLE_NUMBER` compile-time limit to over 2000, such as Debian (#26063).
- Fixed a crash when using a reverse `OneToOneField` in `ModelAdmin.readonly_fields` (#26060).
- Fixed a crash when calling the `migrate` command in a test case with the `available_apps` attribute pointing to an application with migrations disabled using the `MIGRATION_MODULES` setting (#26135).
- Restored the ability for testing and debugging tools to determine the template from which a node came from, even during template inheritance or inclusion. Prior to Django 1.9, debugging tools could access the template origin from the node via `Node.token.source[0]`. This was an undocumented, private API. The origin is now available directly on each node using the `Node.origin` attribute (#25848).
- Fixed a regression in Django 1.8.5 that broke copying a `SimpleLazyObject` with `copy.copy()` (#26122).
- Always included `geometry_field` in the GeoJSON serializer output regardless of the `fields` parameter (#26138).

- Fixed the `contrib.gis` map widgets when using `USE_THOUSAND_SEPARATOR=True` (#20415).
- Made invalid forms display the initial of values of their disabled fields (#26129).

Django 1.9.1 release notes

January 2, 2016

Django 1.9.1 fixes several bugs in 1.9.

Bugfixes

- Fixed `BaseCache.get_or_set()` with the `DummyCache` backend (#25840).
- Fixed a regression in `FormMixin` causing forms to be validated twice (#25548, #26018).
- Fixed a system check crash with nested `ArrayFields` (#25867).
- Fixed a state bug when migrating a `SeparateDatabaseAndState` operation backwards (#25896).
- Fixed a regression in `CommonMiddleware` causing `If-None-Match` checks to always return HTTP 200 (#25900).
- Fixed missing `varchar/text_pattern_ops` index on `CharField` and `TextField` respectively when using `AlterField` on PostgreSQL (#25412).
- Fixed admin's delete confirmation page's summary counts of related objects (#25883).
- Added `from __future__ import unicode_literals` to the default `apps.py` created by `startapp` on Python 2 (#25909). Add this line to your own `apps.py` files created using Django 1.9 if you want your migrations to work on both Python 2 and Python 3.
- Prevented `QuerySet.delete()` from crashing on MySQL when querying across relations (#25882).
- Fixed evaluation of zero-length slices of `QuerySet.values()` (#25894).
- Fixed a state bug when using an `AlterModelManagers` operation (#25852).
- Fixed `TypedChoiceField` change detection with nullable fields (#25942).
- Fixed incorrect timezone warnings in custom admin templates that don't have a `data-admin-utc-offset` attribute in the `body` tag. (#25845).
- Fixed a regression which prevented using a language not in Django's default language list (*`LANGUAGES`*) (#25915).
- Avoided hiding some exceptions, like an invalid `INSTALLED_APPS` setting, behind `AppRegistryNotReady` when starting `runserver` (#25510). This regression appeared in 1.8.5 as a side effect of fixing #24704 and by mistake the fix wasn't applied to the `stable/1.9.x` branch.
- Fixed `migrate --fake-initial` detection of many-to-many tables (#25922).
- Restored the functionality of the admin's `list_editable` add and change buttons (#25903).
- Fixed `isnull` query lookup for `ForeignObject` (#25972).
- Fixed a regression in the admin which ignored line breaks in read-only fields instead of converting them to `<br>` (#25465).
- Fixed incorrect object reference in `SingleObjectMixin.get_context_object_name()` (#26006).
- Made `loaddata` skip disabling and enabling database constraints when it doesn't load any fixtures (#23372).
- Restored `contrib.auth` hashers compatibility with py-bcrypt (#26016).

- Fixed a crash in `QuerySet.values()/values_list()` after an `annotate()` and `order_by()` when `values()/values_list()` includes a field not in the `order_by()` (#25316).

Django 1.9 release notes

December 1, 2015

Welcome to Django 1.9!

These release notes cover the *new features*, as well as some *backwards incompatible changes* you'll want to be aware of when upgrading from Django 1.8 or older versions. We've *dropped some features* that have reached the end of their deprecation cycle, and we've *begun the deprecation process for some features*.

See the Upgrading Django to a newer version guide if you're updating an existing project.

Python compatibility

Django 1.9 requires Python 2.7, 3.4, or 3.5. We **highly recommend** and only officially support the latest release of each series.

The Django 1.8 series is the last to support Python 3.2 and 3.3.

What's new in Django 1.9

Performing actions after a transaction commit The new *`on_commit()`* hook allows performing actions after a database transaction is successfully committed. This is useful for tasks such as sending notification emails, creating queued tasks, or invalidating caches.

This functionality from the django-transaction-hooks package has been integrated into Django.

Password validation Django now offers password validation to help prevent the usage of weak passwords by users. The validation is integrated in the included password change and reset forms and is simple to integrate in any other code. Validation is performed by one or more validators, configured in the new *`AUTH_PASSWORD_VALIDATORS`* setting.

Four validators are included in Django, which can enforce a minimum length, compare the password to the user's attributes like their name, ensure passwords aren't entirely numeric, or check against an included list of common passwords. You can combine multiple validators, and some validators have custom configuration options. For example, you can choose to provide a custom list of common passwords. Each validator provides a help text to explain its requirements to the user.

By default, no validation is performed and all passwords are accepted, so if you don't set *`AUTH_PASSWORD_VALIDATORS`*, you will not see any change. In new projects created with the default *`startproject`* template, a simple set of validators is enabled. To enable basic validation in the included auth forms for your project, you could set, for example:

```
AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME': 'django.contrib.auth.password_validation.UserAttributeSimilarityValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.CommonPasswordValidator',
```

```
    },
    {
        'NAME': 'django.contrib.auth.password_validation.NumericPasswordValidator',
    },
]
```

See *Password validation* for more details.

Permission mixins for class-based views Django now ships with the mixins *AccessMixin*, *LoginRequiredMixin*, *PermissionRequiredMixin*, and *UserPassesTestMixin* to provide the functionality of the `django.contrib.auth.decorators` for class-based views. These mixins have been taken from, or are at least inspired by, the django-braces project.

There are a few differences between Django's and django-braces' implementation, though:

- The *raise_exception* attribute can only be `True` or `False`. Custom exceptions or callables are not supported.
- The *handle_no_permission()* method does not take a `request` argument. The current request is available in `self.request`.
- The custom `test_func()` of *UserPassesTestMixin* does not take a `user` argument. The current user is available in `self.request.user`.
- The *permission_required* attribute supports a string (defining one permission) or a list/tuple of strings (defining multiple permissions) that need to be fulfilled to grant access.
- The new *permission_denied_message* attribute allows passing a message to the `PermissionDenied` exception.

New styling for `contrib.admin` The admin sports a modern, flat design with new SVG icons which look perfect on HiDPI screens. It still provides a fully-functional experience to YUI's A-grade browsers. Older browser may experience varying levels of graceful degradation.

Running tests in parallel The *test* command now supports a *--parallel* option to run a project's tests in multiple processes in parallel.

Each process gets its own database. You must ensure that different test cases don't access the same resources. For instance, test cases that touch the filesystem should create a temporary directory for their own use.

This option is enabled by default for Django's own test suite provided:

- the OS supports it (all but Windows)
- the database backend supports it (all the built-in backends but Oracle)

Minor features

`django.contrib.admin`

- Admin views now have `model_admin` or `admin_site` attributes.
- The URL of the admin change view has been changed (was at `/admin/<app>/<model>/<pk>/` by default and is now at `/admin/<app>/<model>/<pk>/change/`). This should not affect your application unless you have hardcoded admin URLs. In that case, replace those links by *reversing admin URLs* instead. Note that the old URL still redirects to the new one for backwards compatibility, but it may be removed in a future version.

- *ModelAdmin.get_list_select_related()* was added to allow changing the `select_related()` values used in the admin's changelist query based on the request.
- The `available_apps` context variable, which lists the available applications for the current user, has been added to the *AdminSite.each_context()* method.
- *AdminSite.empty_value_display* and *ModelAdmin.empty_value_display* were added to override the display of empty values in admin change list. You can also customize the value for each field.
- Added jQuery events *when an inline form is added or removed* on the change form page.
- The time picker widget includes a '6 p.m' option for consistency of having predefined options every 6 hours.
- JavaScript slug generation now supports Romanian characters.

django.contrib.admindocs

- The model section of the `admindocs` now also describes methods that take arguments, rather than ignoring them.

django.contrib.auth

- The default iteration count for the PBKDF2 password hasher has been increased by 20%. This backwards compatible change will not affect users who have subclassed `django.contrib.auth.hashers.PBKDF2PasswordHasher` to change the default value.
- The `BCryptSHA256PasswordHasher` will now update passwords if its `rounds` attribute is changed.
- `AbstractBaseUser` and `BaseUserManager` were moved to a new `django.contrib.auth.base_user` module so that they can be imported without including `django.contrib.auth` in *INSTALLED_APPS* (doing so raised a deprecation warning in older versions and is no longer supported in Django 1.9).
- The permission argument of *permission_required()* accepts all kinds of iterables, not only list and tuples.
- The new *PersistentRemoteUserMiddleware* makes it possible to use `REMOTE_USER` for setups where the header is only populated on login pages instead of every request in the session.
- The *password_reset()* view accepts an `extra_email_context` parameter.

django.contrib.contenttypes

- It's now possible to use *order_with_respect_to* with a `GenericForeignKey`.

django.contrib.gis

- All `GeoQuerySet` methods have been deprecated and replaced by equivalent database functions. As soon as the legacy methods have been replaced in your code, you should even be able to remove the special `GeoManager` from your GIS-enabled classes.
- The GDAL interface now supports instantiating file-based and in-memory *GDALRaster objects* from raw data. Setters for raster properties such as projection or pixel values have been added.
- For PostGIS users, the new *RasterField* allows *storing GDALRaster objects*. It supports automatic spatial index creation and reprojection when saving a model. It does not yet support spatial querying.
- The new *GDALRaster.warp()* method allows warping a raster by specifying target raster properties such as origin, width, height, or pixel size (amongst others).

- The new *GDALRaster.transform()* method allows transforming a raster into a different spatial reference system by specifying a target `srid`.
- The new *GeoIP2* class allows using MaxMind's GeoLite2 databases which includes support for IPv6 addresses.
- The default OpenLayers library version included in widgets has been updated from 2.13 to 2.13.1.

`django.contrib.postgres`

- Added support for the *rangefield.contained_by* lookup for some built in fields which correspond to the range fields.
- Added *JSONField*.
- Added PostgreSQL specific aggregation functions.
- Added the *TransactionNow* database function.

`django.contrib.sessions`

- The session model and `SessionStore` classes for the `db` and `cached_db` backends are refactored to allow a custom database session backend to build upon them. See *Extending database-backed session engines* for more details.

`django.contrib.sites`

- *get_current_site()* now handles the case where `request.get_host()` returns `domain:port`, e.g. `example.com:80`. If the lookup fails because the host does not match a record in the database and the host has a port, the port is stripped and the lookup is retried with the domain part only.

`django.contrib.syndication`

- Support for multiple enclosures per feed item has been added. If multiple enclosures are defined on a RSS feed, an exception is raised as RSS feeds, unlike Atom feeds, do not support multiple enclosures per feed item.

Cache

- `django.core.cache.backends.base.BaseCache` now has a `get_or_set()` method.
- *django.views.decorators.cache.never_cache()* now sends more persuasive headers (added `no-cache, no-store, must-revalidate` to `Cache-Control`) to better prevent caching. This was also added in Django 1.8.8.

CSRF

- The request header's name used for CSRF authentication can be customized with *CSRF_HEADER_NAME*.
- The CSRF referer header is now validated against the *CSRF_COOKIE_DOMAIN* setting if set. See *How it works* for details.
- The new *CSRF_TRUSTED_ORIGINS* setting provides a way to allow cross-origin unsafe requests (e.g. `POST`) over HTTPS.

Database backends

- The PostgreSQL backend (`django.db.backends.postgresql_psycopg2`) is also available as `django.db.backends.postgresql`. The old name will continue to be available for backwards compatibility.

File Storage

- *`Storage.get_valid_name()`* is now called when the *`upload_to`* is a callable.
- *`File`* now has the `seekable()` method when using Python 3.

Forms

- *`ModelForm`* accepts the new `Meta` option `field_classes` to customize the type of the fields. See *Overriding the default fields* for details.
- You can now specify the order in which form fields are rendered with the *`field_order`* attribute, the `field_order` constructor argument , or the *`order_fields()`* method.
- A form prefix can be specified inside a form class, not only when instantiating a form. See *Prefixes for forms* for details.
- You can now *specify keyword arguments* that you want to pass to the constructor of forms in a formset.
- *`SlugField`* now accepts an *`allow_unicode`* argument to allow Unicode characters in slugs.
- *`CharField`* now accepts a *`strip`* argument to strip input data of leading and trailing whitespace. As this defaults to `True` this is different behavior from previous releases.
- Form fields now support the *`disabled`* argument, allowing the field widget to be displayed disabled by browsers.
- It's now possible to customize bound fields by overriding a field's *`get_bound_field()`* method.

Generic Views

- Class-based views generated using `as_view()` now have `view_class` and `view_initkwargs` attributes.
- *`method_decorator()`* can now be used with a list or tuple of decorators. It can also be used to *decorate classes instead of methods*.

Internationalization

- The *`django.views.i18n.set_language()`* view now properly redirects to *translated URLs*, when available.
- The *`django.views.i18n.javascript_catalog()`* view now works correctly if used multiple times with different configurations on the same page.
- The *`django.utils.timezone.make_aware()`* function gained an `is_dst` argument to help resolve ambiguous times during DST transitions.
- You can now use locale variants supported by gettext. These are usually used for languages which can be written in different scripts, for example Latin and Cyrillic (e.g. `be@latin`).
- Added the *`django.views.i18n.json_catalog()`* view to help build a custom client-side i18n library upon Django translations. It returns a JSON object containing a translations catalog, formatting settings, and a plural rule.

- Added the `name_translated` attribute to the object returned by the *`get_language_info`* template tag. Also added a corresponding template filter: *`language_name_translated`*.
- You can now run *`compilemessages`* from the root directory of your project and it will find all the app message files that were created by *`makemessages`*.
- *`makemessages`* now calls xgettext once per locale directory rather than once per translatable file. This speeds up localization builds.
- *`blocktrans`* supports assigning its output to a variable using `asvar`.
- Two new languages are available: Colombian Spanish and Scottish Gaelic.

Management Commands

- The new *`sendtestemail`* command lets you send a test email to easily confirm that email sending through Django is working.
- To increase the readability of the SQL code generated by *`sqlmigrate`*, the SQL code generated for each migration operation is preceded by the operation's description.
- The *`dumpdata`* command output is now deterministically ordered. Moreover, when the `--output` option is specified, it also shows a progress bar in the terminal.
- The *`createcachetable`* command now has a `--dry-run` flag to print out the SQL rather than execute it.
- The *`startapp`* command creates an `apps.py` file. Since it doesn't use `default_app_config` (*a discouraged API*), you must specify the app config's path, e.g. `'polls.apps.PollsConfig'`, in *`INSTALLED_APPS`* for it to be used (instead of just `'polls'`).
- When using the PostgreSQL backend, the *`dbshell`* command can connect to the database using the password from your settings file (instead of requiring it to be manually entered).
- The `django` package may be run as a script, i.e. `python -m django`, which will behave the same as `django-admin`.
- Management commands that have the `--noinput` option now also take `--no-input` as an alias for that option.

Migrations

- Initial migrations are now marked with an *`initial = True`* class attribute which allows *`migrate --fake-initial`* to more easily detect initial migrations.
- Added support for serialization of `functools.partial` and `LazyObject` instances.
- When supplying `None` as a value in *`MIGRATION_MODULES`*, Django will consider the app an app without migrations.
- When applying migrations, the "Rendering model states" step that's displayed when running migrate with verbosity 2 or higher now computes only the states for the migrations that have already been applied. The model states for migrations being applied are generated on demand, drastically reducing the amount of required memory.

  However, this improvement is not available when unapplying migrations and therefore still requires the precomputation and storage of the intermediate migration states.

  This improvement also requires that Django no longer supports mixed migration plans. Mixed plans consist of a list of migrations where some are being applied and others are being unapplied. This was never officially supported and never had a public API that supports this behavior.

- The *squashmigrations* command now supports specifying the starting migration from which migrations will be squashed.

Models

- *QuerySet.bulk_create()* now works on proxy models.
- Database configuration gained a *TIME_ZONE* option for interacting with databases that store datetimes in local time and don't support time zones when *USE_TZ* is `True`.
- Added the *RelatedManager.set()* method to the related managers created by `ForeignKey`, `GenericForeignKey`, and `ManyToManyField`.
- The *add()* method on a reverse foreign key now has a `bulk` parameter to allow executing one query regardless of the number of objects being added rather than one query per object.
- Added the `keep_parents` parameter to *Model.delete()* to allow deleting only a child's data in a model that uses multi-table inheritance.
- *Model.delete()* and *QuerySet.delete()* return the number of objects deleted.
- Added a system check to prevent defining both `Meta.ordering` and `order_with_respect_to` on the same model.
- *Date and time* lookups can be chained with other lookups (such as *exact*, *gt*, *lt*, etc.). For example: `Entry.objects.filter(pub_date__month__gt=6)`.
- Time lookups (hour, minute, second) are now supported by *TimeField* for all database backends. Support for backends other than SQLite was added but undocumented in Django 1.7.
- You can specify the `output_field` parameter of the *Avg* aggregate in order to aggregate over non-numeric columns, such as `DurationField`.
- Added the *date* lookup to *DateTimeField* to allow querying the field by only the date portion.
- Added the *Greatest* and *Least* database functions.
- Added the *Now* database function, which returns the current date and time.
- *Transform* is now a subclass of *Func()* which allows `Transforms` to be used on the right hand side of an expression, just like regular `Funcs`. This allows registering some database functions like *Length*, *Lower*, and *Upper* as transforms.
- *SlugField* now accepts an *allow_unicode* argument to allow Unicode characters in slugs.
- Added support for referencing annotations in `QuerySet.distinct()`.
- `connection.queries` shows queries with substituted parameters on SQLite.
- Query expressions can now be used when creating new model instances using `save()`, `create()`, and `bulk_create()`.

Requests and Responses

- Unless *HttpResponse.reason_phrase* is explicitly set, it now is determined by the current value of *HttpResponse.status_code*. Modifying the value of `status_code` outside of the constructor will also modify the value of `reason_phrase`.
- The debug view now shows details of chained exceptions on Python 3.
- The default 40x error views now accept a second positional parameter, the exception that triggered the view.
- View error handlers now support *TemplateResponse*, commonly used with class-based views.

- Exceptions raised by the `render()` method are now passed to the `process_exception()` method of each middleware.
- Request middleware can now set *`HttpRequest.urlconf`* to `None` to revert any changes made by previous middleware and return to using the *`ROOT_URLCONF`*.
- The *`DISALLOWED_USER_AGENTS`* check in *`CommonMiddleware`* now raises a *`PermissionDenied`* exception as opposed to returning an *`HttpResponseForbidden`* so that *`handler403`* is invoked.
- Added *`HttpRequest.get_port()`* to fetch the originating port of the request.
- Added the `json_dumps_params` parameter to *`JsonResponse`* to allow passing keyword arguments to the `json.dumps()` call used to generate the response.
- The *`BrokenLinkEmailsMiddleware`* now ignores 404s when the referer is equal to the requested URL. To circumvent the empty referer check already implemented, some Web bots set the referer to the requested URL.

Templates

- Template tags created with the *`simple_tag()`* helper can now store results in a template variable by using the `as` argument.
- Added a *`Context.setdefault()`* method.
- The *django.template* logger was added and includes the following messages:
  - A `DEBUG` level message for missing context variables.
  - A `WARNING` level message for uncaught exceptions raised during the rendering of an `{% include %}` when debug mode is off (helpful since `{% include %}` silences the exception and returns an empty string).
- The *`firstof`* template tag supports storing the output in a variable using 'as'.
- *`Context.update()`* can now be used as a context manager.
- Django template loaders can now extend templates recursively.
- The debug page template postmortem now include output from each engine that is installed.
- *Debug page integration* for custom template engines was added.
- The *`DjangoTemplates`* backend gained the ability to register libraries and builtins explicitly through the template *`OPTIONS`*.
- The `timesince` and `timeuntil` filters were improved to deal with leap years when given large time spans.
- The `include` tag now caches parsed templates objects during template rendering, speeding up reuse in places such as for loops.

Tests

- Added the *`json()`* method to test client responses to give access to the response body as JSON.
- Added the *`force_login()`* method to the test client. Use this method to simulate the effect of a user logging into the site while skipping the authentication and verification steps of *`login()`*.

URLs

- Regular expression lookaround assertions are now allowed in URL patterns.
- The application namespace can now be set using an `app_name` attribute on the included module or object. It can also be set by passing a 2-tuple of (<list of patterns>, <application namespace>) as the first argument to *`include()`*.
- System checks have been added for common URL pattern mistakes.

Validators

- Added *`django.core.validators.int_list_validator()`* to generate validators of strings containing integers separated with a custom character.
- *`EmailValidator`* now limits the length of domain name labels to 63 characters per **RFC 1034**.
- Added *`validate_unicode_slug()`* to validate slugs that may contain Unicode characters.

Backwards incompatible changes in 1.9

: In addition to the changes outlined in this section, be sure to review the *Features removed in 1.9* for the features that have reached the end of their deprecation cycle and therefore been removed. If you haven't updated your code within the deprecation timeline for a given feature, its removal may appear as a backwards incompatible change.

Database backend API

- A couple of new tests rely on the ability of the backend to introspect column defaults (returning the result as `Field.default`). You can set the `can_introspect_default` database feature to `False` if your backend doesn't implement this. You may want to review the implementation on the backends that Django includes for reference (#24245).
- Registering a global adapter or converter at the level of the DB-API module to handle time zone information of `datetime` values passed as query parameters or returned as query results on databases that don't support time zones is discouraged. It can conflict with other libraries.

  The recommended way to add a time zone to `datetime` values fetched from the database is to register a converter for `DateTimeField` in `DatabaseOperations.get_db_converters()`.

  The `needs_datetime_string_cast` database feature was removed. Database backends that set it must register a converter instead, as explained above.
- The `DatabaseOperations.value_to_db_<type>()` methods were renamed to `adapt_<type>field_value()` to mirror the `convert_<type>field_value()` methods.
- To use the new `date` lookup, third-party database backends may need to implement the `DatabaseOperations.datetime_cast_date_sql()` method.
- The `DatabaseOperations.time_extract_sql()` method was added. It calls the existing `date_extract_sql()` method. This method is overridden by the SQLite backend to add time lookups (hour, minute, second) to *`TimeField`*, and may be needed by third-party database backends.
- The `DatabaseOperations.datetime_cast_sql()` method (not to be confused with `DatabaseOperations.datetime_cast_date_sql()` mentioned above) has been removed. This method served to format dates on Oracle long before 1.0, but hasn't been overridden by any core backend in years and hasn't been called anywhere in Django's code or tests.

- In order to support test parallelization, you must implement the `DatabaseCreation._clone_test_db()` method and set `DatabaseFeatures.can_clone_databases = True`. You may have to adjust `DatabaseCreation.get_test_db_clone_settings()`.

Default settings that were tuples are now lists The default settings in `django.conf.global_settings` were a combination of lists and tuples. All settings that were formerly tuples are now lists.

`is_usable` attribute on template loaders is removed Django template loaders previously required an `is_usable` attribute to be defined. If a loader was configured in the template settings and this attribute was `False`, the loader would be silently ignored. In practice, this was only used by the egg loader to detect if setuptools was installed. The `is_usable` attribute is now removed and the egg loader instead fails at runtime if setuptools is not installed.

Related set direct assignment Direct assignment of related objects in the ORM used to perform a `clear()` followed by a call to `add()`. This caused needlessly large data changes and prevented using the *`m2m_changed`* signal to track individual changes in many-to-many relations.

Direct assignment now relies on the the new *`set()`* method on related managers which by default only processes changes between the existing related set and the one that's newly assigned. The previous behavior can be restored by replacing direct assignment by a call to `set()` with the keyword argument `clear=True`.

`ModelForm`, and therefore `ModelAdmin`, internally rely on direct assignment for many-to-many relations and as a consequence now use the new behavior.

Filesystem-based template loaders catch more specific exceptions When using the *`filesystem.Loader`* or *`app_directories.Loader`* template loaders, earlier versions of Django raised a *`TemplateDoesNotExist`* error if a template source existed but was unreadable. This could happen under many circumstances, such as if Django didn't have permissions to open the file, or if the template source was a directory. Now, Django only silences the exception if the template source does not exist. All other situations result in the original `IOError` being raised.

HTTP redirects no longer forced to absolute URIs Relative redirects are no longer converted to absolute URIs. **RFC 2616** required the `Location` header in redirect responses to be an absolute URI, but it has been superseded by **RFC 7231** which allows relative URIs in `Location`, recognizing the actual practice of user agents, almost all of which support them.

Consequently, the expected URLs passed to `assertRedirects` should generally no longer include the scheme and domain part of the URLs. For example, `self.assertRedirects(response, 'http://testserver/some-url/')` should be replaced by `self.assertRedirects(response, '/some-url/')` (unless the redirection specifically contained an absolute URL, of course).

In the rare case that you need the old behavior (discovered with an ancient version of Apache with `mod_scgi` that interprets a relative redirect as an "internal redirect"), you can restore it by writing a custom middleware:

```
class LocationHeaderFix(object):
    def process_response(self, request, response):
        if 'Location' in response:
            response['Location'] = request.build_absolute_uri(response['Location'])
        return response
```

Dropped support for PostgreSQL 9.0 Upstream support for PostgreSQL 9.0 ended in September 2015. As a consequence, Django 1.9 sets 9.1 as the minimum PostgreSQL version it officially supports.

Dropped support for Oracle 11.1 Upstream support for Oracle 11.1 ended in August 2015. As a consequence, Django 1.9 sets 11.2 as the minimum Oracle version it officially supports.

Bulk behavior of `add()` method of related managers To improve performance, the `add()` methods of the related managers created by `ForeignKey` and `GenericForeignKey` changed from a series of `Model.save()` calls to a single `QuerySet.update()` call. The change means that `pre_save` and `post_save` signals aren't sent anymore. You can use the `bulk=False` keyword argument to revert to the previous behavior.

Template `LoaderOrigin` and `StringOrigin` are removed In previous versions of Django, when a template engine was initialized with debug as `True`, an instance of `django.template.loader.LoaderOrigin` or `django.template.base.StringOrigin` was set as the origin attribute on the template object. These classes have been combined into *Origin* and is now always set regardless of the engine debug setting. For a minimal level of backwards compatibility, the old class names will be kept as aliases to the new `Origin` class until Django 2.0.

Changes to the default logging configuration To make it easier to write custom logging configurations, Django's default logging configuration no longer defines 'django.request' and 'django.security' loggers. Instead, it defines a single 'django' logger, filtered at the `INFO` level, with two handlers:

- 'console': filtered at the `INFO` level and only active if `DEBUG=True`.
- 'mail_admins': filtered at the `ERROR` level and only active if `DEBUG=False`.

If you aren't overriding Django's default logging, you should see minimal changes in behavior, but you might see some new logging to the `runserver` console, for example.

If you are overriding Django's default logging, you should check to see how your configuration merges with the new defaults.

`HttpRequest` details in error reporting It was redundant to display the full details of the *HttpRequest* each time it appeared as a stack frame variable in the HTML version of the debug page and error email. Thus, the HTTP request will now display the same standard representation as other variables (`repr(request)`). As a result, the `ExceptionReporterFilter.get_request_repr()` method and the undocumented `django.http.build_request_repr()` function were removed.

The contents of the text version of the email were modified to provide a traceback of the same structure as in the case of AJAX requests. The traceback details are rendered by the `ExceptionReporter.get_traceback_text()` method.

Removal of time zone aware global adapters and converters for datetimes Django no longer registers global adapters and converters for managing time zone information on `datetime` values sent to the database as query parameters or read from the database in query results. This change affects projects that meet all the following conditions:

- The *USE_TZ* setting is `True`.
- The database is SQLite, MySQL, Oracle, or a third-party database that doesn't support time zones. In doubt, you can check the value of `connection.features.supports_timezones`.
- The code queries the database outside of the ORM, typically with `cursor.execute(sql, params)`.

If you're passing aware `datetime` parameters to such queries, you should turn them into naive datetimes in UTC:

```
from django.utils import timezone
param = timezone.make_naive(param, timezone.utc)
```

If you fail to do so, the conversion will be performed as in earlier versions (with a deprecation warning) up until Django 1.11. Django 2.0 won't perform any conversion, which may result in data corruption.

If you're reading `datetime` values from the results, they will be naive instead of aware. You can compensate as follows:

```
from django.utils import timezone
value = timezone.make_aware(value, timezone.utc)
```

You don't need any of this if you're querying the database through the ORM, even if you're using *`raw()`* queries. The ORM takes care of managing time zone information.

Template tag modules are imported when templates are configured The *`DjangoTemplates`* backend now performs discovery on installed template tag modules when instantiated. This update enables libraries to be provided explicitly via the `'libraries'` key of *`OPTIONS`* when defining a *`DjangoTemplates`* backend. Import or syntax errors in template tag modules now fail early at instantiation time rather than when a template with a *`{% load %}`* tag is first compiled.

`django.template.base.add_to_builtins()` is removed Although it was a private API, projects commonly used `add_to_builtins()` to make template tags and filters available without using the *`{% load %}`* tag. This API has been formalized. Projects should now define built-in libraries via the `'builtins'` key of *`OPTIONS`* when defining a *`DjangoTemplates`* backend.

`simple_tag` now wraps tag output in `conditional_escape` In general, template tags do not autoescape their contents, and this behavior is *documented*. For tags like *`inclusion_tag`*, this is not a problem because the included template will perform autoescaping. For *`assignment_tag`*, the output will be escaped when it is used as a variable in the template.

For the intended use cases of *`simple_tag`*, however, it is very easy to end up with incorrect HTML and possibly an XSS exploit. For example:

```
@register.simple_tag(takes_context=True)
def greeting(context):
    return "Hello {0}!".format(context['request'].user.first_name)
```

In older versions of Django, this will be an XSS issue because `user.first_name` is not escaped.

In Django 1.9, this is fixed: if the template context has `autoescape=True` set (the default), then `simple_tag` will wrap the output of the tag function with *`conditional_escape()`*.

To fix your `simple_tags`, it is best to apply the following practices:

- Any code that generates HTML should use either the template system or *`format_html()`*.
- If the output of a `simple_tag` needs escaping, use *`escape()`* or *`conditional_escape()`*.
- If you are absolutely certain that you are outputting HTML from a trusted source (e.g. a CMS field that stores HTML entered by admins), you can mark it as such using *`mark_safe()`*.

Tags that follow these rules will be correct and safe whether they are run on Django 1.9+ or earlier.

`Paginator.page_range` *`Paginator.page_range`* is now an iterator instead of a list.

In versions of Django previous to 1.8, `Paginator.page_range` returned a `list` in Python 2 and a `range` in Python 3. Django 1.8 consistently returned a list, but an iterator is more efficient.

Existing code that depends on `list` specific features, such as indexing, can be ported by converting the iterator into a `list` using `list()`.

Implicit `QuerySet` `__in` lookup removed In earlier versions, queries such as:

```
Model.objects.filter(related_id=RelatedModel.objects.all())
```

would implicitly convert to:

```
Model.objects.filter(related_id__in=RelatedModel.objects.all())
```

resulting in SQL like `"related_id IN (SELECT id FROM ...)"`.

This implicit `__in` no longer happens so the "IN" SQL is now "=", and if the subquery returns multiple results, at least some databases will throw an error.

`contrib.admin` browser support The admin no longer supports Internet Explorer 8 and below, as these browsers have reached end-of-life.

CSS and images to support Internet Explorer 6 and 7 have been removed. PNG and GIF icons have been replaced with SVG icons, which are not supported by Internet Explorer 8 and earlier.

The jQuery library embedded in the admin has been upgraded from version 1.11.2 to 2.1.4. jQuery 2.x has the same API as jQuery 1.x, but does not support Internet Explorer 6, 7, or 8, allowing for better performance and a smaller file size. If you need to support IE8 and must also use the latest version of Django, you can override the admin's copy of jQuery with your own by creating a Django application with this structure:

```
app/static/admin/js/vendor/
    jquery.js
    jquery.min.js
```

`SyntaxError` when installing Django setuptools 5.5.x When installing Django 1.9 or 1.9.1 with setuptools 5.5.x, you'll see:

```
Compiling django/conf/app_template/apps.py ...
  File "django/conf/app_template/apps.py", line 4
    class {{ camel_case_app_name }}Config(AppConfig):
          ^
SyntaxError: invalid syntax

Compiling django/conf/app_template/models.py ...
  File "django/conf/app_template/models.py", line 1
    {{ unicode_literals }}from django.db import models
                          ^
SyntaxError: invalid syntax
```

It's safe to ignore these errors (Django will still install just fine), but you can avoid them by upgrading setuptools to a more recent version. If you're using pip, you can upgrade pip using `pip install -U pip` which will also upgrade setuptools. This is resolved in later versions of Django as described in the Django 1.9.2 release notes.

Miscellaneous

- The jQuery static files in `contrib.admin` have been moved into a `vendor/jquery` subdirectory.
- The text displayed for null columns in the admin changelist `list_display` cells has changed from `(None)` (or its translated equivalent) to – (a dash).
- `django.http.responses.REASON_PHRASES` and `django.core.handlers.wsgi.STATUS_CODE_TEXT` have been removed. Use Python's stdlib instead: `http.client.responses` for Python 3 and httplib.responses for Python 2.

- `ValuesQuerySet` and `ValuesListQuerySet` have been removed.
- The `admin/base.html` template no longer sets `window.__admin_media_prefix__` or `window.__admin_utc_offset__`. Image references in JavaScript that used that value to construct absolute URLs have been moved to CSS for easier customization. The UTC offset is stored on a data attribute of the `<body>` tag.
- `CommaSeparatedIntegerField` validation has been refined to forbid values like `',', ',1'`, and `'1„2'`.
- Form initialization was moved from the *`ProcessFormView.get()`* method to the new *`FormMixin.get_context_data()`* method. This may be backwards incompatible if you have overridden the `get_context_data()` method without calling `super()`.
- Support for PostGIS 1.5 has been dropped.
- The `django.contrib.sites.models.Site.domain` field was changed to be *`unique`*.
- In order to enforce test isolation, database queries are not allowed by default in *`SimpleTestCase`* tests anymore. You can disable this behavior by setting the *`allow_database_queries`* class attribute to `True` on your test class.
- `ResolverMatch.app_name` was changed to contain the full namespace path in the case of nested namespaces. For consistency with `ResolverMatch.namespace`, the empty value is now an empty string instead of `None`.
- For security hardening, session keys must be at least 8 characters.
- Private function `django.utils.functional.total_ordering()` has been removed. It contained a workaround for a `functools.total_ordering()` bug in Python versions older than 2.7.3.
- XML serialization (either through *`dumpdata`* or the syndication framework) used to output any characters it received. Now if the content to be serialized contains any control characters not allowed in the XML 1.0 standard, the serialization will fail with a `ValueError`.
- *`CharField`* now strips input of leading and trailing whitespace by default. This can be disabled by setting the new *`strip`* argument to `False`.
- Template text that is translated and uses two or more consecutive percent signs, e.g. `"%%"`, may have a new *msgid* after `makemessages` is run (most likely the translation will be marked fuzzy). The new `msgid` will be marked `"#, python-format"`.
- If neither *`request.current_app`* nor *`Context.current_app`* are set, the *`url`* template tag will now use the namespace of the current request. Set `request.current_app` to `None` if you don't want to use a namespace hint.
- The *`SILENCED_SYSTEM_CHECKS`* setting now silences messages of all levels. Previously, messages of `ERROR` level or higher were printed to the console.
- The `FlatPage.enable_comments` field is removed from the `FlatPageAdmin` as it's unused by the application. If your project or a third-party app makes use of it, *create a custom ModelAdmin* to add it back.
- The return value of *`setup_databases()`* and the first argument of *`teardown_databases()`* changed. They used to be `(old_names, mirrors)` tuples. Now they're just the first item, `old_names`.
- By default *`LiveServerTestCase`* attempts to find an available port in the 8081-8179 range instead of just trying port 8081.
- The system checks for *`ModelAdmin`* now check instances rather than classes.
- The private API to apply mixed migration plans has been dropped for performance reasons. Mixed plans consist of a list of migrations where some are being applied and others are being unapplied.

- The related model object descriptor classes in `django.db.models.fields.related` (private API) are moved from the `related` module to `related_descriptors` and renamed as follows:
  - `ReverseSingleRelatedObjectDescriptor` is `ForwardManyToOneDescriptor`
  - `SingleRelatedObjectDescriptor` is `ReverseOneToOneDescriptor`
  - `ForeignRelatedObjectsDescriptor` is `ReverseManyToOneDescriptor`
  - `ManyRelatedObjectsDescriptor` is `ManyToManyDescriptor`
- If you implement a custom *`handler404`* view, it must return a response with an HTTP 404 status code. Use *`HttpResponseNotFound`* or pass `status=404` to the *`HttpResponse`*. Otherwise, *`APPEND_SLASH`* won't work correctly with `DEBUG=False`.

Features deprecated in 1.9

`assignment_tag()` Django 1.4 added the `assignment_tag` helper to ease the creation of template tags that store results in a template variable. The *`simple_tag()`* helper has gained this same ability, making the `assignment_tag` obsolete. Tags that use `assignment_tag` should be updated to use `simple_tag`.

`{% cycle %}` syntax with comma-separated arguments The *`cycle`* tag supports an inferior old syntax from previous Django versions:

```
{% cycle row1,row2,row3 %}
```

Its parsing caused bugs with the current syntax, so support for the old syntax will be removed in Django 1.10 following an accelerated deprecation.

`ForeignKey` and `OneToOneField` `on_delete` argument In order to increase awareness about cascading model deletion, the `on_delete` argument of `ForeignKey` and `OneToOneField` will be required in Django 2.0.

Update models and existing migrations to explicitly set the argument. Since the default is `models.CASCADE`, add `on_delete=models.CASCADE` to all `ForeignKey` and `OneToOneField`s that don't use a different option. You can also pass it as the second positional argument if you don't care about compatibility with older versions of Django.

`Field.rel` changes `Field.rel` and its methods and attributes have changed to match the related fields API. The `Field.rel` attribute is renamed to `remote_field` and many of its methods and attributes are either changed or renamed.

The aim of these changes is to provide a documented API for relation fields.

`GeoManager` and `GeoQuerySet` custom methods All custom `GeoQuerySet` methods (`area()`, `distance()`, `gml()`, ...) have been replaced by equivalent geographic expressions in annotations (see in new features). Hence the need to set a custom `GeoManager` to GIS-enabled models is now obsolete. As soon as your code doesn't call any of the deprecated methods, you can simply remove the `objects = GeoManager()` lines from your models.

Template loader APIs have changed Django template loaders have been updated to allow recursive template extending. This change necessitated a new template loader API. The old `load_template()` and `load_template_sources()` methods are now deprecated. Details about the new API can be found *in the template loader documentation*.

Passing a 3-tuple or an `app_name` to `include()` The instance namespace part of passing a tuple as an argument to `include()` has been replaced by passing the `namespace` argument to `include()`. For example:

```
polls_patterns = [
     url(...),
]

urlpatterns = [
    url(r'^polls/', include((polls_patterns, 'polls', 'author-polls'))),
]
```

becomes:

```
polls_patterns = ([
     url(...),
], 'polls')  # 'polls' is the app_name

urlpatterns = [
    url(r'^polls/', include(polls_patterns, namespace='author-polls')),
]
```

The `app_name` argument to `include()` has been replaced by passing a 2-tuple (as above), or passing an object or module with an `app_name` attribute (as below). If the `app_name` is set in this new way, the `namespace` argument is no longer required. It will default to the value of `app_name`. For example, the URL patterns in the tutorial are changed from:

`mysite/urls.py`

```
urlpatterns = [
    url(r'^polls/', include('polls.urls', namespace="polls")),
    ...
]
```

to:

`mysite/urls.py`

```
urlpatterns = [
    url(r'^polls/', include('polls.urls')),  # 'namespace="polls"' removed
    ...
]
```

`polls/urls.py`

```
app_name = 'polls'  # added
urlpatterns = [...]
```

This change also means that the old way of including an `AdminSite` instance is deprecated. Instead, pass `admin.site.urls` directly to *`url()`*:

`urls.py`

```
from django.conf.urls import url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', admin.site.urls),
]
```

URL application namespace required if setting an instance namespace In the past, an instance namespace without an application namespace would serve the same purpose as the application namespace, but it was impossible to reverse the patterns if there was an application namespace with the same name. Includes that specify an instance namespace require that the included URLconf sets an application namespace.

`current_app` parameter to `contrib.auth` views All views in `django.contrib.auth.views` have the following structure:

```
def view(request, ..., current_app=None, ...):

    ...

    if current_app is not None:
        request.current_app = current_app

    return TemplateResponse(request, template_name, context)
```

As of Django 1.8, `current_app` is set on the `request` object. For consistency, these views will require the caller to set `current_app` on the `request` instead of passing it in a separate argument.

`django.contrib.gis.geoip` The *`django.contrib.gis.geoip2`* module supersedes `django.contrib.gis.geoip`. The new module provides a similar API except that it doesn't provide the legacy GeoIP-Python API compatibility methods.

Miscellaneous

- The `weak` argument to `django.dispatch.signals.Signal.disconnect()` has been deprecated as it has no effect.
- The `check_aggregate_support()` method of `django.db.backends.base.BaseDatabaseOperations` has been deprecated and will be removed in Django 2.0. The more general `check_expression_support()` should be used instead.
- `django.forms.extras` is deprecated. You can find *`SelectDateWidget`* in `django.forms.widgets` (or simply `django.forms`) instead.
- Private API `django.db.models.fields.add_lazy_relation()` is deprecated.
- The `django.contrib.auth.tests.utils.skipIfCustomUser()` decorator is deprecated. With the test discovery changes in Django 1.6, the tests for `django.contrib` apps are no longer run as part of the user's project. Therefore, the `@skipIfCustomUser` decorator is no longer needed to decorate tests in `django.contrib.auth`.
- If you customized some *error handlers*, the view signatures with only one request parameter are deprecated. The views should now also accept a second `exception` positional parameter.
- The `django.utils.feedgenerator.Atom1Feed.mime_type` and `django.utils.feedgenerator.RssFeed.mime_type` attributes are deprecated in favor of `content_type`.
- *`Signer`* now issues a warning if an invalid separator is used. This will become an exception in Django 1.10.
- `django.db.models.Field._get_val_from_obj()` is deprecated in favor of `Field.value_from_object()`.
- `django.template.loaders.eggs.Loader` is deprecated as distributing applications as eggs is not recommended.

- The `callable_obj` keyword argument to `SimpleTestCase.assertRaisesMessage()` is deprecated. Pass the callable as a positional argument instead.
- The `allow_tags` attribute on methods of `ModelAdmin` has been deprecated. Use *`format_html()`*, *`format_html_join()`*, or *`mark_safe()`* when constructing the method's return value instead.
- The `enclosure` keyword argument to `SyndicationFeed.add_item()` is deprecated. Use the new `enclosures` argument which accepts a list of `Enclosure` objects instead of a single one.
- The `django.template.loader.LoaderOrigin` and `django.template.base.StringOrigin` aliases for `django.template.base.Origin` are deprecated.

Features removed in 1.9

These features have reached the end of their deprecation cycle and are removed in Django 1.9. See *Features deprecated in 1.7* for details, including how to remove usage of these features.

- `django.utils.dictconfig` is removed.
- `django.utils.importlib` is removed.
- `django.utils.tzinfo` is removed.
- `django.utils.unittest` is removed.
- The `syncdb` command is removed.
- `django.db.models.signals.pre_syncdb` and `django.db.models.signals.post_syncdb` is removed.
- Support for `allow_syncdb` on database routers is removed.
- Automatic syncing of apps without migrations is removed. Migrations are compulsory for all apps unless you pass the *`migrate --run-syncdb`* option.
- The SQL management commands for apps without migrations, `sql`, `sqlall`, `sqlclear`, `sqldropindexes`, and `sqlindexes`, are removed.
- Support for automatic loading of `initial_data` fixtures and initial SQL data is removed.
- All models need to be defined inside an installed application or declare an explicit *`app_label`*. Furthermore, it isn't possible to import them before their application is loaded. In particular, it isn't possible to import models inside the root package of an application.
- The model and form `IPAddressField` is removed. A stub field remains for compatibility with historical migrations.
- `AppCommand.handle_app()` is no longer supported.
- `RequestSite` and `get_current_site()` are no longer importable from `django.contrib.sites.models`.
- FastCGI support via the `runfcgi` management command is removed.
- `django.utils.datastructures.SortedDict` is removed.
- `ModelAdmin.declared_fieldsets` is removed.
- The `util` modules that provided backwards compatibility are removed:
  - `django.contrib.admin.util`
  - `django.contrib.gis.db.backends.util`
  - `django.db.backends.util`

  - `django.forms.util`

- `ModelAdmin.get_formsets` is removed.
- The backward compatible shims introduced to rename the `BaseMemcachedCache._get_memcache_timeout()` method to `get_backend_timeout()` is removed.
- The `--natural` and `-n` options for *`dumpdata`* are removed.
- The `use_natural_keys` argument for `serializers.serialize()` is removed.
- Private API `django.forms.forms.get_declared_fields()` is removed.
- The ability to use a `SplitDateTimeWidget` with `DateTimeField` is removed.
- The `WSGIRequest.REQUEST` property is removed.
- The class `django.utils.datastructures.MergeDict` is removed.
- The `zh-cn` and `zh-tw` language codes are removed.
- The internal `django.utils.functional.memoize()` is removed.
- `django.core.cache.get_cache` is removed.
- `django.db.models.loading` is removed.
- Passing callable arguments to querysets is no longer possible.
- `BaseCommand.requires_model_validation` is removed in favor of `requires_system_checks`. Admin validators is replaced by admin checks.
- The `ModelAdmin.validator_class` and `default_validator_class` attributes are removed.
- `ModelAdmin.validate()` is removed.
- `django.db.backends.DatabaseValidation.validate_field` is removed in favor of the `check_field` method.
- The `validate` management command is removed.
- `django.utils.module_loading.import_by_path` is removed in favor of `django.utils.module_loading.import_string`.
- `ssi` and `url` template tags are removed from the `future` template tag library.
- `django.utils.text.javascript_quote()` is removed.
- Database test settings as independent entries in the database settings, prefixed by `TEST_`, are no longer supported.
- The *cache_choices* option to *`ModelChoiceField`* and *`ModelMultipleChoiceField`* is removed.
- The default value of the *`RedirectView.permanent`* attribute has changed from `True` to `False`.
- `django.contrib.sitemaps.FlatPageSitemap` is removed in favor of `django.contrib.flatpages.sitemaps.FlatPageSitemap`.
- Private API `django.test.utils.TestTemplateLoader` is removed.
- The `django.contrib.contenttypes.generic` module is removed.

9.1.3 1.8 release

Django 1.8.16 release notes

November 1, 2016

Django 1.8.16 fixes two security issues in 1.8.15.

User with hardcoded password created when running tests on Oracle

When running tests with an Oracle database, Django creates a temporary database user. In older versions, if a password isn't manually specified in the database settings `TEST` dictionary, a hardcoded password is used. This could allow an attacker with network access to the database server to connect.

This user is usually dropped after the test suite completes, but not when using the `manage.py test --keepdb` option or if the user has an active session (such as an attacker's connection).

A randomly generated password is now used for each test run.

DNS rebinding vulnerability when `DEBUG=True`

Older versions of Django don't validate the `Host` header against `settings.ALLOWED_HOSTS` when `settings.DEBUG=True`. This makes them vulnerable to a DNS rebinding attack.

While Django doesn't ship a module that allows remote code execution, this is at least a cross-site scripting vector, which could be quite serious if developers load a copy of the production database in development or connect to some production services for which there's no development instance, for example. If a project uses a package like the `django-debug-toolbar`, then the attacker could execute arbitrary SQL, which could be especially bad if the developers connect to the database with a superuser account.

`settings.ALLOWED_HOSTS` is now validated regardless of `DEBUG`. For convenience, if `ALLOWED_HOSTS` is empty and `DEBUG=True`, the following variations of localhost are allowed `['localhost', '127.0.0.1', '::1']`. If your local settings file has your production `ALLOWED_HOSTS` value, you must now omit it to get those fallback values.

Django 1.8.15 release notes

September 26, 2016

Django 1.8.15 fixes a security issue in 1.8.14.

CSRF protection bypass on a site with Google Analytics

An interaction between Google Analytics and Django's cookie parsing could allow an attacker to set arbitrary cookies leading to a bypass of CSRF protection.

The parser for `request.COOKIES` is simplified to better match the behavior of browsers and to mitigate this attack. `request.COOKIES` may now contain cookies that are invalid according to **RFC 6265** but are possible to set via `document.cookie`.

Django 1.8.14 release notes

July 18, 2016

Django 1.8.14 fixes a security issue and a bug in 1.8.13.

XSS in admin's add/change related popup

Unsafe usage of JavaScript's `Element.innerHTML` could result in XSS in the admin's add/change related popup. `Element.textContent` is now used to prevent execution of the data.

The debug view also used `innerHTML`. Although a security issue wasn't identified there, out of an abundance of caution it's also updated to use `textContent`.

Bugfixes

- Fixed missing `varchar/text_pattern_ops` index on `CharField` and `TextField` respectively when using `AddField` on PostgreSQL (#26889).

Django 1.8.13 release notes

May 2, 2016

Django 1.8.13 fixes several bugs in 1.8.12.

Bugfixes

- Fixed `TimeField` microseconds round-tripping on MySQL and SQLite (#26498).
- Restored conversion of an empty string to null when saving values of `GenericIPAddressField` on SQLite and MySQL (#26557).

Django 1.8.12 release notes

April 1, 2016

Django 1.8.12 fixes several bugs in 1.8.11.

Bugfixes

- Made `MultiPartParser` ignore filenames that normalize to an empty string to fix crash in `MemoryFileUploadHandler` on specially crafted user input (#26325).
- Fixed data loss on SQLite where `DurationField` values with fractional seconds could be saved as `None` (#26324).
- Restored the functionality of the admin's `raw_id_fields` in `list_editable` (#26387).

Django 1.8.11 release notes

March 5, 2016

Django 1.8.11 fixes a regression on Python 2 in the 1.8.10 security release where `utils.http.is_safe_url()` crashes on bytestring URLs (#26308).

Django 1.8.10 release notes

March 1, 2016

Django 1.8.10 fixes two security issues and several bugs in 1.8.9.

CVE-2016-2512: Malicious redirect and possible XSS attack via user-supplied redirect URLs containing basic auth

Django relies on user input in some cases (e.g. *`django.contrib.auth.views.login()`* and i18n) to redirect the user to an "on success" URL. The security check for these redirects (namely `django.utils.http.is_safe_url()`) considered some URLs with basic authentication credentials "safe" when they shouldn't be.

For example, a URL like `http://mysite.example.com\@attacker.com` would be considered safe if the request's host is `http://mysite.example.com`, but redirecting to this URL sends the user to `attacker.com`.

Also, if a developer relies on `is_safe_url()` to provide safe redirect targets and puts such a URL into a link, they could suffer from an XSS attack.

CVE-2016-2513: User enumeration through timing difference on password hasher work factor upgrade

In each major version of Django since 1.6, the default number of iterations for the `PBKDF2PasswordHasher` and its subclasses has increased. This improves the security of the password as the speed of hardware increases, however, it also creates a timing difference between a login request for a user with a password encoded in an older number of iterations and login request for a nonexistent user (which runs the default hasher's default number of iterations since Django 1.6).

This only affects users who haven't logged in since the iterations were increased. The first time a user logs in after an iterations increase, their password is updated with the new iterations and there is no longer a timing difference.

The new `BasePasswordHasher.harden_runtime()` method allows hashers to bridge the runtime gap between the work factor (e.g. iterations) supplied in existing encoded passwords and the default work factor of the hasher. This method is implemented for `PBKDF2PasswordHasher` and `BCryptPasswordHasher`. The number of rounds for the latter hasher hasn't changed since Django 1.4, but some projects may subclass it and increase the work factor as needed.

A warning will be emitted for any *third-party password hashers that don't implement* a `harden_runtime()` method.

If you have different password hashes in your database (such as SHA1 hashes from users who haven't logged in since the default hasher switched to PBKDF2 in Django 1.4), the timing difference on a login request for these users may be even greater and this fix doesn't remedy that difference (or any difference when changing hashers). You may be able to *upgrade those hashes* to prevent a timing attack for that case.

Bugfixes

- Fixed a crash on PostgreSQL that prevented using `TIME_ZONE=None` and `USE_TZ=False` (#26177).
- Added system checks for query name clashes of hidden relationships (#26162).
- Made `forms.FileField` and `utils.translation.lazy_number()` picklable (#26212).
- Fixed *`RangeField`* and *`ArrayField`* serialization with `None` values (#26215).
- Reallowed dashes in top-level domain names of URLs checked by `URLValidator` to fix a regression in Django 1.8 (#26204).
- Fixed `BoundField` to reallow slices of subwidgets (#26267).
- Prevented `ContentTypeManager` instances from sharing their cache (#26286).

Django 1.8.9 release notes

February 1, 2016

Django 1.8.9 fixes several bugs in 1.8.8.

Bugfixes

- Fixed a regression that caused the "user-tools" items to display on the admin's logout page (#26035).
- Fixed a crash in the translations system when the current language has no translations (#26046).
- Fixed a regression that caused the incorrect day to be selected when opening the admin calendar widget for timezones from GMT+0100 to GMT+1200 (#24980).
- Fixed a regression in 1.8.8 causing incorrect index handling in migrations on PostgreSQL when adding `db_index=True` or `unique=True` to a `CharField` or `TextField` that already had the other specified, or when removing one of them from a field that had both, or when adding `unique=True` to a field already listed in `unique_together` (#26034).
- Fixed a crash when using an `__in` lookup inside a `Case` expression (#26071).
- Fixed a crash when using a reverse `OneToOneField` in `ModelAdmin.readonly_fields` (#26060).
- Fixed a regression in Django 1.8.5 that broke copying a `SimpleLazyObject` with `copy.copy()` (#26122).
- Fixed the `contrib.gis` map widgets when using `USE_THOUSAND_SEPARATOR=True` (#20415).

Django 1.8.8 release notes

January 2, 2016

Django 1.8.8 fixes several bugs in 1.8.7.

Python 3.2 users, please be advised that we've decided to drop support for Python 3.2 in Django 1.8.x at the end of 2016. We won't break things intentionally after that, but we won't test subsequent releases against Python 3.2 either. Upstream support for Python 3.2 ends February 2016 so we don't find much value in providing security updates for a version of Python that could be insecure. To read more about the decision and to let us know if this will be problematic for you, please read the django-developers thread.

Bugfixes

- Fixed incorrect `unique_together` field name generation by `inspectdb` (#25274).
- Corrected `__len` query lookup on `ArrayField` for empty arrays (#25772).
- Restored the ability to use custom formats from `formats.py` with `django.utils.formats.get_format()` and the `date` template filter (#25812).
- Fixed a state bug when migrating a `SeparateDatabaseAndState` operation backwards (#25896).
- Fixed missing `varchar/text_pattern_ops` index on `CharField` and `TextField` respectively when using `AlterField` on PostgreSQL (#25412).
- Fixed a state bug when using an `AlterModelManagers` operation (#25852).
- Fixed a regression which prevented using a language not in Django's default language list (*LANGUAGES*) (#25915).
- `django.views.decorators.cache.never_cache()` now sends more persuasive headers (added `no-cache, no-store, must-revalidate` to `Cache-Control`) to better prevent caching (#13008). This fixes a problem where a page refresh in Firefox cleared the selected entries in the admin's `filter_horizontal` and `filter_vertical` widgets, which could result in inadvertent data loss if a user didn't notice that and then submitted the form (#22955).
- Fixed a regression in the admin which ignored line breaks in read-only fields instead of converting them to `<br>` (#25465).
- Made `loaddata` skip disabling and enabling database constraints when it doesn't load any fixtures (#23372).
- Fixed a crash in `QuerySet.values()/values_list()` after an `annotate()` and `order_by()` when `values()/values_list()` includes a field not in the `order_by()` (#25316).

Django 1.8.7 release notes

November 24, 2015

Django 1.8.7 fixes a security issue and several bugs in 1.8.6.

Additionally, Django's vendored version of six, *django.utils.six*, has been upgraded to the latest release (1.10.0).

Fixed settings leak possibility in `date` template filter

If an application allows users to specify an unvalidated format for dates and passes this format to the *date* filter, e.g. `{{ last_updated|date:user_date_format }}`, then a malicious user could obtain any secret in the application's settings by specifying a settings key instead of a date format. e.g. `"SECRET_KEY"` instead of `"j/m/Y"`.

To remedy this, the underlying function used by the `date` template filter, `django.utils.formats.get_format()`, now only allows accessing the date/time formatting settings.

Bugfixes

- Fixed a crash of the debug view during the autumn DST change when *USE_TZ* is `False` and `pytz` is installed.
- Fixed a regression in 1.8.6 that caused database routers without an `allow_migrate()` method to crash (#25686).

- Fixed a regression in 1.8.6 by restoring the ability to use `Manager` objects for the `queryset` argument of `ModelChoiceField` (#25683).
- Fixed a regression in 1.8.6 that caused an application with South migrations in the `migrations` directory to fail (#25618).
- Fixed a data loss possibility with *`Prefetch`* if `to_attr` is set to a `ManyToManyField` (#25693).
- Fixed a regression in 1.8 by making `gettext()` once again return UTF-8 bytestrings on Python 2 if the input is a bytestring (#25720).
- Fixed serialization of *`DateRangeField`* and *`DateTimeRangeField`* (#24937).
- Fixed the exact lookup of `ArrayField` (#25666).
- Fixed `Model.refresh_from_db()` updating of `ForeignKey` fields with `on_delete=models.SET_NULL` (#25715).
- Fixed a duplicate query regression in 1.8 on proxied model deletion (#25685).
- Fixed `set_FOO_order()` crash when the `ForeignKey` of a model with `order_with_respect_to` references a model with a `OneToOneField` primary key (#25786).
- Fixed incorrect validation for `PositiveIntegerField` and `PositiveSmallIntegerField` on MySQL resulting in values greater than 4294967295 or 65535, respectively, passing validation and being silently truncated by the database (#25767).

Django 1.8.6 release notes

November 4, 2015

Django 1.8.6 adds official support for Python 3.5 and fixes several bugs in 1.8.5.

Bugfixes

- Fixed a regression causing `ModelChoiceField` to ignore `prefetch_related()` on its queryset (#25496).
- Allowed "mode=memory" in SQLite test database name if supported (#12118).
- Fixed system check crash on `ForeignKey` to abstract model (#25503).
- Fixed incorrect queries when you have multiple `ManyToManyField`s on different models that have the same field name, point to the same model, and have their reverse relations disabled (#25545).
- Allowed filtering over a `RawSQL` annotation (#25506).
- Made the `Concat` database function idempotent on SQLite (#25517).
- Avoided a confusing stack trace when starting *`runserver`* with an invalid *`INSTALLED_APPS`* setting (#25510). This regression appeared in 1.8.5 as a side effect of fixing #24704.
- Made deferred models use their proxied model's `_meta.apps` for caching and retrieval (#25563). This prevents any models generated in data migrations using `QuerySet.defer()` from leaking to test and application code.
- Fixed a typo in the name of the *strictly_above* PostGIS lookup (#25592).
- Fixed crash with `contrib.postgres.forms.SplitArrayField` and `IntegerField` on invalid value (#25597).
- Added a helpful error message when Django and South migrations exist in the same directory (#25618).

- Fixed a regression in `URLValidator` that allowed URLs with consecutive dots in the domain section (like `http://example..com/`) to pass (#25620).
- Fixed a crash with `GenericRelation` and `BaseModelAdmin.to_field_allowed` (#25622).

Django 1.8.5 release notes

October 3, 2015

Django 1.8.5 fixes several bugs in 1.8.4.

Bugfixes

- Made the development server's autoreload more robust (#24704).
- Fixed `AssertionError` in some delete queries with a model containing a field that is both a foreign and primary key (#24951).
- Fixed `AssertionError` in some complex queries (#24525).
- Fixed a migrations crash with `GenericForeignKey` (#25040).
- Made `translation.override()` clear the overridden language when a translation isn't initially active (#25295).
- Fixed crash when using a value in `ModelAdmin.list_display` that clashed with a reverse field on the model (#25299).
- Fixed autocompletion for options of non-`argparse` management commands (#25372).
- Alphabetized ordering of imports in `from django.db import migrations, models` statement in newly created migrations (#25384).
- Fixed migrations crash on MySQL when adding a text or a blob field with an unhashable default (#25393).
- Changed `Count` queries to execute `COUNT(*)` instead of `COUNT('*')` as versions of Django before 1.8 did (#25377). This may fix a performance regression on some databases.
- Fixed custom queryset chaining with `values()` and `values_list()` (#20625).
- Moved the *unsaved model instance assignment data loss check* on reverse relations to `Model.save()` (#25160).
- Readded inline foreign keys to form instances when validating model formsets (#25431).
- Allowed using ORM write methods after disabling autocommit with *`set_autocommit(False)`* (#24921).
- Fixed the `manage.py test --keepdb` option on Oracle (#25421).
- Fixed incorrect queries with multiple many-to-many fields on a model with the same 'to' model and with `related_name` set to '+' (#24505, #25486).
- Fixed pickling a `SimpleLazyObject` wrapping a model (#25389).

Django 1.8.4 release notes

August 18, 2015

Django 1.8.4 fixes a security issue and several bugs in 1.8.3.

Denial-of-service possibility in `logout()` view by filling session store

Previously, a session could be created when anonymously accessing the *`django.contrib.auth.views.logout()`* view (provided it wasn't decorated with *`login_required()`* as done in the admin). This could allow an attacker to easily create many new session records by sending repeated requests, potentially filling up the session store or causing other users' session records to be evicted.

The *`SessionMiddleware`* has been modified to no longer create empty session records, including when *`SESSION_SAVE_EVERY_REQUEST`* is active.

Bugfixes

- Added the ability to serialize values from the newly added *`UUIDField`* (#25019).
- Added a system check warning if the old `TEMPLATE_*` settings are defined in addition to the new `TEMPLATES` setting.
- Fixed `QuerySet.raw()` so `InvalidQuery` is not raised when using the `db_column` name of a `ForeignKey` field with `primary_key=True` (#12768).
- Prevented an exception in `TestCase.setUpTestData()` from leaking the transaction (#25176).
- Fixed `has_changed()` method in `contrib.postgres.forms.HStoreField` (#25215, #25233).
- Fixed the recording of squashed migrations when running the `migrate` command (#25231).
- Moved the *unsaved model instance assignment data loss check* to `Model.save()` to allow easier usage of in-memory models (#25160).
- Prevented `varchar_patterns_ops` and `text_patterns_ops` indexes for `ArrayField` (#25180).

Django 1.8.3 release notes

July 8, 2015

Django 1.8.3 fixes several security issues and bugs in 1.8.2.

Also, `django.utils.deprecation.RemovedInDjango20Warning` was renamed to `RemovedInDjango110Warning` as the version roadmap was revised to 1.9, 1.10, 1.11 (LTS), 2.0 (drops Python 2 support). For backwards compatibility, `RemovedInDjango20Warning` remains as an importable alias.

Denial-of-service possibility by filling session store

In previous versions of Django, the session backends created a new empty record in the session storage anytime `request.session` was accessed and there was a session key provided in the request cookies that didn't already have a session record. This could allow an attacker to easily create many new session records simply by sending repeated requests with unknown session keys, potentially filling up the session store or causing other users' session records to be evicted.

The built-in session backends now create a session record only if the session is actually modified; empty session records are not created. Thus this potential DoS is now only possible if the site chooses to expose a session-modifying view to anonymous users.

As each built-in session backend was fixed separately (rather than a fix in the core sessions framework), maintainers of third-party session backends should check whether the same vulnerability is present in their backend and correct it if so.

Header injection possibility since validators accept newlines in input

Some of Django's built-in validators (*EmailValidator*, most seriously) didn't prohibit newline characters (due to the usage of `$` instead of `\Z` in the regular expressions). If you use values with newlines in HTTP response or email headers, you can suffer from header injection attacks. Django itself isn't vulnerable because *HttpResponse* and the mail sending utilities in *django.core.mail* prohibit newlines in HTTP and SMTP headers, respectively. While the validators have been fixed in Django, if you're creating HTTP responses or email messages in other ways, it's a good idea to ensure that those methods prohibit newlines as well. You might also want to validate that any existing data in your application doesn't contain unexpected newlines.

validate_ipv4_address(), *validate_slug()*, and *URLValidator* are also affected, however, as of Django 1.6 the `GenericIPAddresseField`, `IPAddressField`, `SlugField`, and `URLField` form fields which use these validators all strip the input, so the possibility of newlines entering your data only exists if you are using these validators outside of the form fields.

The undocumented, internally unused `validate_integer()` function is now stricter as it validates using a regular expression instead of simply casting the value using `int()` and checking if an exception was raised.

Denial-of-service possibility in URL validation

URLValidator included a regular expression that was extremely slow to evaluate against certain invalid inputs. This regular expression has been simplified and optimized.

Bugfixes

- Fixed `BaseRangeField.prepare_value()` to use each `base_field`'s `prepare_value()` method (#24841).
- Fixed crash during *makemigrations* if a migrations module either is missing `__init__.py` or is a file (#24848).
- Fixed `QuerySet.exists()` returning incorrect results after annotation with `Count()` (#24835).
- Corrected `HStoreField.has_changed()` (#24844).
- Reverted an optimization to the CSRF template context processor which caused a regression (#24836).
- Fixed a regression which caused template context processors to overwrite variables set on a `RequestContext` after it's created (#24847).
- Prevented the loss of `null`/`not null` column properties during field renaming of MySQL databases (#24817).
- Fixed a crash when using a reverse one-to-one relation in `ModelAdmin.list_display` (#24851).
- Fixed quoting of SQL when renaming a field to `AutoField` in PostgreSQL (#24892).
- Fixed lack of unique constraint when changing a field from `primary_key=True` to `unique=True` (#24893).
- Fixed queryset pickling when using `prefetch_related()` after deleting objects (#24831).
- Allowed using `choices` longer than 1 day with `DurationField` (#24897).
- Fixed a crash when loading squashed migrations from two apps with a dependency between them, where the dependent app's replaced migrations are partially applied (#24895).
- Fixed recording of applied status for squashed (replacement) migrations (#24628).
- Fixed queryset annotations when using `Case` expressions with `exclude()` (#24833).

- Corrected join promotion for multiple `Case` expressions. Annotating a query with multiple `Case` expressions could unexpectedly filter out results (#24924).
- Fixed usage of transforms in subqueries (#24744).
- Fixed `SimpleTestCase.assertRaisesMessage()` on Python 2.7.10 (#24903).
- Provided better backwards compatibility for the `verbosity` argument in `optparse` management commands by casting it to an integer (#24769).
- Fixed `prefetch_related()` on databases other than PostgreSQL for models using UUID primary keys (#24912).
- Fixed removing `unique_together` constraints on MySQL (#24972).
- Fixed crash when uploading images with MIME types that Pillow doesn't detect, such as bitmap, in `forms.ImageField` (#24948).
- Fixed a regression when deleting a model through the admin that has a `GenericRelation` with a `related_query_name` (#24940).
- Reallowed non-ASCII values for `ForeignKey.related_name` on Python 3 by fixing the false positive system check (#25016).
- Fixed inline forms that use a parent object that has a `UUIDField` primary key and a child object that has an `AutoField` primary key (#24958).
- Fixed a regression in the `unordered_list` template filter on certain inputs (#25031).
- Fixed a regression in `URLValidator` that invalidated Punycode TLDs (#25059).
- Improved *pyinotify* `runserver` polling (#23882).

Django 1.8.2 release notes

May 20, 2015

Django 1.8.2 fixes a security issue and several bugs in 1.8.1.

Fixed session flushing in the `cached_db` backend

A change to `session.flush()` in the `cached_db` session backend in Django 1.8 mistakenly sets the session key to an empty string rather than `None`. An empty string is treated as a valid session key and the session cookie is set accordingly. Any users with an empty string in their session cookie will use the same session store. `session.flush()` is called by `django.contrib.auth.logout()` and, more seriously, by `django.contrib.auth.login()` when a user switches accounts. If a user is logged in and logs in again to a different account (without logging out) the session is flushed to avoid reuse. After the session is flushed (and its session key becomes `''`) the account details are set on the session and the session is saved. Any users with an empty string in their session cookie will now be logged into that account.

Bugfixes

- Fixed check for template engine alias uniqueness (#24685).
- Fixed crash when reusing the same `Case` instance in a query (#24752).
- Corrected join promotion for `Case` expressions. For example, annotating a query with a `Case` expression could unexpectedly filter out results (#24766).

- Fixed negated `Q` objects in expressions. Cases like `Case(When(~Q(friends__age__lte=30)))` tried to generate a subquery which resulted in a crash (#24705).
- Fixed incorrect GROUP BY clause generation on MySQL when the query's model has a self-referential foreign key (#24748).
- Implemented `ForeignKey.get_db_prep_value()` so that `ForeignKeys` pointing to *`UUIDField`* and inheritance on models with `UUIDField` primary keys work correctly (#24698, #24712).
- Fixed `isnull` lookup for `HStoreField` (#24751).
- Fixed a MySQL crash when a migration removes a combined index (unique_together or index_together) containing a foreign key (#24757).
- Fixed session cookie deletion when using *`SESSION_COOKIE_DOMAIN`* (#24799).
- On PostgreSQL, when no access is granted for the `postgres` database, Django now falls back to the default database when it normally requires a "no database" connection (#24791).
- Fixed display of `contrib.admin`'s `ForeignKey` widget when it's used in a row with other fields (#24784).

Django 1.8.1 release notes

May 1, 2015

Django 1.8.1 fixes several bugs in 1.8 and includes some optimizations in the migrations framework.

Bugfixes

- Added support for serializing `timedelta` objects in migrations (#24566).
- Restored proper parsing of the *`testserver`* command's positional arguments (fixture names) (#24571).
- Prevented `TypeError` in translation functions `check_for_language()` and `get_language_bidi()` when translations are deactivated (#24569).
- Fixed *`squashmigrations`* command when using *`SeparateDatabaseAndState`* (#24278).
- Stripped microseconds from `datetime` values when using an older version of the MySQLdb DB API driver as it does not support fractional seconds (#24584).
- Fixed a migration crash when altering *`ManyToManyField`*s (#24513).
- Fixed a crash with `QuerySet.update()` on foreign keys to one-to-one fields (#24578).
- Fixed a regression in the model detail view of *`admindocs`* when a model has a reverse foreign key relation (#24624).
- Prevented arbitrary file inclusions in *`admindocs`* (#24625).
- Fixed a crash with `QuerySet.update()` on foreign keys to instances with `uuid` primary keys (#24611).
- Fixed database introspection with SQLite 3.8.9 (released April 8, 2015) (#24637).
- Updated `urlpatterns` examples generated by *`startproject`* to remove usage of referencing views by dotted path in *`url()`* which is deprecated in Django 1.8 (#24635).
- Fixed queries where an expression was referenced in `order_by()`, but wasn't part of the select clause. An example query is `qs.annotate(foo=F('field')).values('pk').order_by('foo')` (#24615).
- Fixed a database table name quoting regression (#24605).

- Prevented the loss of `null`/`not null` column properties during field alteration of MySQL databases (#24595).
- Fixed JavaScript path of `contrib.admin`'s related field widget when using alternate static file storages (#24655).
- Fixed a migration crash when adding new relations to models (#24573).
- Fixed a migration crash when applying migrations with model managers on Python 3 that were generated on Python 2 (#24701).
- Restored the ability to use iterators as queryset filter arguments (#24719).
- Fixed a migration crash when renaming the target model of a many-to-many relation (#24725).
- Removed flushing of the test database with *`test --keepdb`*, which prevented apps with data migrations from using the option (#24729).
- Fixed `makemessages` crash in some locales (#23271).
- Fixed help text positioning of `contrib.admin` fields that use the `ModelAdmin.filter_horizontal` and `filter_vertical` options (#24676).
- Fixed *AttributeError: function 'GDALAllRegister' not found* error when initializing `contrib.gis` on Windows.

Optimizations

- Changed `ModelState` to deepcopy fields instead of deconstructing and reconstructing (#24591). This speeds up the rendering of model states and reduces memory usage when running *`manage.py migrate`* (although other changes in this release may negate any performance benefits).

Django 1.8 release notes

April 1, 2015

Welcome to Django 1.8!

These release notes cover the *new features*, as well as some *backwards incompatible changes* you'll want to be aware of when upgrading from Django 1.7 or older versions. We've also *begun the deprecation process for some features*, and some features have reached the end of their deprecation process and *have been removed*.

See the Upgrading Django to a newer version guide if you're updating an existing project.

Django 1.8 has been designated as Django's second *long-term support release*. It will receive security updates for at least three years after its release. Support for the previous LTS, Django 1.4, will end 6 months from the release date of Django 1.8.

Python compatibility

Django 1.8 requires Python 2.7, 3.2, 3.3, 3.4, or 3.5. We **highly recommend** and only officially support the latest release of each series.

Django 1.8 is the first release to support Python 3.5.

Due to the end of upstream support for Python 3.2 in February 2016, we won't test Django 1.8.x on Python 3.2 after the end of 2016.

What's new in Django 1.8

`Model._meta` API Django now has a formalized API for Model._meta, providing an officially supported way to *retrieve fields* and filter fields based on their *attributes*.

The `Model._meta` object has been part of Django since the days of pre-0.96 "Magic Removal" – it just wasn't an official, stable API. In recognition of this, we've endeavored to maintain backwards-compatibility with the old API endpoint where possible. However, API endpoints that aren't part of the new official API have been deprecated and will eventually be removed. A *guide to migrating from the old API to the new API* has been provided.

Multiple template engines Django 1.8 defines a stable API for integrating template backends. It includes built-in support for the Django template language and for *Jinja2*. It supports rendering templates with multiple engines within the same project. Learn more about the new features in the topic guide and check the upgrade instructions for details.

Security enhancements Several features of the django-secure third-party library have been integrated into Django. *django.middleware.security.SecurityMiddleware* provides several security enhancements to the request/response cycle. The new *check --deploy* option allows you to check your production settings file for ways to increase the security of your site.

New PostgreSQL specific functionality Django now has a module with extensions for PostgreSQL specific features, such as *ArrayField*, *HStoreField*, *Range Fields*, and *unaccent* lookup. A full breakdown of the features is available in the documentation.

New data types

- Django now has a *UUIDField* for storing universally unique identifiers. It is stored as the native `uuid` data type on PostgreSQL and as a fixed length character field on other backends. There is a corresponding *form field*.
- Django now has a *DurationField* for storing periods of time - modeled in Python by `timedelta`. It is stored in the native `interval` data type on PostgreSQL, as a `INTERVAL DAY(9) TO SECOND(6)` on Oracle, and as a `bigint` of microseconds on other backends. Date and time related arithmetic has also been improved on all backends. There is a corresponding *form field*.

Query Expressions, Conditional Expressions, and Database Functions Query Expressions allow you to create, customize, and compose complex SQL expressions. This has enabled annotate to accept expressions other than aggregates. Aggregates are now able to reference multiple fields, as well as perform arithmetic, similar to `F()` objects. *order_by()* has also gained the ability to accept expressions.

Conditional Expressions allow you to use `if` ... `elif` ... `else` logic within queries.

A collection of database functions is also included with functionality such as *Coalesce*, *Concat*, and *Substr*.

`TestCase` data setup *TestCase* has been refactored to allow for data initialization at the class level using transactions and savepoints. Database backends which do not support transactions, like MySQL with the MyISAM storage engine, will still be able to run these tests but won't benefit from the improvements. Tests are now run within two nested *atomic()* blocks: one for the whole class and one for each test.

- The class method *TestCase.setUpTestData()* adds the ability to setup test data at the class level. Using this technique can speed up the tests as compared to using `setUp()`.
- Fixture loading within `TestCase` is now performed once for the whole `TestCase`.

Minor features

django.contrib.admin

- *ModelAdmin* now has a *has_module_permission()* method to allow limiting access to the module on the admin index page.
- *InlineModelAdmin* now has an attribute *show_change_link* that supports showing a link to an inline object's change form.
- Use the new `django.contrib.admin.RelatedOnlyFieldListFilter` in *ModelAdmin.list_filter* to limit the `list_filter` choices to foreign objects which are attached to those from the `ModelAdmin`.
- The *ModelAdmin.delete_view()* displays a summary of objects to be deleted on the deletion confirmation page.
- The jQuery library embedded in the admin has been upgraded to version 1.11.2.
- You can now specify *AdminSite.site_url* in order to display a link to the front-end site.
- You can now specify *ModelAdmin.show_full_result_count* to control whether or not the full count of objects should be displayed on a filtered admin page.
- The `AdminSite.password_change()` method now has an `extra_context` parameter.
- You can now control who may login to the admin site by overriding only *AdminSite.has_permission()* and *AdminSite.login_form*. The `base.html` template has a new block `usertools` which contains the user-specific header. A new context variable `has_permission`, which gets its value from *has_permission()*, indicates whether the user may access the site.
- Foreign key dropdowns now have buttons for changing or deleting related objects using a popup.

django.contrib.admindocs

- reStructuredText is now parsed in model docstrings.

django.contrib.auth

- Authorization backends can now raise *PermissionDenied* in *has_perm()* and *has_module_perms()* to short-circuit permission checking.
- *PasswordResetForm* now has a method *send_email()* that can be overridden to customize the mail to be sent.
- The `max_length` of *Permission.name* has been increased from 50 to 255 characters. Please run the database migration.
- *USERNAME_FIELD* and *REQUIRED_FIELDS* now supports *ForeignKey*s.
- The default iteration count for the PBKDF2 password hasher has been increased by 33%. This backwards compatible change will not affect users who have subclassed `django.contrib.auth.hashers.PBKDF2PasswordHasher` to change the default value.

django.contrib.gis

- A new GeoJSON serializer is now available.
- It is now allowed to include a subquery as a geographic lookup argument, for example `City.objects.filter(point__within=Country.objects.filter(continent='Africa').values('m`

- The SpatiaLite backend now supports `Collect` and `Extent` aggregates when the database version is 3.0 or later.
- The PostGIS 2 `CREATE EXTENSION postgis` and the SpatiaLite `SELECT InitSpatialMetaData` initialization commands are now automatically run by *`migrate`*.
- The GDAL interface now supports retrieving properties of *raster (image) data file*.
- Compatibility shims for `SpatialRefSys` and `GeometryColumns` changed in Django 1.2 have been removed.
- All GDAL-related exceptions are now raised with `GDALException`. The former `OGRException` has been kept for backwards compatibility but should not be used any longer.

`django.contrib.sessions`

- Session cookie is now deleted after *`flush()`* is called.

`django.contrib.sitemaps`

- The new *`Sitemap.i18n`* attribute allows you to generate a sitemap based on the *`LANGUAGES`* setting.

`django.contrib.sites`

- *`get_current_site()`* will now lookup the current site based on *`request.get_host()`* if the *`SITE_ID`* setting is not defined.
- The default *`Site`* created when running `migrate` now respects the *`SITE_ID`* setting (instead of always using `pk=1`).

Cache

- The `incr()` method of the `django.core.cache.backends.locmem.LocMemCache` backend is now thread-safe.

Cryptography

- The `max_age` parameter of the *`django.core.signing.TimestampSigner.unsign()`* method now also accepts a `datetime.timedelta` object.

Database backends

- The MySQL backend no longer strips microseconds from `datetime` values as MySQL 5.6.4 and up supports fractional seconds depending on the declaration of the datetime field (when `DATETIME` includes fractional precision greater than 0). New datetime database columns created with Django 1.8 and MySQL 5.6.4 and up will support microseconds. See the *MySQL database notes* for more details.
- The MySQL backend no longer creates explicit indexes for foreign keys when using the InnoDB storage engine, as MySQL already creates them automatically.
- The Oracle backend no longer defines the `connection_persists_old_columns` feature as `True`. Instead, Oracle will now include a cache busting clause when getting the description of a table.

Email

- *Email backends* now support the context manager protocol for opening and closing connections.
- The SMTP email backend now supports `keyfile` and `certfile` authentication with the *EMAIL_SSL_CERTFILE* and *EMAIL_SSL_KEYFILE* settings.
- The SMTP *EmailBackend* now supports setting the `timeout` parameter with the *EMAIL_TIMEOUT* setting.
- *EmailMessage* and `EmailMultiAlternatives` now support the `reply_to` parameter.

File Storage

- *Storage.get_available_name()* and *Storage.save()* now take a `max_length` argument to implement storage-level maximum filename length constraints. Filenames exceeding this argument will get truncated. This prevents a database error when appending a unique suffix to a long filename that already exists on the storage. See the *deprecation note* about adding this argument to your custom storage classes.

Forms

- Form widgets now render attributes with a value of `True` or `False` as HTML5 boolean attributes.
- The new *has_error()* method allows checking if a specific error has happened.
- If *required_css_class* is defined on a form, then the `<label>` tags for required fields will have this class present in its attributes.
- The rendering of non-field errors in unordered lists (`<ul>`) now includes `nonfield` in its list of classes to distinguish them from field-specific errors.
- *Field* now accepts a *label_suffix* argument, which will override the form's *label_suffix*. This enables customizing the suffix on a per-field basis — previously it wasn't possible to override a form's *label_suffix* while using shortcuts such as `{{ form.as_p }}` in templates.
- *SelectDateWidget* now accepts an *empty_label* argument, which will override the top list choice label when *DateField* is not required.
- After an *ImageField* has been cleaned and validated, the `UploadedFile` object will have an additional `image` attribute containing the Pillow `Image` instance used to check if the file was a valid image. It will also update `UploadedFile.content_type` with the image's content type as determined by Pillow.
- You can now pass a callable that returns an iterable of choices when instantiating a *ChoiceField*.

Generic Views

- Generic views that use *MultipleObjectMixin* may now specify the ordering applied to the *queryset* by setting *ordering* or overriding *get_ordering()*.
- The new *SingleObjectMixin.query_pk_and_slug* attribute allows changing the behavior of *get_object()* so that it'll perform its lookup using both the primary key and the slug.
- The *get_form()* method doesn't require a `form_class` to be provided anymore. If not provided `form_class` defaults to *get_form_class()*.
- Placeholders in *ModelFormMixin.success_url* now support the Python `str.format()` syntax. The legacy `%(<foo>)s` syntax is still supported but will be removed in Django 1.10.

Internationalization

- *FORMAT_MODULE_PATH* can now be a list of strings representing module paths. This allows importing several format modules from different reusable apps. It also allows overriding those custom formats in your main Django project.

Logging

- The *django.utils.log.AdminEmailHandler* class now has a *send_mail()* method to make it more subclass friendly.

Management Commands

- Database connections are now always closed after a management command called from the command line has finished doing its job.
- Commands from alternate package formats like eggs are now also discovered.
- The new *dumpdata --output* option allows specifying a file to which the serialized data is written.
- The new *makemessages --exclude* and *compilemessages --exclude* options allow excluding specific locales from processing.
- *compilemessages* now has a `--use-fuzzy` or `-f` option which includes fuzzy translations into compiled files.
- The *loaddata --ignorenonexistent* option now ignores data for models that no longer exist.
- *runserver* now uses daemon threads for faster reloading.
- *inspectdb* now outputs `Meta.unique_together`. It is also able to introspect *AutoField* for MySQL and PostgreSQL databases.
- When calling management commands with options using *call_command()*, the option name can match the command line option name (without the initial dashes) or the final option destination variable name, but in either case, the resulting option received by the command is now always the `dest` name specified in the command option definition (as long as the command uses the `argparse` module).
- The *dbshell* command now supports MySQL's optional SSL certificate authority setting (`--ssl-ca`).
- The new *makemigrations --name* allows giving the migration(s) a custom name instead of a generated one.
- The *loaddata* command now prevents repeated fixture loading. If *FIXTURE_DIRS* contains duplicates or a default fixture directory path (`app_name/fixtures`), an exception is raised.
- The new *makemigrations --exit* option allows exiting with an error code if no migrations are created.
- The new *showmigrations* command allows listing all migrations and their dependencies in a project.

Middleware

- The *CommonMiddleware.response_redirect_class* attribute allows you to customize the redirects issued by the middleware.
- A debug message will be logged to the `django.request` logger when a middleware raises a *MiddlewareNotUsed* exception in *DEBUG* mode.

Migrations

- The *RunSQL* operation can now handle parameters passed to the SQL statements.
- It is now possible to have migrations (most probably *data migrations*) for applications without models.
- Migrations can now *serialize model managers* as part of the model state.
- A *generic mechanism to handle the deprecation of model fields* was added.
- The *RunPython.noop()* and *RunSQL.noop* class method/attribute were added to ease in making `RunPython` and `RunSQL` operations reversible.
- The migration operations *RunPython* and *RunSQL* now call the *allow_migrate()* method of database routers. The router can use the newly introduced `app_label` and `hints` arguments to make a routing decision. To take advantage of this feature you need to update the router to the new `allow_migrate` signature, see the *deprecation section* for more details.

Models

- Django now logs at most 9000 queries in `connections.queries`, in order to prevent excessive memory usage in long-running processes in debug mode.
- There is now a model `Meta` option to define a *default_related_name* for all relational fields of a model.
- Pickling models and querysets across different versions of Django isn't officially supported (it may work, but there's no guarantee). An extra variable that specifies the current Django version is now added to the pickled state of models and querysets, and Django raises a `RuntimeWarning` when these objects are unpickled in a different version than the one in which they were pickled.
- Added *Model.from_db()* which Django uses whenever objects are loaded using the ORM. The method allows customizing model loading behavior.
- `extra(select={...})` now allows you to escape a literal `%s` sequence using `%%s`.
- Custom Lookups can now be registered using a decorator pattern.
- The new *Transform.bilateral* attribute allows creating bilateral transformations. These transformations are applied to both `lhs` and `rhs` when used in a lookup expression, providing opportunities for more sophisticated lookups.
- SQL special characters (, %, _) are now escaped properly when a pattern lookup (e.g. `contains`, `startswith`, etc.) is used with an `F()` expression as the right-hand side. In those cases, the escaping is performed by the database, which can lead to somewhat complex queries involving nested `REPLACE` function calls.
- You can now refresh model instances by using *Model.refresh_from_db()*.
- You can now get the set of deferred fields for a model using *Model.get_deferred_fields()*.
- Model field `default`'s are now used when primary key field's are set to `None`.

Signals

- Exceptions from the `(receiver, exception)` tuples returned by *Signal.send_robust()* now have their traceback attached as a `__traceback__` attribute.
- The `environ` argument, which contains the WSGI environment structure from the request, was added to the *request_started* signal.
- You can now import the *setting_changed()* signal from `django.core.signals` to avoid loading `django.test` in non-test situations. Django no longer does so itself.

System Check Framework

- *register* can now be used as a function.

Templates

- *urlize* now supports domain-only links that include characters after the top-level domain (e.g. `djangoproject.com/` and `djangoproject.com/download/`).
- *urlize* doesn't treat exclamation marks at the end of a domain or its query string as part of the URL (the URL in e.g. `'djangoproject.com!` is `djangoproject.com`)
- Added a *locmem.Loader* class that loads Django templates from a Python dictionary.
- The *now* tag can now store its output in a context variable with the usual syntax: `{% now 'j n Y' as varname %}`.

Requests and Responses

- `WSGIRequest` now respects paths starting with `//`.
- The *HttpRequest.build_absolute_uri()* method now handles paths starting with `//` correctly.
- If *DEBUG* is `True` and a request raises a *SuspiciousOperation*, the response will be rendered with a detailed error page.
- The `query_string` argument of *QueryDict* is now optional, defaulting to `None`, so a blank `QueryDict` can now be instantiated with `QueryDict()` instead of `QueryDict(None)` or `QueryDict('')`.
- The `GET` and `POST` attributes of an *HttpRequest* object are now *QueryDict*s rather than dictionaries, and the `FILES` attribute is now a `MultiValueDict`. This brings this class into line with the documentation and with `WSGIRequest`.
- The *HttpResponse.charset* attribute was added.
- `WSGIRequestHandler` now follows RFC in converting URI to IRI, using `uri_to_iri()`.
- The *HttpRequest.get_full_path()* method now escapes unsafe characters from the path portion of a Uniform Resource Identifier (URI) properly.
- *HttpResponse* now implements a few additional methods like *getvalue()* so that instances can be used as stream objects.
- The new *HttpResponse.setdefault()* method allows setting a header unless it has already been set.
- You can use the new *FileResponse* to stream files.
- The *condition()* decorator for conditional view processing now supports the `If-unmodified-since` header.

Tests

- The *RequestFactory.trace()* and *Client.trace()* methods were implemented, allowing you to create `TRACE` requests in your tests.
- The `count` argument was added to *assertTemplateUsed()*. This allows you to assert that a template was rendered a specific number of times.
- The new *assertJSONNotEqual()* assertion allows you to test that two JSON fragments are not equal.
- Added options to the *test* command to preserve the test database (*--keepdb*), to run the test cases in reverse order (*--reverse*), and to enable SQL logging for failing tests (*--debug-sql*).

- Added the *resolver_match* attribute to test client responses.
- Added several settings that allow customization of test tablespace parameters for Oracle: *DATAFILE*, *DATAFILE_TMP*, *DATAFILE_MAXSIZE* and *DATAFILE_TMP_MAXSIZE*.
- The *override_settings()* decorator can now affect the master router in *DATABASE_ROUTERS*.
- Added test client support for file uploads with file-like objects.
- A shared cache is now used when testing with an SQLite in-memory database when using Python 3.4+ and SQLite 3.7.13+. This allows sharing the database between threads.

Validators

- *URLValidator* now supports IPv6 addresses, unicode domains, and URLs containing authentication data.

Backwards incompatible changes in 1.8

: In addition to the changes outlined in this section, be sure to review the *deprecation plan* for any features that have been removed. If you haven't updated your code within the deprecation timeline for a given feature, its removal may appear as a backwards incompatible change.

Related object operations are run in a transaction Some operations on related objects such as *add()* or direct assignment ran multiple data modifying queries without wrapping them in transactions. To reduce the risk of data corruption, all data modifying methods that affect multiple related objects (i.e. `add()`, `remove()`, `clear()`, and direct assignment) now perform their data modifying queries from within a transaction, provided your database supports transactions.

This has one backwards incompatible side effect, signal handlers triggered from these methods are now executed within the method's transaction and any exception in a signal handler will prevent the whole operation.

Assigning unsaved objects to relations raises an error

: To more easily allow in-memory usage of models, this change was reverted in Django 1.8.4 and replaced with a check during `model.save()`. For example:

```
>>> book = Book.objects.create(name="Django")
>>> book.author = Author(name="John")
>>> book.save()
Traceback (most recent call last):
...
ValueError: save() prohibited to prevent data loss due to unsaved related object 'author'.
```

A similar check on assignment to reverse one-to-one relations was removed in Django 1.8.5.

Assigning unsaved objects to a *ForeignKey*, *GenericForeignKey*, and *OneToOneField* now raises a `ValueError`.

Previously, the assignment of an unsaved object would be silently ignored. For example:

```
>>> book = Book.objects.create(name="Django")
>>> book.author = Author(name="John")
>>> book.author.save()
>>> book.save()
```

```
>>> Book.objects.get(name="Django")
>>> book.author
>>>
```

Now, an error will be raised to prevent data loss:

```
>>> book.author = Author(name="john")
Traceback (most recent call last):
...
ValueError: Cannot assign "<Author: John>": "Author" instance isn't saved in the database.
```

If you require allowing the assignment of unsaved instances (the old behavior) and aren't concerned about the data loss possibility (e.g. you never save the objects to the database), you can disable this check by using the `ForeignKey.allow_unsaved_instance_assignment` attribute. (This attribute was removed in 1.8.4 as it's no longer relevant.)

Management commands that only accept positional arguments If you have written a custom management command that only accepts positional arguments and you didn't specify the `args` command variable, you might get an error like `Error: unrecognized arguments: ...`, as variable parsing is now based on `argparse` which doesn't implicitly accept positional arguments. You can make your command backwards compatible by simply setting the `args` class variable. However, if you don't have to keep compatibility with older Django versions, it's better to implement the new *`add_arguments()`* method as described in Writing custom django-admin commands.

Custom test management command arguments through test runner The method to add custom arguments to the *test* management command through the test runner has changed. Previously, you could provide an *option_list* class variable on the test runner to add more arguments (à la `optparse`). Now to implement the same behavior, you have to create an `add_arguments(cls, parser)` class method on the test runner and call `parser.add_argument` to add any custom arguments, as parser is now an `argparse.ArgumentParser` instance.

Model check ensures auto-generated column names are within limits specified by database A field name that's longer than the column name length supported by a database can create problems. For example, with MySQL you'll get an exception trying to create the column, and with PostgreSQL the column name is truncated by the database (you may see a warning in the PostgreSQL logs).

A model check has been introduced to better alert users to this scenario before the actual creation of database tables.

If you have an existing model where this check seems to be a false positive, for example on PostgreSQL where the name was already being truncated, simply use *`db_column`* to specify the name that's being used.

The check also applies to the columns generated in an implicit `ManyToManyField.through` model. If you run into an issue there, use *`through`* to create an explicit model and then specify *`db_column`* on its column(s) as needed.

Query relation lookups now check object types Querying for model lookups now checks if the object passed is of correct type and raises a `ValueError` if not. Previously, Django didn't care if the object was of correct type; it just used the object's related field attribute (e.g. `id`) for the lookup. Now, an error is raised to prevent incorrect lookups:

```
>>> book = Book.objects.create(name="Django")
>>> book = Book.objects.filter(author=book)
Traceback (most recent call last):
...
ValueError: Cannot query "<Book: Django>": Must be "Author" instance.
```

`select_related()` now checks given fields `select_related()` now validates that the given fields actually exist. Previously, nonexistent fields were silently ignored. Now, an error is raised:

```
>>> book = Book.objects.select_related('nonexistent_field')
Traceback (most recent call last):
...
FieldError: Invalid field name(s) given in select_related: 'nonexistent_field'
```

The validation also makes sure that the given field is relational:

```
>>> book = Book.objects.select_related('name')
Traceback (most recent call last):
...
FieldError: Non-relational field given in select_related: 'name'
```

Default `EmailField.max_length` increased to 254 The old default 75 character `max_length` was not capable of storing all possible RFC3696/5321-compliant email addresses. In order to store all possible valid email addresses, the `max_length` has been increased to 254 characters. You will need to generate and apply database migrations for your affected models (or add `max_length=75` if you wish to keep the length on your current fields). A migration for *`django.contrib.auth.models.User.email`* is included.

Support for PostgreSQL versions older than 9.0 The end of upstream support periods was reached in July 2014 for PostgreSQL 8.4. As a consequence, Django 1.8 sets 9.0 as the minimum PostgreSQL version it officially supports.

This also includes dropping support for PostGIS 1.3 and 1.4 as these versions are not supported on versions of PostgreSQL later than 8.4.

Django also now requires the use of Psycopg2 version 2.4.5 or higher (or 2.5+ if you want to use *`django.contrib.postgres`*).

Support for MySQL versions older than 5.5 The end of upstream support periods was reached in January 2012 for MySQL 5.0 and December 2013 for MySQL 5.1. As a consequence, Django 1.8 sets 5.5 as the minimum MySQL version it officially supports.

Support for Oracle versions older than 11.1 The end of upstream support periods was reached in July 2010 for Oracle 9.2, January 2012 for Oracle 10.1, and July 2013 for Oracle 10.2. As a consequence, Django 1.8 sets 11.1 as the minimum Oracle version it officially supports.

Specific privileges used instead of roles for tests on Oracle Earlier versions of Django granted the CONNECT and RESOURCE roles to the test user on Oracle. These roles have been deprecated, so Django 1.8 uses the specific underlying privileges instead. This changes the privileges required of the main user for running tests (unless the project is configured to avoid creating a test user). The exact privileges required now are detailed in *Oracle notes*.

`AbstractUser.last_login` allows null values The *`AbstractUser.last_login`* field now allows null values. Previously, it defaulted to the time when the user was created which was misleading if the user never logged in. If you are using the default user (*`django.contrib.auth.models.User`*), run the database migration included in `contrib.auth`.

If you are using a custom user model that inherits from `AbstractUser`, you'll need to run *`makemigrations`* and generate a migration for your app that contains that model. Also, if wish to set `last_login` to `NULL` for users who haven't logged in, you can run this query:

```
from django.db import models
from django.contrib.auth import get_user_model
from django.contrib.auth.models import AbstractBaseUser

UserModel = get_user_model()
if issubclass(UserModel, AbstractBaseUser):
    UserModel._default_manager.filter(
        last_login=models.F('date_joined')
    ).update(last_login=None)
```

`django.contrib.gis`

- Support for GEOS 3.1 and GDAL 1.6 has been dropped.
- Support for SpatiaLite < 2.4 has been dropped.
- GIS-specific lookups have been refactored to use the *`django.db.models.Lookup`* API.
- The default `str` representation of *`GEOSGeometry`* objects has been changed from WKT to EWKT format (including the SRID). As this representation is used in the serialization framework, that means that `dumpdata` output will now contain the SRID value of geometry objects.

Priority of context processors for `TemplateResponse` brought in line with `render` The *`TemplateResponse`* constructor is designed to be a drop-in replacement for the *`render()`* function. However, it had a slight incompatibility, in that for `TemplateResponse`, context data from the passed in context dictionary could be shadowed by context data returned from context processors, whereas for `render` it was the other way around. This was a bug, and the behavior of `render` is more appropriate, since it allows the globally defined context processors to be overridden locally in the view. If you were relying on the fact context data in a `TemplateResponse` could be overridden using a context processor, you will need to change your code.

Overriding `setUpClass` / `tearDownClass` in test cases The decorators *`override_settings()`* and *`modify_settings()`* now act at the class level when used as class decorators. As a consequence, when overriding `setUpClass()` or `tearDownClass()`, the `super` implementation should always be called.

Removal of `django.contrib.formtools` The formtools contrib app has been moved to a separate package and the relevant documentation pages have been updated or removed.

The new package is available on GitHub and on PyPI.

Database connection reloading between tests Django previously closed database connections between each test within a `TestCase`. This is no longer the case as Django now wraps the whole `TestCase` within a transaction. If some of your tests relied on the old behavior, you should have them inherit from `TransactionTestCase` instead.

Cleanup of the `django.template` namespace If you've been relying on private APIs exposed in the `django.template` module, you may have to import them from `django.template.base` instead.

Also private APIs `django.template.base.compile_string()`, `django.template.loader.find_template()`, and `django.template.loader.get_template_from_string()` were removed.

`model` attribute on private model relations In earlier versions of Django, on a model with a reverse foreign key relationship (for example), `model._meta.get_all_related_objects()` returned the relationship as a `django.db.models.related.RelatedObject` with the `model` attribute set to the source of the relationship. Now, this method returns the relationship as `django.db.models.fields.related.ManyToOneRel` (private API `RelatedObject` has been removed), and the `model` attribute is set to the target of the relationship instead of the source. The source model is accessible on the `related_model` attribute instead.

Consider this example from the tutorial in Django 1.8:

```
>>> p = Poll.objects.get(pk=1)
>>> p._meta.get_all_related_objects()
[<ManyToOneRel: polls.choice>]
>>> p._meta.get_all_related_objects()[0].model
<class 'polls.models.Poll'>
>>> p._meta.get_all_related_objects()[0].related_model
<class 'polls.models.Choice'>
```

and compare it to the behavior on older versions:

```
>>> p._meta.get_all_related_objects()
[<RelatedObject: polls:choice related to poll>]
>>> p._meta.get_all_related_objects()[0].model
<class 'polls.models.Choice'>
```

To access the source model, you can use a pattern like this to write code that will work with both Django 1.8 and older versions:

```
for relation in opts.get_all_related_objects():
    to_model = getattr(relation, 'related_model', relation.model)
```

Also note that `get_all_related_objects()` is deprecated in 1.8. See the *upgrade guide* for the new API.

Database backend API The following changes to the database backend API are documented to assist those writing third-party backends in updating their code:

- `BaseDatabaseXXX` classes have been moved to `django.db.backends.base`. Please import them from the new locations:

```
    from django.db.backends.base.base import BaseDatabaseWrapper
    from django.db.backends.base.client import BaseDatabaseClient
    from django.db.backends.base.creation import BaseDatabaseCreation
    from django.db.backends.base.features import BaseDatabaseFeatures
    from django.db.backends.base.introspection import BaseDatabaseIntrospection
    from django.db.backends.base.introspection import FieldInfo, TableInfo
    from django.db.backends.base.operations import BaseDatabaseOperations
    from django.db.backends.base.schema import BaseDatabaseSchemaEditor
    from django.db.backends.base.validation import BaseDatabaseValidation
```

- The `data_types`, `data_types_suffix`, and `data_type_check_constraints` attributes have moved from the `DatabaseCreation` class to `DatabaseWrapper`.
- The `SQLCompiler.as_sql()` method now takes a `subquery` parameter (#24164).
- The `BaseDatabaseOperations.date_interval_sql()` method now only takes a `timedelta` parameter.

`django.contrib.admin`

- `AdminSite` no longer takes an `app_name` argument and its `app_name` attribute has been removed. The application name is always `admin` (as opposed to the instance name which you can still customize using `AdminSite(name="...")`.
- The `ModelAdmin.get_object()` method (private API) now takes a third argument named `from_field` in order to specify which field should match the provided `object_id`.
- The *`ModelAdmin.response_delete()`* method now takes a second argument named `obj_id` which is the serialized identifier used to retrieve the object before deletion.

Default autoescaping of functions in `django.template.defaultfilters` In order to make built-in template filters that output HTML "safe by default" when calling them in Python code, the following functions in `django.template.defaultfilters` have been changed to automatically escape their input value:

- `join`
- `linebreaksbr`
- `linebreaks_filter`
- `linenumbers`
- `unordered_list`
- `urlize`
- `urlizetrunc`

You can revert to the old behavior by specifying `autoescape=False` if you are passing trusted content. This change doesn't have any effect when using the corresponding filters in templates.

Miscellaneous

- `connections.queries` is now a read-only attribute.
- Database connections are considered equal only if they're the same object. They aren't hashable any more.
- *`GZipMiddleware`* used to disable compression for some content types when the request is from Internet Explorer, in order to work around a bug in IE6 and earlier. This behavior could affect performance on IE7 and later. It was removed.
- `URLField.to_python` no longer adds a trailing slash to pathless URLs.
- The *`length`* template filter now returns `0` for an undefined variable, rather than an empty string.
- `ForeignKey.default_error_message['invalid']` has been changed from `'%(model)s instance with pk %(pk)r does not exist.'` to `'%(model)s instance with %(field)s %(value)r does not exist.'` If you are using this message in your own code, please update the list of interpolated parameters. Internally, Django will continue to provide the `pk` parameter in `params` for backwards compatibility.
- `UserCreationForm.error_messages['duplicate_username']` is no longer used. If you wish to customize that error message, *override it on the form* using the `'unique'` key in `Meta.error_messages['username']` or, if you have a custom form field for `'username'`, using the the `'unique'` key in its *`error_messages`* argument.
- The block `usertools` in the `base.html` template of *`django.contrib.admin`* now requires the `has_permission` context variable to be set. If you have any custom admin views that use this template, update them to pass *`AdminSite.has_permission()`* as this new variable's value or simply include *`AdminSite.each_context(request)`* in the context.

- Internal changes were made to the *ClearableFileInput* widget to allow more customization. The undocumented `url_markup_template` attribute was removed in favor of `template_with_initial`.
- For consistency with other major vendors, the `en_GB` locale now has Monday as the first day of the week.
- Seconds have been removed from any locales that had them in `TIME_FORMAT`, `DATETIME_FORMAT`, or `SHORT_DATETIME_FORMAT`.
- The default max size of the Oracle test tablespace has increased from 300M (or 200M, before 1.7.2) to 500M.
- `reverse()` and `reverse_lazy()` now return Unicode strings instead of byte strings.
- The `CacheClass` shim has been removed from all cache backends. These aliases were provided for backwards compatibility with Django 1.3. If you are still using them, please update your project to use the real class name found in the *BACKEND* key of the *CACHES* setting.
- By default, *call_command()* now always skips the check framework (unless you pass it `skip_checks=False`).
- When iterating over lines, *File* now uses universal newlines. The following are recognized as ending a line: the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old Macintosh convention `'\r'`.
- The Memcached cache backends `MemcachedCache` and `PyLibMCCache` will delete a key if `set()` fails. This is necessary to ensure the `cache_db` session store always fetches the most current session data.
- Private APIs `override_template_loaders` and `override_with_test_loader` in `django.test.utils` were removed. Override `TEMPLATES` with `override_settings` instead.
- Warnings from the MySQL database backend are no longer converted to exceptions when *DEBUG* is `True`.
- *HttpRequest* now has a simplified `repr` (e.g. `<WSGIRequest: GET '/somepath/'>`). This won't change the behavior of the *SafeExceptionReporterFilter* class.
- Class-based views that use *ModelFormMixin* will raise an *ImproperlyConfigured* exception when both the `fields` and `form_class` attributes are specified. Previously, `fields` was silently ignored.
- When following redirects, the test client now raises *RedirectCycleError* if it detects a loop or hits a maximum redirect limit (rather than passing silently).
- Translatable strings set as the `default` parameter of the field are cast to concrete strings later, so the return type of `Field.get_default()` is different in some cases. There is no change to default values which are the result of a callable.
- `GenericIPAddressField.empty_strings_allowed` is now `False`. Database backends that interpret empty strings as null (only Oracle among the backends that Django includes) will no longer convert null values back to an empty string. This is consistent with other backends.
- When the *leave_locale_alone* attribute is `False`, translations are now deactivated instead of forcing the "en-us" locale. In the case your models contained non-English strings and you counted on English translations to be activated in management commands, this will not happen any longer. It might be that new database migrations are generated (once) after migrating to 1.8.
- *django.utils.translation.get_language()* now returns `None` instead of *LANGUAGE_CODE* when translations are temporarily deactivated.
- When a translation doesn't exist for a specific literal, the fallback is now taken from the *LANGUAGE_CODE* language (instead of from the untranslated `msgid` message).
- The `name` field of *django.contrib.contenttypes.models.ContentType* has been removed by a migration and replaced by a property. That means it's not possible to query or filter a `ContentType` by this field any longer.

Be careful if you upgrade to Django 1.8 and skip Django 1.7. If you run `manage.py migrate --fake`, this migration will be skipped and you'll see a `RuntimeError: Error creating new content types.` exception because the `name` column won't be dropped from the database. Use `manage.py migrate --fake-initial` to fake only the initial migration instead.

- The new *`migrate --fake-initial`* option allows faking initial migrations. In 1.7, initial migrations were always automatically faked if all tables created in an initial migration already existed.
- An app *without* migrations with a `ForeignKey` to an app *with* migrations may now result in a foreign key constraint error when migrating the database or running tests. In Django 1.7, this could fail silently and result in a missing constraint. To resolve the error, add migrations to the app without them.

Features deprecated in 1.8

Selected methods in `django.db.models.options.Options` As part of the formalization of the `Model._meta` API (from the *`django.db.models.options.Options`* class), a number of methods have been deprecated and will be removed in Django 1.10:

- `get_all_field_names()`
- `get_all_related_objects()`
- `get_all_related_objects_with_model()`
- `get_all_related_many_to_many_objects()`
- `get_all_related_m2m_objects_with_model()`
- `get_concrete_fields_with_model()`
- `get_field_by_name()`
- `get_fields_with_model()`
- `get_m2m_with_model()`

A *migration guide* has been provided to assist in converting your code from the old API to the new, official API.

Loading `cycle` and `firstof` template tags from `future` library Django 1.6 introduced `{% load cycle from future %}` and `{% load firstof from future %}` syntax for forward compatibility of the *`cycle`* and *`firstof`* template tags. This syntax is now deprecated and will be removed in Django 1.10. You can simply remove the `{% load ... from future %}` tags.

`django.conf.urls.patterns()` In the olden days of Django, it was encouraged to reference views as strings in `urlpatterns`:

```
urlpatterns = patterns('',
    url('^$', 'myapp.views.myview'),
)
```

and Django would magically import `myapp.views.myview` internally and turn the string into a real function reference. In order to reduce repetition when referencing many views from the same module, the `patterns()` function takes a required initial `prefix` argument which is prepended to all views-as-strings in that set of `urlpatterns`:

```
urlpatterns = patterns('myapp.views',
    url('^$', 'myview'),
    url('^other/$', 'otherview'),
)
```

In the modern era, we have updated the tutorial to instead recommend importing your views module and referencing your view functions (or classes) directly. This has a number of advantages, all deriving from the fact that we are using normal Python in place of "Django String Magic": the errors when you mistype a view name are less obscure, IDEs can help with autocompletion of view names, etc.

So these days, the above use of the `prefix` arg is much more likely to be written (and is better written) as:

```
from myapp import views

urlpatterns = patterns('',
    url('^$', views.myview),
    url('^other/$', views.otherview),
)
```

Thus `patterns()` serves little purpose and is a burden when teaching new users (answering the newbie's question "why do I need this empty string as the first argument to `patterns()`?"). For these reasons, we are deprecating it. Updating your code is as simple as ensuring that `urlpatterns` is a list of *`django.conf.urls.url()`* instances. For example:

```
from django.conf.urls import url
from myapp import views

urlpatterns = [
    url('^$', views.myview),
    url('^other/$', views.otherview),
]
```

Passing a string as `view` to `url()` Related to the previous item, referencing views as strings in the `url()` function is deprecated. Pass the callable view as described in the previous section instead.

Template-related settings As a consequence of the multiple template engines refactor, several settings are deprecated in favor of *`TEMPLATES`*:

- `ALLOWED_INCLUDE_ROOTS`
- `TEMPLATE_CONTEXT_PROCESSORS`
- `TEMPLATE_DEBUG`
- `TEMPLATE_DIRS`
- `TEMPLATE_LOADERS`
- `TEMPLATE_STRING_IF_INVALID`

`django.core.context_processors` Built-in template context processors have been moved to `django.template.context_processors`.

`django.test.SimpleTestCase.urls` The attribute `SimpleTestCase.urls` for specifying URLconf configuration in tests has been deprecated and will be removed in Django 1.10. Use *`@override_settings(ROOT_URLCONF=...)`* instead.

`prefix` argument to `i18n_patterns()` Related to the previous item, the `prefix` argument to *`django.conf.urls.i18n.i18n_patterns()`* has been deprecated. Simply pass a list of *`django.conf.urls.url()`* instances instead.

Using an incorrect count of unpacked values in the `for` template tag Using an incorrect count of unpacked values in *`for`* tag will raise an exception rather than fail silently in Django 1.10.

Passing a dotted path to `reverse()` and `url` Reversing URLs by Python path is an expensive operation as it causes the path being reversed to be imported. This behavior has also resulted in a security issue. Use *named URL patterns* for reversing instead.

If you are using *`django.contrib.sitemaps`*, add the `name` argument to the `url` that references *`django.contrib.sitemaps.views.sitemap()`*:

```
from django.contrib.sitemaps.views import sitemap

url(r'^sitemap\.xml$', sitemap, {'sitemaps': sitemaps},
    name='django.contrib.sitemaps.views.sitemap')
```

to ensure compatibility when reversing by Python path is removed in Django 1.10.

Similarly for GIS sitemaps, add `name='django.contrib.gis.sitemaps.views.kml'` or `name='django.contrib.gis.sitemaps.views.kmz'`.

If you are using a Python path for the *`LOGIN_URL`* or *`LOGIN_REDIRECT_URL`* setting, use the name of the `url()` instead.

Aggregate methods and modules The `django.db.models.sql.aggregates` and `django.contrib.gis.db.models.sql.aggregates` modules (both private API), have been deprecated as `django.db.models.aggregates` and `django.contrib.gis.db.models.aggregates` are now also responsible for SQL generation. The old modules will be removed in Django 1.10.

If you were using the old modules, see Query Expressions for instructions on rewriting custom aggregates using the new stable API.

The following methods and properties of `django.db.models.sql.query.Query` have also been deprecated and the backwards compatibility shims will be removed in Django 1.10:

- `Query.aggregates`, replaced by `annotations`.
- `Query.aggregate_select`, replaced by `annotation_select`.
- `Query.add_aggregate()`, replaced by `add_annotation()`.
- `Query.set_aggregate_mask()`, replaced by `set_annotation_mask()`.
- `Query.append_aggregate_mask()`, replaced by `append_annotation_mask()`.

Extending management command arguments through `Command.option_list` Management commands now use `argparse` instead of `optparse` to parse command-line arguments passed to commands. This also means that the way to add custom arguments to commands has changed: instead of extending the `option_list` class list, you should now override the *`add_arguments()`* method and add arguments through `argparse.add_argument()`. See *this example* for more details.

`django.core.management.NoArgsCommand` The class `NoArgsCommand` is now deprecated and will be removed in Django 1.10. Use *`BaseCommand`* instead, which takes no arguments by default.

Listing all migrations in a project The `--list` option of the *`migrate`* management command is deprecated and will be removed in Django 1.10. Use *`showmigrations`* instead.

cache_choices option of ModelChoiceField and ModelMultipleChoiceField *ModelChoiceField* and *ModelMultipleChoiceField* took an undocumented, untested option `cache_choices`. This cached querysets between multiple renderings of the same `Form` object. This option is subject to an accelerated deprecation and will be removed in Django 1.9.

django.template.resolve_variable() The function has been informally marked as "Deprecated" for some time. Replace `resolve_variable(path, context)` with `django.template.Variable(path).resolve(context)`.

django.contrib.webdesign It provided the *lorem* template tag which is now included in the built-in tags. Simply remove `'django.contrib.webdesign'` from *INSTALLED_APPS* and `{% load webdesign %}` from your templates.

error_message argument to django.forms.RegexField It provided backwards compatibility for pre-1.0 code, but its functionality is redundant. Use `Field.error_messages['invalid']` instead.

Old unordered_list syntax An older (pre-1.0), more restrictive and verbose input format for the *unordered_list* template filter has been deprecated:

```
``['States', [['Kansas', [['Lawrence', []], ['Topeka', []]]], ['Illinois', []]]]``
```

Using the new syntax, this becomes:

```
``['States', ['Kansas', ['Lawrence', 'Topeka'], 'Illinois']]``
```

django.forms.Field._has_changed() Rename this method to *has_changed()* by removing the leading underscore. The old name will still work until Django 1.10.

django.utils.html.remove_tags() and removetags template filter `django.utils.html.remove_tags()` as well as the template filter `removetags` have been deprecated as they cannot guarantee safe output. Their existence is likely to lead to their use in security-sensitive contexts where they are not actually safe.

The unused and undocumented `django.utils.html.strip_entities()` function has also been deprecated.

is_admin_site argument to django.contrib.auth.views.password_reset() It's a legacy option that should no longer be necessary.

SubfieldBase `django.db.models.fields.subclassing.SubfieldBase` has been deprecated and will be removed in Django 1.10. Historically, it was used to handle fields where type conversion was needed when loading from the database, but it was not used in `.values()` calls or in aggregates. It has been replaced with *from_db_value()*.

The new approach doesn't call the *to_python()* method on assignment as was the case with `SubfieldBase`. If you need that behavior, reimplement the `Creator` class from Django's source code in your project.

django.utils.checksums The `django.utils.checksums` module has been deprecated and will be removed in Django 1.10. The functionality it provided (validating checksum using the Luhn algorithm) was undocumented and not used in Django. The module has been moved to the django-localflavor package (version 1.1+).

`InlineAdminForm.original_content_type_id` The `original_content_type_id` attribute on `InlineAdminForm` has been deprecated and will be removed in Django 1.10. Historically, it was used to construct the "view on site" URL. This URL is now accessible using the `absolute_url` attribute of the form.

`django.views.generic.edit.FormMixin.get_form()`'s `form_class` argument `FormMixin` subclasses that override the `get_form()` method should make sure to provide a default value for the `form_class` argument since it's now optional.

Rendering templates loaded by `get_template()` with a `Context` The return type of *`get_template()`* has changed in Django 1.8: instead of a *`django.template.Template`*, it returns a `Template` instance whose exact type depends on which backend loaded it.

Both classes provide a `render()` method, however, the former takes a *`django.template.Context`* as an argument while the latter expects a `dict`. This change is enforced through a deprecation path for Django templates.

Since it's easier to understand with examples, the *upgrade guide* shows how to adapt affected code.

All this also applies to *`select_template()`*.

`Template` and `Context` classes in template responses Some methods of *`SimpleTemplateResponse`* and *`TemplateResponse`* accepted *`django.template.Context`* and *`django.template.Template`* objects as arguments. They should now receive `dict` and backend-dependent template objects respectively.

This also applies to the return types if you have subclassed either template response class.

Check the template response API documentation for details.

`current_app` argument of template-related APIs The following functions and classes will no longer accept a `current_app` parameter to set an URL namespace in Django 1.10:

- `django.shortcuts.render()`
- `django.template.Context()`
- `django.template.RequestContext()`
- `django.template.response.TemplateResponse()`

Set `request.current_app` instead, where `request` is the first argument to these functions or classes. If you're using a plain `Context`, use a `RequestContext` instead.

`dictionary` and `context_instance` arguments of rendering functions The following functions will no longer accept the `dictionary` and `context_instance` parameters in Django 1.10:

- `django.shortcuts.render()`
- `django.shortcuts.render_to_response()`
- `django.template.loader.render_to_string()`

Use the `context` parameter instead. When `dictionary` is passed as a positional argument, which is the most common idiom, no changes are needed.

If you're passing a *`Context`* in `context_instance`, pass a `dict` in the `context` parameter instead. If you're passing a *`RequestContext`*, pass the request separately in the `request` parameter.

`dirs` argument of template-finding functions The following functions will no longer accept a `dirs` parameter to override `TEMPLATE_DIRS` in Django 1.10:

- *`django.template.loader.get_template()`*
- *`django.template.loader.select_template()`*
- *`django.shortcuts.render()`*
- `django.shortcuts.render_to_response()`

The parameter didn't work consistently across different template loaders and didn't work for included templates.

`django.template.loader.BaseLoader` `django.template.loader.BaseLoader` was renamed to `django.template.loaders.base.Loader`. If you've written a custom template loader that inherits `BaseLoader`, you must inherit `Loader` instead.

`django.test.utils.TestTemplateLoader` Private API `django.test.utils.TestTemplateLoader` is deprecated in favor of `django.template.loaders.locmem.Loader` and will be removed in Django 1.9.

Support for the `max_length` argument on custom `Storage` classes `Storage` subclasses should add `max_length=None` as a parameter to *`get_available_name()`* and/or *`save()`* if they override either method. Support for storages that do not accept this argument will be removed in Django 1.10.

`qn` replaced by `compiler` In previous Django versions, various internal ORM methods (mostly `as_sql` methods) accepted a `qn` (for "quote name") argument, which was a reference to a function that quoted identifiers for sending to the database. In Django 1.8, that argument has been renamed to `compiler` and is now a full `SQLCompiler` instance. For backwards-compatibility, calling a `SQLCompiler` instance performs the same name-quoting that the `qn` function used to. However, this backwards-compatibility shim is immediately deprecated: you should rename your `qn` arguments to `compiler`, and call `compiler.quote_name_unless_alias(...)` where you previously called `qn(...)`.

Default value of `RedirectView.permanent` The default value of the *`RedirectView.permanent`* attribute will change from `True` to `False` in Django 1.9.

Using `AuthenticationMiddleware` without `SessionAuthenticationMiddleware` `django.contrib.auth.middleware.SessionAuthenticationMiddleware` was added in Django 1.7. In Django 1.7.2, its functionality was moved to `auth.get_user()` and, for backwards compatibility, enabled only if `'django.contrib.auth.middleware.SessionAuthenticationMiddleware'` appears in *`MIDDLEWARE_CLASSES`*.

In Django 1.10, session verification will be enabled regardless of whether or not `SessionAuthenticationMiddleware` is enabled (at which point `SessionAuthenticationMiddleware` will have no significance). You can add it to your `MIDDLEWARE_CLASSES` sometime before then to opt-in. Please read the *upgrade considerations* first.

`django.contrib.sitemaps.FlatPageSitemap` `django.contrib.sitemaps.FlatPageSitemap` has moved to `django.contrib.flatpages.sitemaps.FlatPageSitemap`. The old import location is deprecated and will be removed in Django 1.9.

Model `Field.related` Private attribute `django.db.models.Field.related` is deprecated in favor of `Field.rel`. The latter is an instance of `django.db.models.fields.related.ForeignObjectRel` which replaces `django.db.models.related.RelatedObject`. The `django.db.models.related` module has been removed and the `Field.related` attribute will be removed in Django 1.10.

`ssi` template tag The `ssi` template tag allows files to be included in a template by absolute path. This is of limited use in most deployment situations, and the *`include`* tag often makes more sense. This tag is now deprecated and will be removed in Django 1.10.

`=` as comparison operator in `if` template tag Using a single equals sign with the `{% if %}` template tag for equality testing was undocumented and untested. It's now deprecated in favor of ==.

`%(<foo>)s` syntax in `ModelFormMixin.success_url` The legacy `%(<foo>)s` syntax in *`ModelFormMixin.success_url`* is deprecated and will be removed in Django 1.10.

`GeoQuerySet` aggregate methods The `collect()`, `extent()`, `extent3d()`, `make_line()`, and `unionagg()` aggregate methods are deprecated and should be replaced by their function-based aggregate equivalents (`Collect`, `Extent`, `Extent3D`, `MakeLine`, and `Union`).

Signature of the `allow_migrate` router method The signature of the *`allow_migrate()`* method of database routers has changed from `allow_migrate(db, model)` to `allow_migrate(db, app_label, model_name=None, **hints)`.

When `model_name` is set, the value that was previously given through the `model` positional argument may now be found inside the `hints` dictionary under the key `'model'`.

After switching to the new signature the router will also be called by the *`RunPython`* and *`RunSQL`* operations.

Features removed in 1.8

These features have reached the end of their deprecation cycle and are removed in Django 1.8. See *Features deprecated in 1.6* for details, including how to remove usage of these features.

- `django.contrib.comments` is removed.
- The following transaction management APIs are removed:
  - `TransactionMiddleware`
  - the decorators and context managers `autocommit`, `commit_on_success`, and `commit_manually`, defined in `django.db.transaction`
  - the functions `commit_unless_managed` and `rollback_unless_managed`, also defined in `django.db.transaction`
  - the `TRANSACTIONS_MANAGED` setting
- The *`cycle`* and *`firstof`* template tags auto-escape their arguments.
- The `SEND_BROKEN_LINK_EMAILS` setting is removed.
- `django.middleware.doc.XViewMiddleware` is removed.
- The `Model._meta.module_name` alias is removed.

- The backward compatible shims introduced to rename `get_query_set` and similar queryset methods are removed. This affects the following classes: `BaseModelAdmin`, `ChangeList`, `BaseCommentNode`, `GenericForeignKey`, `Manager`, `SingleRelatedObjectDescriptor` and `ReverseSingleRelatedObjectDescriptor`.
- The backward compatible shims introduced to rename the attributes `ChangeList.root_query_set` and `ChangeList.query_set` are removed.
- `django.views.defaults.shortcut` and `django.conf.urls.shortcut` are removed.
- Support for the Python Imaging Library (PIL) module is removed.
- The following private APIs are removed:
  - `django.db.backend`
  - `django.db.close_connection()`
  - `django.db.backends.creation.BaseDatabaseCreation.set_autocommit()`
  - `django.db.transaction.is_managed()`
  - `django.db.transaction.managed()`
- `django.forms.widgets.RadioInput` is removed.
- The module `django.test.simple` and the class `django.test.simple.DjangoTestSuiteRunner` are removed.
- The module `django.test._doctest` is removed.
- The `CACHE_MIDDLEWARE_ANONYMOUS_ONLY` setting is removed. This change affects both `django.middleware.cache.CacheMiddleware` and `django.middleware.cache.UpdateCacheMiddleware` despite the lack of a deprecation warning in the latter class.
- Usage of the hard-coded *Hold down "Control", or "Command" on a Mac, to select more than one.* string to override or append to user-provided `help_text` in forms for `ManyToMany` model fields is not performed by Django anymore either at the model or forms layer.
- The `Model._meta.get_(add|change|delete)_permission` methods are removed.
- The session key `django_language` is no longer read for backwards compatibility.
- Geographic Sitemaps are removed (`django.contrib.gis.sitemaps.views.index` and `django.contrib.gis.sitemaps.views.sitemap`).
- `django.utils.html.fix_ampersands`, the `fix_ampersands` template filter, and `django.utils.html.clean_html` are removed.

9.1.4 1.7 release

Django 1.7.11 release notes

November 24, 2015

Django 1.7.11 fixes a security issue and a data loss bug in 1.7.10.

Fixed settings leak possibility in `date` template filter

If an application allows users to specify an unvalidated format for dates and passes this format to the *date* filter, e.g. `{{ last_updated|date:user_date_format }}`, then a malicious user could obtain any secret in the application's settings by specifying a settings key instead of a date format. e.g. `"SECRET_KEY"` instead of `"j/m/Y"`.

To remedy this, the underlying function used by the `date` template filter, `django.utils.formats.get_format()`, now only allows accessing the date/time formatting settings.

Bugfixes

- Fixed a data loss possibility with *Prefetch* if `to_attr` is set to a `ManyToManyField` (#25693).

Django 1.7.10 release notes

August 18, 2015

Django 1.7.10 fixes a security issue in 1.7.9.

Denial-of-service possibility in `logout()` view by filling session store

Previously, a session could be created when anonymously accessing the *django.contrib.auth.views.logout()* view (provided it wasn't decorated with *login_required()* as done in the admin). This could allow an attacker to easily create many new session records by sending repeated requests, potentially filling up the session store or causing other users' session records to be evicted.

The *SessionMiddleware* has been modified to no longer create empty session records, including when *SESSION_SAVE_EVERY_REQUEST* is active.

Additionally, the `contrib.sessions.backends.base.SessionBase.flush()` and `cache_db.SessionStore.flush()` methods have been modified to avoid creating a new empty session. Maintainers of third-party session backends should check if the same vulnerability is present in their backend and correct it if so.

Django 1.7.9 release notes

July 8, 2015

Django 1.7.9 fixes several security issues and bugs in 1.7.8.

Denial-of-service possibility by filling session store

In previous versions of Django, the session backends created a new empty record in the session storage anytime `request.session` was accessed and there was a session key provided in the request cookies that didn't already have a session record. This could allow an attacker to easily create many new session records simply by sending repeated requests with unknown session keys, potentially filling up the session store or causing other users' session records to be evicted.

The built-in session backends now create a session record only if the session is actually modified; empty session records are not created. Thus this potential DoS is now only possible if the site chooses to expose a session-modifying view to anonymous users.

As each built-in session backend was fixed separately (rather than a fix in the core sessions framework), maintainers of third-party session backends should check whether the same vulnerability is present in their backend and correct it if so.

Header injection possibility since validators accept newlines in input

Some of Django's built-in validators (*EmailValidator*, most seriously) didn't prohibit newline characters (due to the usage of `$` instead of `\Z` in the regular expressions). If you use values with newlines in HTTP response or email headers, you can suffer from header injection attacks. Django itself isn't vulnerable because *HttpResponse* and the mail sending utilities in *django.core.mail* prohibit newlines in HTTP and SMTP headers, respectively. While the validators have been fixed in Django, if you're creating HTTP responses or email messages in other ways, it's a good idea to ensure that those methods prohibit newlines as well. You might also want to validate that any existing data in your application doesn't contain unexpected newlines.

validate_ipv4_address(), *validate_slug()*, and *URLValidator* are also affected, however, as of Django 1.6 the `GenericIPAddresseField`, `IPAddressField`, `SlugField`, and `URLField` form fields which use these validators all strip the input, so the possibility of newlines entering your data only exists if you are using these validators outside of the form fields.

The undocumented, internally unused `validate_integer()` function is now stricter as it validates using a regular expression instead of simply casting the value using `int()` and checking if an exception was raised.

Bugfixes

- Prevented the loss of `null`/`not null` column properties during field renaming of MySQL databases (#24817).
- Fixed `SimpleTestCase.assertRaisesMessage()` on Python 2.7.10 (#24903).

Django 1.7.8 release notes

May 1, 2015

Django 1.7.8 fixes:

- Database introspection with SQLite 3.8.9 (released April 8, 2015) (#24637).
- A database table name quoting regression in 1.7.2 (#24605).
- The loss of `null`/`not null` column properties during field alteration of MySQL databases (#24595).

Django 1.7.7 release notes

March 18, 2015

Django 1.7.7 fixes several bugs and security issues in 1.7.6.

Denial-of-service possibility with `strip_tags()`

Last year *strip_tags()* was changed to work iteratively. The problem is that the size of the input it's processing can increase on each iteration which results in an infinite loop in `strip_tags()`. This issue only affects versions of Python that haven't received a bugfix in HTMLParser; namely Python < 2.7.7 and 3.3.5. Some operating system vendors have also backported the fix for the Python bug into their packages of earlier versions.

To remedy this issue, `strip_tags()` will now return the original input if it detects the length of the string it's processing increases. Remember that absolutely NO guarantee is provided about the results of `strip_tags()` being HTML safe. So NEVER mark safe the result of a `strip_tags()` call without escaping it first, for example with *`escape()`*.

Mitigated possible XSS attack via user-supplied redirect URLs

Django relies on user input in some cases (e.g. *`django.contrib.auth.views.login()`* and i18n) to redirect the user to an "on success" URL. The security checks for these redirects (namely `django.utils.http.is_safe_url()`) accepted URLs with leading control characters and so considered URLs like `\x08javascript:...` safe. This issue doesn't affect Django currently, since we only put this URL into the `Location` response header and browsers seem to ignore JavaScript there. Browsers we tested also treat URLs prefixed with control characters such as `%08//example.com` as relative paths so redirection to an unsafe target isn't a problem either.

However, if a developer relies on `is_safe_url()` to provide safe redirect targets and puts such a URL into a link, they could suffer from an XSS attack as some browsers such as Google Chrome ignore control characters at the start of a URL in an anchor `href`.

Bugfixes

- Fixed renaming of classes in migrations where renaming a subclass would cause incorrect state to be recorded for objects that referenced the superclass (#24354).
- Stopped writing migration files in dry run mode when merging migration conflicts. When `makemigrations --merge` is called with `verbosity=3` the migration file is written to `stdout` (#24427).

Django 1.7.6 release notes

March 9, 2015

Django 1.7.6 fixes a security issue and several bugs in 1.7.5.

Mitigated an XSS attack via properties in `ModelAdmin.readonly_fields`

The *`ModelAdmin.readonly_fields`* attribute in the Django admin allows displaying model fields and model attributes. While the former were correctly escaped, the latter were not. Thus untrusted content could be injected into the admin, presenting an exploitation vector for XSS attacks.

In this vulnerability, every model attribute used in `readonly_fields` that is not an actual model field (e.g. a `property`) will **fail to be escaped** even if that attribute is not marked as safe. In this release, autoescaping is now correctly applied.

Bugfixes

- Fixed crash when coercing `ManyRelatedManager` to a string (#24352).
- Fixed a bug that prevented migrations from adding a foreign key constraint when converting an existing field to a foreign key (#24447).

Django 1.7.5 release notes

February 25, 2015

Django 1.7.5 fixes several bugs in 1.7.4.

Bugfixes

- Reverted a fix that prevented a migration crash when unapplying `contrib.contenttypes`'s or `contrib.auth`'s first migration (#24075) due to severe impact on the test performance (#24251) and problems in multi-database setups (#24298).
- Fixed a regression that prevented custom fields inheriting from `ManyToManyField` from being recognized in migrations (#24236).
- Fixed crash in `contrib.sites` migrations when a default database isn't used (#24332).
- Added the ability to set the isolation level on PostgreSQL with psycopg2 $\geq$ 2.4.2 (#24318). It was advertised as a new feature in Django 1.6 but it didn't work in practice.
- Formats for the Azerbaijani locale (`az`) have been added.

Django 1.7.4 release notes

January 27, 2015

Django 1.7.4 fixes several bugs in 1.7.3.

Bugfixes

- Fixed a migration crash when unapplying `contrib.contenttypes`'s or `contrib.auth`'s first migration (#24075).
- Made the migration's `RenameModel` operation rename `ManyToManyField` tables (#24135).
- Fixed a migration crash on MySQL when migrating from a `OneToOneField` to a `ForeignKey` (#24163).
- Prevented the `static.serve` view from producing `ResourceWarnings` in certain circumstances (security fix regression, #24193).
- Fixed schema check for ManyToManyField to look for internal type instead of checking class instance, so you can write custom m2m-like fields with the same behavior. (#24104).

Django 1.7.3 release notes

January 13, 2015

Django 1.7.3 fixes several security issues and bugs in 1.7.2.

WSGI header spoofing via underscore/dash conflation

When HTTP headers are placed into the WSGI environ, they are normalized by converting to uppercase, converting all dashes to underscores, and prepending *HTTP_*. For instance, a header `X-Auth-User` would become `HTTP_X_AUTH_USER` in the WSGI environ (and thus also in Django's `request.META` dictionary).

Unfortunately, this means that the WSGI environ cannot distinguish between headers containing dashes and headers containing underscores: `X-Auth-User` and `X-Auth_User` both become `HTTP_X_AUTH_USER`. This means that if a header is used in a security-sensitive way (for instance, passing authentication information along from a front-end proxy), even if the proxy carefully strips any incoming value for `X-Auth-User`, an attacker may be able to provide an `X-Auth_User` header (with underscore) and bypass this protection.

In order to prevent such attacks, both Nginx and Apache 2.4+ strip all headers containing underscores from incoming requests by default. Django's built-in development server now does the same. Django's development server is not recommended for production use, but matching the behavior of common production servers reduces the surface area for behavior changes during deployment.

Mitigated possible XSS attack via user-supplied redirect URLs

Django relies on user input in some cases (e.g. *`django.contrib.auth.views.login()`* and i18n) to redirect the user to an "on success" URL. The security checks for these redirects (namely `django.utils.http.is_safe_url()`) didn't strip leading whitespace on the tested URL and as such considered URLs like `\njavascript:...` safe. If a developer relied on `is_safe_url()` to provide safe redirect targets and put such a URL into a link, they could suffer from a XSS attack. This bug doesn't affect Django currently, since we only put this URL into the `Location` response header and browsers seem to ignore JavaScript there.

Denial-of-service attack against `django.views.static.serve`

In older versions of Django, the *`django.views.static.serve()`* view read the files it served one line at a time. Therefore, a big file with no newlines would result in memory usage equal to the size of that file. An attacker could exploit this and launch a denial-of-service attack by simultaneously requesting many large files. This view now reads the file in chunks to prevent large memory usage.

Note, however, that this view has always carried a warning that it is not hardened for production use and should be used only as a development aid. Now may be a good time to audit your project and serve your files in production using a real front-end web server if you are not doing so.

Database denial-of-service with `ModelMultipleChoiceField`

Given a form that uses `ModelMultipleChoiceField` and `show_hidden_initial=True` (not a documented API), it was possible for a user to cause an unreasonable number of SQL queries by submitting duplicate values for the field's data. The validation logic in `ModelMultipleChoiceField` now deduplicates submitted values to address this issue.

Bugfixes

- The default iteration count for the PBKDF2 password hasher has been increased by 25%. This part of the normal major release process was inadvertently omitted in 1.7. This backwards compatible change will not affect users who have subclassed `django.contrib.auth.hashers.PBKDF2PasswordHasher` to change the default value.
- Fixed a crash in the CSRF middleware when handling non-ASCII referer header (#23815).
- Fixed a crash in the `django.contrib.auth.redirect_to_login` view when passing a `reverse_lazy()` result on Python 3 (#24097).
- Added correct formats for Greek (`el`) (#23967).

- Fixed a migration crash when unapplying a migration where multiple operations interact with the same model (#24110).

Django 1.7.2 release notes

January 2, 2015

Django 1.7.2 fixes several bugs in 1.7.1.

Additionally, Django's vendored version of six, *django.utils.six*, has been upgraded to the latest release (1.9.0).

Bugfixes

- Fixed migration's renaming of auto-created many-to-many tables when changing *Meta.db_table* (#23630).
- Fixed a migration crash when adding an explicit `id` field to a model on SQLite (#23702).
- Added a warning for duplicate models when a module is reloaded. Previously a `RuntimeError` was raised every time two models clashed in the app registry. (#23621).
- Prevented *flush* from loading initial data for migrated apps (#23699).
- Fixed a *makemessages* regression in 1.7.1 when *STATIC_ROOT* has the default `None` value (#23717).
- Added GeoDjango compatibility with mysqlclient database driver.
- Fixed MySQL 5.6+ crash with `GeometryFields` in migrations (#23719).
- Fixed a migration crash when removing a field that is referenced in `AlterIndexTogether` or `AlterUniqueTogether` (#23614).
- Updated the first day of the week in the Ukrainian locale to Monday.
- Added support for transactional spatial metadata initialization on SpatiaLite 4.1+ (#23152).
- Fixed a migration crash that prevented changing a nullable field with a default to non-nullable with the same default (#23738).
- Fixed a migration crash when adding `GeometryFields` with `blank=True` on PostGIS (#23731).
- Allowed usage of `DateTimeField()` as `Transform.output_field` (#23420).
- Fixed a migration serializing bug involving `float("nan")` and `float("inf")` (#23770).
- Fixed a regression where custom form fields having a `queryset` attribute but no `limit_choices_to` could not be used in a *ModelForm* (#23795).
- Fixed a custom field type validation error with MySQL backend when `db_type` returned `None` (#23761).
- Fixed a migration crash when a field is renamed that is part of an `index_together` (#23859).
- Fixed *squashmigrations* to respect the `--no-optimize` parameter (#23799).
- Made *RenameModel* reversible (#22248)
- Avoided unnecessary rollbacks of migrations from other apps when migrating backwards (#23410).
- Fixed a rare query error when using deeply nested subqueries (#23605).
- Fixed a crash in migrations when deleting a field that is part of a `index/unique_together` constraint (#23794).

- Fixed `django.core.files.File.__repr__()` when the file's `name` contains Unicode characters (#23888).
- Added missing context to the admin's `delete_selected` view that prevented custom site header, etc. from appearing (#23898).
- Fixed a regression with dynamically generated inlines and allowed field references in the admin (#23754).
- Fixed an infinite loop bug for certain cyclic migration dependencies, and made the error message for cyclic dependencies much more helpful.
- Added missing `index_together` handling for SQLite (#23880).
- Fixed a crash when `RunSQL` SQL content was collected by the schema editor, typically when using `sqlmigrate` (#23909).
- Fixed a regression in `contrib.admin` add/change views which caused some `ModelAdmin` methods to receive the incorrect `obj` value (#23934).
- Fixed `runserver` crash when socket error message contained Unicode characters (#23946).
- Fixed serialization of `type` when adding a `deconstruct()` method (#23950).
- Prevented the `django.contrib.auth.middleware.SessionAuthenticationMiddleware` from setting a `"Vary: Cookie"` header on all responses (#23939).
- Fixed a crash when adding `blank=True` to `TextField()` on MySQL (#23920).
- Fixed index creation by the migration infrastructure, particularly when dealing with PostgreSQL specific `{text|varchar}_pattern_ops` indexes (#23954).
- Fixed bug in `makemigrations` that created broken migration files when dealing with multiple table inheritance and inheriting from more than one model (#23956).
- Fixed a crash when a `MultiValueField` has invalid data (#23674).
- Fixed a crash in the admin when using "Save as new" and also deleting a related inline (#23857).
- Always converted `related_name` to text (unicode), since that is required on Python 3 for interpolation. Removed conversion of `related_name` to text in migration deconstruction (#23455 and #23982).
- Enlarged the sizes of tablespaces which are created by default for testing on Oracle (the main tablespace was increased from 200M to 300M and the temporary tablespace from 100M to 150M). This was required to accommodate growth in Django's own test suite (#23969).
- Fixed `timesince` filter translations in Korean (#23989).
- Fixed the SQLite `SchemaEditor` to properly add defaults in the absence of a user specified `default`. For example, a `CharField` with `blank=True` didn't set existing rows to an empty string which resulted in a crash when adding the `NOT NULL` constraint (#23987).
- `makemigrations` no longer prompts for a default value when adding `TextField()` or `CharField()` without a `default` (#23405).
- Fixed a migration crash when adding `order_with_respect_to` to a table with existing rows (#23983).
- Restored the `pre_migrate` signal if all apps have migrations (#23975).
- Made admin system checks run for custom `AdminSites` (#23497).
- Ensured the app registry is fully populated when unpickling models. When an external script (like a queueing infrastructure) reloads pickled models, it could crash with an `AppRegistryNotReady` exception (#24007).
- Added quoting to field indexes in the SQL generated by migrations to prevent a crash when the index name requires it (##24015).

- Added `datetime.time` support to migrations questioner (#23998).
- Fixed admindocs crash on apps installed as eggs (#23525).
- Changed migrations autodetector to generate an `AlterModelOptions` operation instead of `DeleteModel` and `CreateModel` operations when changing `Meta.managed`. This prevents data loss when changing `managed` from `False` to `True` and vice versa (#24037).
- Enabled the `sqlsequencereset` command on apps with migrations (#24054).
- Added tablespace SQL to apps with migrations (#24051).
- Corrected `contrib.sites` default site creation in a multiple database setup (#24000).
- Restored support for objects that aren't `str` or `bytes` in `django.utils.safestring.mark_for_escaping()` on Python 3.
- Supported strings escaped by third-party libraries with the `__html__` convention in the template engine (#23831).
- Prevented extraneous `DROP DEFAULT` SQL in migrations (#23581).
- Restored the ability to use more than five levels of subqueries (#23758).
- Fixed crash when `ValidationError` is initialized with a `ValidationError` that is initialized with a dictionary (#24008).
- Prevented a crash on apps without migrations when running `migrate --list` (#23366).

Django 1.7.1 release notes

October 22, 2014

Django 1.7.1 fixes several bugs in 1.7.

Bugfixes

- Allowed related many-to-many fields to be referenced in the admin (#23604).
- Added a more helpful error message if you try to migrate an app without first creating the `contenttypes` table (#22411).
- Modified migrations dependency algorithm to avoid possible infinite recursion.
- Fixed a `UnicodeDecodeError` when the `flush` error message contained Unicode characters (#22882).
- Reinstated missing `CHECK` SQL clauses which were omitted on some backends when not using migrations (#23416).
- Fixed serialization of `type` objects in migrations (#22951).
- Allowed inline and hidden references to admin fields (#23431).
- The `@deconstructible` decorator now fails with a `ValueError` if the decorated object cannot automatically be imported (#23418).
- Fixed a typo in an `inlineformset_factory()` error message that caused a crash (#23451).
- Restored the ability to use *`ABSOLUTE_URL_OVERRIDES`* with the `'auth.User'` model (#11775). As a side effect, the setting now adds a `get_absolute_url()` method to any model that appears in `ABSOLUTE_URL_OVERRIDES` but doesn't define `get_absolute_url()`.
- Avoided masking some *ImportError* exceptions during application loading (#22920).

- Empty `index_together` or `unique_together` model options no longer results in infinite migrations (#23452).
- Fixed crash in `contrib.sitemaps` if `lastmod` returned a `date` rather than a `datetime` (#23403).
- Allowed migrations to work with `app_labels` that have the same last part (e.g. `django.contrib.auth` and `vendor.auth`) (#23483).
- Restored the ability to deepcopy `F` objects (#23492).
- Formats for Welsh (`cy`) and several Chinese locales (`zh_CN`, `zh_Hans`, `zh_Hant` and `zh_TW`) have been added. Formats for Macedonian have been fixed (trailing dot removed, #23532).
- Added quoting of constraint names in the SQL generated by migrations to prevent crash with uppercase characters in the name (#23065).
- Fixed renaming of models with a self-referential many-to-many field (`ManyToManyField('self')`) (#23503).
- Added the *`get_extra()`*, *`get_max_num()`*, and *`get_min_num()`* hooks to *`GenericInlineModelAdmin`* (#23539).
- Made `migrations.RunSQL` no longer require percent sign escaping. This is now consistent with `cursor.execute()` (#23426).
- Made the *`SERIALIZE`* entry in the *`TEST`* dictionary usable (#23421).
- Fixed bug in migrations that prevented foreign key constraints to unmanaged models with a custom primary key (#23415).
- Added `SchemaEditor` for MySQL GIS backend so that spatial indexes will be created for apps with migrations (#23538).
- Added `SchemaEditor` for Oracle GIS backend so that spatial metadata and indexes will be created for apps with migrations (#23537).
- Coerced the `related_name` model field option to unicode during migration generation to generate migrations that work with both Python 2 and 3 (#23455).
- Fixed `MigrationWriter` to handle builtin types without imports (#23560).
- Fixed `deepcopy` on `ErrorList` (#23594).
- Made the *`admindocs`* view to browse view details check if the view specified in the URL exists in the URLconf. Previously it was possible to import arbitrary packages from the Python path. This was not considered a security issue because `admindocs` is only accessible to staff users (#23601).
- Fixed `UnicodeDecodeError` crash in `AdminEmailHandler` with non-ASCII characters in the request (#23593).
- Fixed missing `get_or_create` and `update_or_create` on related managers causing `IntegrityError` (#23611).
- Made *`urlsafe_base64_decode()`* return the proper type (byte string) on Python 3 (#23333).
- *`makemigrations`* can now serialize timezone-aware values (#23365).
- Added a prompt to the migrations questioner when removing the null constraint from a field to prevent an IntegrityError on existing NULL rows (#23609).
- Fixed generic relations in `ModelAdmin.list_filter` (#23616).
- Restored RFC compliance for the SMTP backend on Python 3 (#23063).
- Fixed a crash while parsing cookies containing invalid content (#23638).

- The system check framework now raises error **models.E020** when the class method `Model.check()` is unreachable (#23615).
- Made the Oracle test database creation drop the test user in the event of an unclean exit of a previous test run (#23649).
- Fixed *`makemigrations`* to detect changes to *`Meta.db_table`* (#23629).
- Fixed a regression when feeding the Django test client with an empty data string (#21740).
- Fixed a regression in *`makemessages`* where static files were unexpectedly ignored (#23583).

Django 1.7 release notes

September 2, 2014

Welcome to Django 1.7!

These release notes cover the *new features*, as well as some *backwards incompatible changes* you'll want to be aware of when upgrading from Django 1.6 or older versions. We've *begun the deprecation process for some features*, and some features have reached the end of their deprecation process and *have been removed*.

Python compatibility

Django 1.7 requires Python 2.7, 3.2, 3.3, or 3.4. We **highly recommend** and only officially support the latest release of each series.

The Django 1.6 series is the last to support Python 2.6. Django 1.7 is the first release to support Python 3.4.

This change should affect only a small number of Django users, as most operating-system vendors today are shipping Python 2.7 or newer as their default version. If you're still using Python 2.6, however, you'll need to stick to Django 1.6 until you can upgrade your Python version. Per our support policy, Django 1.6 will continue to receive security support until the release of Django 1.8.

What's new in Django 1.7

Schema migrations Django now has built-in support for schema migrations. It allows models to be updated, changed, and deleted by creating migration files that represent the model changes and which can be run on any development, staging or production database.

Migrations are covered in their own documentation, but a few of the key features are:

- `syncdb` has been deprecated and replaced by `migrate`. Don't worry - calls to `syncdb` will still work as before.
- A new `makemigrations` command provides an easy way to autodetect changes to your models and make migrations for them.

  `django.db.models.signals.pre_syncdb` and `django.db.models.signals.post_syncdb` have been deprecated, to be replaced by *`pre_migrate`* and *`post_migrate`* respectively. These new signals have slightly different arguments. Check the documentation for details.
- The `allow_syncdb` method on database routers is now called `allow_migrate`, but still performs the same function. Routers with `allow_syncdb` methods will still work, but that method name is deprecated and you should change it as soon as possible (nothing more than renaming is required).
- `initial_data` fixtures are no longer loaded for apps with migrations; if you want to load initial data for an app, we suggest you create a migration for your application and define a *`RunPython`* or *`RunSQL`* operation in the `operations` section of the migration.

- Test rollback behavior is different for apps with migrations; in particular, Django will no longer emulate rollbacks on non-transactional databases or inside `TransactionTestCase` *unless specifically requested*.
- It is not advised to have apps without migrations depend on (have a *`ForeignKey`* or *`ManyToManyField`* to) apps with migrations.

App-loading refactor Historically, Django applications were tightly linked to models. A singleton known as the "app cache" dealt with both installed applications and models. The models module was used as an identifier for applications in many APIs.

As the concept of Django applications matured, this code showed some shortcomings. It has been refactored into an "app registry" where models modules no longer have a central role and where it's possible to attach configuration data to applications.

Improvements thus far include:

- Applications can run code at startup, before Django does anything else, with the *`ready()`* method of their configuration.
- Application labels are assigned correctly to models even when they're defined outside of `models.py`. You don't have to set *`app_label`* explicitly any more.
- It is possible to omit `models.py` entirely if an application doesn't have any models.
- Applications can be relabeled with the *`label`* attribute of application configurations, to work around label conflicts.
- The name of applications can be customized in the admin with the *`verbose_name`* of application configurations.
- The admin automatically calls *`autodiscover()`* when Django starts. You can consequently remove this line from your URLconf.
- Django imports all application configurations and models as soon as it starts, through a deterministic and straightforward process. This should make it easier to diagnose import issues such as import loops.

New method on Field subclasses To help power both schema migrations and to enable easier addition of composite keys in future releases of Django, the *`Field`* API now has a new required method: `deconstruct()`.

This method takes no arguments, and returns a tuple of four items:

- `name`: The field's attribute name on its parent model, or None if it is not part of a model
- `path`: A dotted, Python path to the class of this field, including the class name.
- `args`: Positional arguments, as a list
- `kwargs`: Keyword arguments, as a dict

These four values allow any field to be serialized into a file, as well as allowing the field to be copied safely, both essential parts of these new features.

This change should not affect you unless you write custom Field subclasses; if you do, you may need to reimplement the `deconstruct()` method if your subclass changes the method signature of `__init__` in any way. If your field just inherits from a built-in Django field and doesn't override `__init__`, no changes are necessary.

If you do need to override `deconstruct()`, a good place to start is the built-in Django fields (`django/db/models/fields/__init__.py`) as several fields, including `DecimalField` and `DateField`, override it and show how to call the method on the superclass and simply add or remove extra arguments.

This also means that all arguments to fields must themselves be serializable; to see what we consider serializable, and to find out how to make your own classes serializable, read the *migration serialization documentation*.

Calling custom `QuerySet` methods from the `Manager` Historically, the recommended way to make reusable model queries was to create methods on a custom `Manager` class. The problem with this approach was that after the first method call, you'd get back a `QuerySet` instance and couldn't call additional custom manager methods.

Though not documented, it was common to work around this issue by creating a custom `QuerySet` so that custom methods could be chained; but the solution had a number of drawbacks:

- The custom `QuerySet` and its custom methods were lost after the first call to `values()` or `values_list()`.
- Writing a custom `Manager` was still necessary to return the custom `QuerySet` class and all methods that were desired on the `Manager` had to be proxied to the `QuerySet`. The whole process went against the DRY principle.

The *`QuerySet.as_manager()`* class method can now directly *create Manager with QuerySet methods*:

```
class FoodQuerySet(models.QuerySet):
    def pizzas(self):
        return self.filter(kind='pizza')

    def vegetarian(self):
        return self.filter(vegetarian=True)

class Food(models.Model):
    kind = models.CharField(max_length=50)
    vegetarian = models.BooleanField(default=False)
    objects = FoodQuerySet.as_manager()

Food.objects.pizzas().vegetarian()
```

Using a custom manager when traversing reverse relations It is now possible to *specify a custom manager* when traversing a reverse relationship:

```
class Blog(models.Model):
    pass

class Entry(models.Model):
    blog = models.ForeignKey(Blog)

    objects = models.Manager()  # Default Manager
    entries = EntryManager()    # Custom Manager

b = Blog.objects.get(id=1)
b.entry_set(manager='entries').all()
```

New system check framework We've added a new System check framework for detecting common problems (like invalid models) and providing hints for resolving those problems. The framework is extensible so you can add your own checks for your own apps and libraries.

To perform system checks, you use the *`check`* management command. This command replaces the older `validate` management command.

New `Prefetch` object for advanced `prefetch_related` operations. The new *`Prefetch`* object allows customizing prefetch operations.

You can specify the `QuerySet` used to traverse a given relation or customize the storage location of prefetch results.

This enables things like filtering prefetched relations, calling *`select_related()`* from a prefetched relation, or prefetching the same relation multiple times with different querysets. See *`prefetch_related()`* for more details.

Admin shortcuts support time zones The "today" and "now" shortcuts next to date and time input widgets in the admin are now operating in the *current time zone*. Previously, they used the browser time zone, which could result in saving the wrong value when it didn't match the current time zone on the server.

In addition, the widgets now display a help message when the browser and server time zone are different, to clarify how the value inserted in the field will be interpreted.

Using database cursors as context managers Prior to Python 2.7, database cursors could be used as a context manager. The specific backend's cursor defined the behavior of the context manager. The behavior of magic method lookups was changed with Python 2.7 and cursors were no longer usable as context managers.

Django 1.7 allows a cursor to be used as a context manager. That is, the following can be used:

```
with connection.cursor() as c:
    c.execute(...)
```

instead of:

```
c = connection.cursor()
try:
    c.execute(...)
finally:
    c.close()
```

Custom lookups It is now possible to write custom lookups and transforms for the ORM. Custom lookups work just like Django's built-in lookups (e.g. `lte`, `icontains`) while transforms are a new concept.

The *`django.db.models.Lookup`* class provides a way to add lookup operators for model fields. As an example it is possible to add `day_lte` operator for `DateFields`.

The *`django.db.models.Transform`* class allows transformations of database values prior to the final lookup. For example it is possible to write a `year` transform that extracts year from the field's value. Transforms allow for chaining. After the `year` transform has been added to `DateField` it is possible to filter on the transformed value, for example `qs.filter(author__birthdate__year__lte=1981)`.

For more information about both custom lookups and transforms refer to the custom lookups documentation.

Improvements to `Form` error handling

`Form.add_error()` Previously there were two main patterns for handling errors in forms:

- Raising a *`ValidationError`* from within certain functions (e.g. `Field.clean()`, `Form.clean_<fieldname>()`, or `Form.clean()` for non-field errors.)
- Fiddling with `Form._errors` when targeting a specific field in `Form.clean()` or adding errors from outside of a "clean" method (e.g. directly from a view).

Using the former pattern was straightforward since the form can guess from the context (i.e. which method raised the exception) where the errors belong and automatically process them. This remains the canonical way of adding errors when possible. However the latter was fiddly and error-prone, since the burden of handling edge cases fell on the user.

The new *add_error()* method allows adding errors to specific form fields from anywhere without having to worry about the details such as creating instances of `django.forms.utils.ErrorList` or dealing with `Form.cleaned_data`. This new API replaces manipulating `Form._errors` which now becomes a private API.

See *Cleaning and validating fields that depend on each other* for an example using `Form.add_error()`.

Error metadata The *ValidationError* constructor accepts metadata such as error `code` or `params` which are then available for interpolating into the error message (see *Raising ValidationError* for more details); however, before Django 1.7 those metadata were discarded as soon as the errors were added to *Form.errors*.

Form.errors and `django.forms.utils.ErrorList` now store the `ValidationError` instances so these metadata can be retrieved at any time through the new *Form.errors.as_data* method.

The retrieved `ValidationError` instances can then be identified thanks to their error `code` which enables things like rewriting the error's message or writing custom logic in a view when a given error is present. It can also be used to serialize the errors in a custom format such as XML.

The new *Form.errors.as_json()* method is a convenience method which returns error messages along with error codes serialized as JSON. `as_json()` uses `as_data()` and gives an idea of how the new system could be extended.

Error containers and backward compatibility Heavy changes to the various error containers were necessary in order to support the features above, specifically *Form.errors*, `django.forms.utils.ErrorList`, and the internal storages of *ValidationError*. These containers which used to store error strings now store `ValidationError` instances and public APIs have been adapted to make this as transparent as possible, but if you've been using private APIs, some of the changes are backwards incompatible; see *ValidationError constructor and internal storage* for more details.

Minor features

`django.contrib.admin`

- You can now implement *site_header*, *site_title*, and *index_title* attributes on a custom *AdminSite* in order to easily change the admin site's page title and header text. No more needing to override templates!
- Buttons in *django.contrib.admin* now use the `border-radius` CSS property for rounded corners rather than GIF background images.
- Some admin templates now have `app-<app_name>` and `model-<model_name>` classes in their `<body>` tag to allow customizing the CSS per app or per model.
- The admin changelist cells now have a `field-<field_name>` class in the HTML to enable style customizations.
- The admin's search fields can now be customized per-request thanks to the new *django.contrib.admin.ModelAdmin.get_search_fields()* method.
- The *ModelAdmin.get_fields()* method may be overridden to customize the value of *ModelAdmin.fields*.
- In addition to the existing `admin.site.register` syntax, you can use the new *register()* decorator to register a *ModelAdmin*.

- You may specify *ModelAdmin.list_display_links* = None to disable links on the change list page grid.
- You may now specify *ModelAdmin.view_on_site* to control whether or not to display the "View on site" link.
- You can specify a descending ordering for a *ModelAdmin.list_display* value by prefixing the admin_order_field value with a hyphen.
- The *ModelAdmin.get_changeform_initial_data()* method may be overridden to define custom behavior for setting initial change form data.

django.contrib.auth

- Any **kwargs passed to *email_user()* are passed to the underlying *send_mail()* call.
- The *permission_required()* decorator can take a list of permissions as well as a single permission.
- You can override the new *AuthenticationForm.confirm_login_allowed()* method to more easily customize the login policy.
- *django.contrib.auth.views.password_reset()* takes an optional html_email_template_name parameter used to send a multipart HTML email for password resets.
- The *AbstractBaseUser.get_session_auth_hash()* method was added and if your *AUTH_USER_MODEL* inherits from *AbstractBaseUser*, changing a user's password now invalidates old sessions if the django.contrib.auth.middleware.SessionAuthenticationMiddleware is enabled. See *Session invalidation on password change* for more details.

django.contrib.formtools

- Calls to WizardView.done() now include a form_dict to allow easier access to forms by their step name.

django.contrib.gis

- The default OpenLayers library version included in widgets has been updated from 2.11 to 2.13.
- Prepared geometries now also support the crosses, disjoint, overlaps, touches and within predicates, if GEOS 3.3 or later is installed.

django.contrib.messages

- The backends for *django.contrib.messages* that use cookies, will now follow the *SESSION_COOKIE_SECURE* and *SESSION_COOKIE_HTTPONLY* settings.
- The *messages context processor* now adds a dictionary of default levels under the name DEFAULT_MESSAGE_LEVELS.
- *Message* objects now have a level_tag attribute that contains the string representation of the message level.

django.contrib.redirects

- *RedirectFallbackMiddleware* has two new attributes (*response_gone_class* and *response_redirect_class*) that specify the types of *HttpResponse* instances the middleware returns.

django.contrib.sessions

- The `"django.contrib.sessions.backends.cached_db"` session backend now respects *SESSION_CACHE_ALIAS*. In previous versions, it always used the *default* cache.

django.contrib.sitemaps

- The *sitemap framework* now makes use of *lastmod* to set a `Last-Modified` header in the response. This makes it possible for the *ConditionalGetMiddleware* to handle conditional `GET` requests for sitemaps which set `lastmod`.

django.contrib.sites

- The new *django.contrib.sites.middleware.CurrentSiteMiddleware* allows setting the current site on each request.

django.contrib.staticfiles

- The *static files storage classes* may be subclassed to override the permissions that collected static files and directories receive by setting the *file_permissions_mode* and *directory_permissions_mode* parameters. See *collectstatic* for example usage.
- The *CachedStaticFilesStorage* backend gets a sibling class called *ManifestStaticFilesStorage* that doesn't use the cache system at all but instead a JSON file called `staticfiles.json` for storing the mapping between the original file name (e.g. `css/styles.css`) and the hashed file name (e.g. `css/styles.55e7cbb9ba48.css`). The `staticfiles.json` file is created when running the *collectstatic* management command and should be a less expensive alternative for remote storages such as Amazon S3.

  See the *ManifestStaticFilesStorage* docs for more information.
- *findstatic* now accepts verbosity flag level 2, meaning it will show the relative paths of the directories it searched. See *findstatic* for example output.

django.contrib.syndication

- The *Atom1Feed* syndication feed's `updated` element now utilizes `updateddate` instead of `pubdate`, allowing the `published` element to be included in the feed (which relies on `pubdate`).

Cache

- Access to caches configured in *CACHES* is now available via *django.core.cache.caches*. This dict-like object provides a different instance per thread. It supersedes `django.core.cache.get_cache()` which is now deprecated.
- If you instantiate cache backends directly, be aware that they aren't thread-safe any more, as *django.core.cache.caches* now yields different instances per thread.
- Defining the *TIMEOUT* argument of the *CACHES* setting as `None` will set the cache keys as "non-expiring" by default. Previously, it was only possible to pass `timeout=None` to the cache backend's `set()` method.

Cross Site Request Forgery

- The *CSRF_COOKIE_AGE* setting facilitates the use of session-based CSRF cookies.

Email

- *send_mail()* now accepts an html_message parameter for sending a multipart text/plain and text/html email.
- The SMTP *EmailBackend* now accepts a timeout parameter.

File Storage

- File locking on Windows previously depended on the PyWin32 package; if it wasn't installed, file locking failed silently. That dependency has been removed, and file locking is now implemented natively on both Windows and Unix.

File Uploads

- The new *UploadedFile.content_type_extra* attribute contains extra parameters passed to the content-type header on a file upload.
- The new *FILE_UPLOAD_DIRECTORY_PERMISSIONS* setting controls the file system permissions of directories created during file upload, like *FILE_UPLOAD_PERMISSIONS* does for the files themselves.
- The *FileField.upload_to* attribute is now optional. If it is omitted or given None or an empty string, a subdirectory won't be used for storing the uploaded files.
- Uploaded files are now explicitly closed before the response is delivered to the client. Partially uploaded files are also closed as long as they are named file in the upload handler.
- *Storage.get_available_name()* now appends an underscore plus a random 7 character alphanumeric string (e.g. "_x3a1gho"), rather than iterating through an underscore followed by a number (e.g. "_1", "_2", etc.) to prevent a denial-of-service attack. This change was also made in the 1.6.6, 1.5.9, and 1.4.14 security releases.

Forms

- The <label> and <input> tags rendered by *RadioSelect* and *CheckboxSelectMultiple* when looping over the radio buttons or checkboxes now include for and id attributes, respectively. Each radio button or checkbox includes an id_for_label attribute to output the element's ID.
- The <textarea> tags rendered by *Textarea* now include a maxlength attribute if the *TextField* model field has a max_length.
- *Field.choices* now allows you to customize the "empty choice" label by including a tuple with an empty string or None for the key and the custom label as the value. The default blank option "----------" will be omitted in this case.
- *MultiValueField* allows optional subfields by setting the require_all_fields argument to False. The required attribute for each individual field will be respected, and a new incomplete validation error will be raised when any required fields are empty.
- The *clean()* method on a form no longer needs to return self.cleaned_data. If it does return a changed dictionary then that will still be used.
- After a temporary regression in Django 1.6, it's now possible again to make *TypedChoiceField* coerce method return an arbitrary value.
- *SelectDateWidget.months* can be used to customize the wording of the months displayed in the select widget.
- The min_num and validate_min parameters were added to *formset_factory()* to allow validating a minimum number of submitted forms.

- The metaclasses used by `Form` and `ModelForm` have been reworked to support more inheritance scenarios. The previous limitation that prevented inheriting from both `Form` and `ModelForm` simultaneously have been removed as long as `ModelForm` appears first in the MRO.
- It's now possible to remove a field from a `Form` when subclassing by setting the name to `None`.
- It's now possible to customize the error messages for `ModelForm`'s `unique`, `unique_for_date`, and `unique_together` constraints. In order to support `unique_together` or any other `NON_FIELD_ERROR`, `ModelForm` now looks for the `NON_FIELD_ERROR` key in the `error_messages` dictionary of the `ModelForm`'s inner `Meta` class. See *considerations regarding model's error_messages* for more details.

Internationalization

- The *`django.middleware.locale.LocaleMiddleware.response_redirect_class`* attribute allows you to customize the redirects issued by the middleware.
- The *`LocaleMiddleware`* now stores the user's selected language with the session key `_language`. This should only be accessed using the *`LANGUAGE_SESSION_KEY`* constant. Previously it was stored with the key `django_language` and the `LANGUAGE_SESSION_KEY` constant did not exist, but keys reserved for Django should start with an underscore. For backwards compatibility `django_language` is still read from in 1.7. Sessions will be migrated to the new key as they are written.
- The *`blocktrans`* tag now supports a `trimmed` option. This option will remove newline characters from the beginning and the end of the content of the `{% blocktrans %}` tag, replace any whitespace at the beginning and end of a line and merge all lines into one using a space character to separate them. This is quite useful for indenting the content of a `{% blocktrans %}` tag without having the indentation characters end up in the corresponding entry in the PO file, which makes the translation process easier.
- When you run *`makemessages`* from the root directory of your project, any extracted strings will now be automatically distributed to the proper app or project message file. See *Localization: how to create language files* for details.
- The *`makemessages`* command now always adds the `--previous` command line flag to the `msgmerge` command, keeping previously translated strings in po files for fuzzy strings.
- The following settings to adjust the language cookie options were introduced: *`LANGUAGE_COOKIE_AGE`*, *`LANGUAGE_COOKIE_DOMAIN`* and *`LANGUAGE_COOKIE_PATH`*.
- Added Format localization for Esperanto.

Management Commands

- The new *`--no-color`* option for `django-admin` disables the colorization of management command output.
- The new *`dumpdata --natural-foreign`* and *`dumpdata --natural-primary`* options, and the new `use_natural_foreign_keys` and `use_natural_primary_keys` arguments for `serializers.serialize()`, allow the use of natural primary keys when serializing.
- It is no longer necessary to provide the cache table name or the `--database` option for the *`createcachetable`* command. Django takes this information from your settings file. If you have configured multiple caches or multiple databases, all cache tables are created.
- The *`runserver`* command received several improvements:
  - On Linux systems, if pyinotify is installed, the development server will reload immediately when a file is changed. Previously, it polled the filesystem for changes every second. That caused a small delay before reloads and reduced battery life on laptops.

  - In addition, the development server automatically reloads when a translation file is updated, i.e. after running *compilemessages*.
  - All HTTP requests are logged to the console, including requests for static files or `favicon.ico` that used to be filtered out.

- Management commands can now produce syntax colored output under Windows if the ANSICON third-party tool is installed and active.
- *collectstatic* command with symlink option is now supported on Windows NT 6 (Windows Vista and newer).
- Initial SQL data now works better if the sqlparse Python library is installed.

  Note that it's deprecated in favor of the *RunSQL* operation of migrations, which benefits from the improved behavior.

Models

- The *QuerySet.update_or_create()* method was added.
- The new *default_permissions* model `Meta` option allows you to customize (or disable) creation of the default add, change, and delete permissions.
- Explicit *OneToOneField* for *Multi-table inheritance* are now discovered in abstract classes.
- It is now possible to avoid creating a backward relation for *OneToOneField* by setting its *related_name* to `'+'` or ending it with `'+'`.
- *F expressions* support the power operator (`**`).
- The `remove()` and `clear()` methods of the related managers created by `ForeignKey` and `GenericForeignKey` now accept the `bulk` keyword argument to control whether or not to perform operations in bulk (i.e. using `QuerySet.update()`). Defaults to `True`.
- It is now possible to use `None` as a query value for the *iexact* lookup.
- It is now possible to pass a callable as value for the attribute *limit_choices_to* when defining a `ForeignKey` or `ManyToManyField`.
- Calling *only()* and *defer()* on the result of *QuerySet.values()* now raises an error (before that, it would either result in a database error or incorrect data).
- You can use a single list for *index_together* (rather than a list of lists) when specifying a single set of fields.
- Custom intermediate models having more than one foreign key to any of the models participating in a many-to-many relationship are now permitted, provided you explicitly specify which foreign keys should be used by setting the new *ManyToManyField.through_fields* argument.
- Assigning a model instance to a non-relation field will now throw an error. Previously this used to work if the field accepted integers as input as it took the primary key.
- Integer fields are now validated against database backend specific min and max values based on their *internal_type*. Previously model field validation didn't prevent values out of their associated column data type range from being saved resulting in an integrity error.
- It is now possible to explicitly *order_by()* a relation `_id` field by using its attribute name.

Signals

- The `enter` argument was added to the *setting_changed* signal.

- The model signals can be now be connected to using a `str` of the `'app_label.ModelName'` form – just like related fields – to lazily reference their senders.

Templates

- The *`Context.push()`* method now returns a context manager which automatically calls *`pop()`* upon exiting the `with` statement. Additionally, *`push()`* now accepts parameters that are passed to the `dict` constructor used to build the new context level.
- The new *`Context.flatten()`* method returns a `Context`'s stack as one flat dictionary.
- `Context` objects can now be compared for equality (internally, this uses *`Context.flatten()`* so the internal structure of each `Context`'s stack doesn't matter as long as their flattened version is identical).
- The *`widthratio`* template tag now accepts an `"as"` parameter to capture the result in a variable.
- The *`include`* template tag will now also accept anything with a `render()` method (such as a `Template`) as an argument. String arguments will be looked up using *`get_template()`* as always.
- It is now possible to *`include`* templates recursively.
- Template objects now have an origin attribute set when `TEMPLATE_DEBUG` is `True`. This allows template origins to be inspected and logged outside of the `django.template` infrastructure.
- `TypeError` exceptions are no longer silenced when raised during the rendering of a template.
- The following functions now accept a `dirs` parameter which is a list or tuple to override `TEMPLATE_DIRS`:
  - *`django.template.loader.get_template()`*
  - *`django.template.loader.select_template()`*
  - *`django.shortcuts.render()`*
  - `django.shortcuts.render_to_response()`
- The *`time`* filter now accepts timezone-related *format specifiers* `'e'`, `'O'` , `'T'` and `'Z'` and is able to digest *time-zone-aware* `datetime` instances performing the expected rendering.
- The *`cache`* tag will now try to use the cache called "template_fragments" if it exists and fall back to using the default cache otherwise. It also now accepts an optional `using` keyword argument to control which cache it uses.
- The new *`truncatechars_html`* filter truncates a string to be no longer than the specified number of characters, taking HTML into account.

Requests and Responses

- The new *`HttpRequest.scheme`* attribute specifies the scheme of the request (`http` or `https` normally).
- The shortcut *`redirect()`* now supports relative URLs.
- The new *`JsonResponse`* subclass of *`HttpResponse`* helps easily create JSON-encoded responses.

Tests

- *`DiscoverRunner`* has two new attributes, *`test_suite`* and *`test_runner`*, which facilitate overriding the way tests are collected and run.
- The `fetch_redirect_response` argument was added to *`assertRedirects()`*. Since the test client can't fetch externals URLs, this allows you to use `assertRedirects` with redirects that aren't part of your Django app.

- Correct handling of scheme when making comparisons in *assertRedirects()*.
- The `secure` argument was added to all the request methods of *Client*. If `True`, the request will be made through HTTPS.
- *assertNumQueries()* now prints out the list of executed queries if the assertion fails.
- The `WSGIRequest` instance generated by the test handler is now attached to the *django.test.Response.wsgi_request* attribute.
- The database settings for testing have been collected into a dictionary named *TEST*.

Utilities

- Improved *strip_tags()* accuracy (but it still cannot guarantee an HTML-safe result, as stated in the documentation).

Validators

- *RegexValidator* now accepts the optional *flags* and Boolean *inverse_match* arguments. The *inverse_match* attribute determines if the *ValidationError* should be raised when the regular expression pattern matches (`True`) or does not match (`False`, by default) the provided `value`. The *flags* attribute sets the flags used when compiling a regular expression string.
- *URLValidator* now accepts an optional `schemes` argument which allows customization of the accepted URI schemes (instead of the defaults `http(s)` and `ftp(s)`).
- *validate_email()* now accepts addresses with IPv6 literals, like `example@[2001:db8::1]`, as specified in RFC 5321.

Backwards incompatible changes in 1.7

: In addition to the changes outlined in this section, be sure to review the *deprecation plan* for any features that have been removed. If you haven't updated your code within the deprecation timeline for a given feature, its removal may appear as a backwards incompatible change.

`allow_syncdb` / `allow_migrate` While Django will still look at `allow_syncdb` methods even though they should be renamed to `allow_migrate`, there is a subtle difference in which models get passed to these methods.

For apps with migrations, `allow_migrate` will now get passed *historical models*, which are special versioned models without custom attributes, methods or managers. Make sure your `allow_migrate` methods are only referring to fields or other items in `model._meta`.

initial_data Apps with migrations will not load `initial_data` fixtures when they have finished migrating. Apps without migrations will continue to load these fixtures during the phase of `migrate` which emulates the old `syncdb` behavior, but any new apps will not have this support.

Instead, you are encouraged to load initial data in migrations if you need it (using the `RunPython` operation and your model classes); this has the added advantage that your initial data will not need updating every time you change the schema.

Additionally, like the rest of Django's old `syncdb` code, `initial_data` has been started down the deprecation path and will be removed in Django 1.9.

deconstruct() and serializability Django now requires all Field classes and all of their constructor arguments to be serializable. If you modify the constructor signature in your custom Field in any way, you'll need to implement a deconstruct() method; we've expanded the custom field documentation with *instructions on implementing this method*.

The requirement for all field arguments to be *serializable* means that any custom class instances being passed into Field constructors - things like custom Storage subclasses, for instance - need to have a *deconstruct method defined on them as well*, though Django provides a handy class decorator that will work for most applications.

App-loading changes

Start-up sequence Django 1.7 loads application configurations and models as soon as it starts. While this behavior is more straightforward and is believed to be more robust, regressions cannot be ruled out. See *Troubleshooting* for solutions to some problems you may encounter.

Standalone scripts If you're using Django in a plain Python script — rather than a management command — and you rely on the *DJANGO_SETTINGS_MODULE* environment variable, you must now explicitly initialize Django at the beginning of your script with:

```
>>> import django
>>> django.setup()
```

Otherwise, you will hit an `AppRegistryNotReady` exception.

WSGI scripts Until Django 1.3, the recommended way to create a WSGI application was:

```
import django.core.handlers.wsgi
application = django.core.handlers.wsgi.WSGIHandler()
```

In Django 1.4, support for WSGI was improved and the API changed to:

```
from django.core.wsgi import get_wsgi_application
application = get_wsgi_application()
```

If you're still using the former style in your WSGI script, you need to upgrade to the latter, or you will hit an `AppRegistryNotReady` exception.

App registry consistency It is no longer possible to have multiple installed applications with the same label. In previous versions of Django, this didn't always work correctly, but didn't crash outright either.

If you have two apps with the same label, you should create an *AppConfig* for one of them and override its *label* there. You should then adjust your code wherever it references this application or its models with the old label.

It isn't possible to import the same model twice through different paths any more. As of Django 1.6, this may happen only if you're manually putting a directory and a subdirectory on `PYTHONPATH`. Refer to the section on the new project layout in the 1.4 release notes for migration instructions.

You should make sure that:

- All models are defined in applications that are listed in *INSTALLED_APPS* or have an explicit *app_label*.
- Models aren't imported as a side-effect of loading their application. Specifically, you shouldn't import models in the root module of an application nor in the module that define its configuration class.

Django will enforce these requirements as of version 1.9, after a deprecation period.

Subclassing AppCommand Subclasses of *AppCommand* must now implement a *handle_app_config()* method instead of `handle_app()`. This method receives an *AppConfig* instance instead of a models module.

Introspecting applications Since *INSTALLED_APPS* now supports application configuration classes in addition to application modules, you should review code that accesses this setting directly and use the app registry (*django.apps.apps*) instead.

The app registry has preserved some features of the old app cache. Even though the app cache was a private API, obsolete methods and arguments will be removed through a standard deprecation path, with the exception of the following changes that take effect immediately:

- `get_model` raises `LookupError` instead of returning `None` when no model is found.
- The `only_installed` argument of `get_model` and `get_models` no longer exists, nor does the `seed_cache` argument of `get_model`.

Management commands and order of `INSTALLED_APPS` When several applications provide management commands with the same name, Django loads the command from the application that comes first in *INSTALLED_APPS*. Previous versions loaded the command from the application that came last.

This brings discovery of management commands in line with other parts of Django that rely on the order of *INSTALLED_APPS*, such as static files, templates, and translations.

`ValidationError` constructor and internal storage The behavior of the `ValidationError` constructor has changed when it receives a container of errors as an argument (e.g. a `list` or an `ErrorList`):

- It converts any strings it finds to instances of `ValidationError` before adding them to its internal storage.
- It doesn't store the given container but rather copies its content to its own internal storage; previously the container itself was added to the `ValidationError` instance and used as internal storage.

This means that if you access the `ValidationError` internal storages, such as `error_list`; `error_dict`; or the return value of `update_error_dict()` you may find instances of `ValidationError` where you would have previously found strings.

Also if you directly assigned the return value of `update_error_dict()` to `Form._errors` you may inadvertently add *list* instances where `ErrorList` instances are expected. This is a problem because unlike a simple *list*, an `ErrorList` knows how to handle instances of `ValidationError`.

Most use-cases that warranted using these private APIs are now covered by the newly introduced *Form.add_error()* method:

```
# Old pattern:
try:
    # ...
except ValidationError as e:
    self._errors = e.update_error_dict(self._errors)

# New pattern:
try:
    # ...
except ValidationError as e:
    self.add_error(None, e)
```

If you need both Django <= 1.6 and 1.7 compatibility you can't use *Form.add_error()* since it wasn't available before Django 1.7, but you can use the following workaround to convert any `list` into `ErrorList`:

```
try:
    # ...
except ValidationError as e:
    self._errors = e.update_error_dict(self._errors)

# Additional code to ensure ``ErrorDict`` is exclusively
# composed of ``ErrorList`` instances.
for field, error_list in self._errors.items():
    if not isinstance(error_list, self.error_class):
        self._errors[field] = self.error_class(error_list)
```

Behavior of `LocMemCache` regarding pickle errors An inconsistency existed in previous versions of Django regarding how pickle errors are handled by different cache backends. `django.core.cache.backends.locmem.LocMemCache` used to fail silently when such an error occurs, which is inconsistent with other backends and leads to cache-specific errors. This has been fixed in Django 1.7, see #21200 for more details.

Cache keys are now generated from the request's absolute URL Previous versions of Django generated cache keys using a request's path and query string but not the scheme or host. If a Django application was serving multiple subdomains or domains, cache keys could collide. In Django 1.7, cache keys vary by the absolute URL of the request including scheme, host, path, and query string. For example, the URL portion of a cache key is now generated from `https://www.example.com/path/to/?key=val` rather than `/path/to/?key=val`. The cache keys generated by Django 1.7 will be different from the keys generated by older versions of Django. After upgrading to Django 1.7, the first request to any previously cached URL will be a cache miss.

Passing `None` to `Manager.db_manager()` In previous versions of Django, it was possible to use `db_manager(using=None)` on a model manager instance to obtain a manager instance using default routing behavior, overriding any manually specified database routing. In Django 1.7, a value of `None` passed to db_manager will produce a router that *retains* any manually assigned database routing – the manager will *not* be reset. This was necessary to resolve an inconsistency in the way routing information cascaded over joins. See #13724 for more details.

pytz may be required If your project handles datetimes before 1970 or after 2037 and Django raises a `ValueError` when encountering them, you will have to install pytz. You may be affected by this problem if you use Django's time zone-related date formats or *`django.contrib.syndication`*.

`remove()` and `clear()` methods of related managers The `remove()` and `clear()` methods of the related managers created by `ForeignKey`, `GenericForeignKey`, and `ManyToManyField` suffered from a number of issues. Some operations ran multiple data modifying queries without wrapping them in a transaction, and some operations didn't respect default filtering when it was present (i.e. when the default manager on the related model implemented a custom `get_queryset()`).

Fixing the issues introduced some backward incompatible changes:

- The default implementation of `remove()` for `ForeignKey` related managers changed from a series of `Model.save()` calls to a single `QuerySet.update()` call. The change means that `pre_save` and `post_save` signals aren't sent anymore. You can use the `bulk=False` keyword argument to revert to the previous behavior.
- The `remove()` and `clear()` methods for `GenericForeignKey` related managers now perform bulk delete. The `Model.delete()` method isn't called on each instance anymore. You can use the `bulk=False` keyword argument to revert to the previous behavior.

- The `remove()` and `clear()` methods for `ManyToManyField` related managers perform nested queries when filtering is involved, which may or may not be an issue depending on your database and your data itself. See *this note* for more details.

Admin login redirection strategy Historically, the Django admin site passed the request from an unauthorized or unauthenticated user directly to the login view, without HTTP redirection. In Django 1.7, this behavior changed to conform to a more traditional workflow where any unauthorized request to an admin page will be redirected (by HTTP status code 302) to the login page, with the `next` parameter set to the referring path. The user will be redirected there after a successful login.

Note also that the admin login form has been updated to not contain the `this_is_the_login_form` field (now unused) and the `ValidationError` code has been set to the more regular `invalid_login` key.

`select_for_update()` requires a transaction Historically, queries that use *`select_for_update()`* could be executed in autocommit mode, outside of a transaction. Before Django 1.6, Django's automatic transactions mode allowed this to be used to lock records until the next write operation. Django 1.6 introduced database-level autocommit; since then, execution in such a context voids the effect of `select_for_update()`. It is, therefore, assumed now to be an error and raises an exception.

This change was made because such errors can be caused by including an app which expects global transactions (e.g. *`ATOMIC_REQUESTS`* set to `True`), or Django's old autocommit behavior, in a project which runs without them; and further, such errors may manifest as data-corruption bugs. It was also made in Django 1.6.3.

This change may cause test failures if you use `select_for_update()` in a test class which is a subclass of *`TransactionTestCase`* rather than *`TestCase`*.

Contrib middleware removed from default `MIDDLEWARE_CLASSES` The *app-loading refactor* deprecated using models from apps which are not part of the *`INSTALLED_APPS`* setting. This exposed an incompatibility between the default *`INSTALLED_APPS`* and *`MIDDLEWARE_CLASSES`* in the global defaults (`django.conf.global_settings`). To bring these settings in sync and prevent deprecation warnings when doing things like testing reusable apps with minimal settings, *`SessionMiddleware`*, *`AuthenticationMiddleware`*, and *`MessageMiddleware`* were removed from the defaults. These classes will still be included in the default settings generated by *`startproject`*. Most projects will not be affected by this change but if you were not previously declaring the *`MIDDLEWARE_CLASSES`* in your project settings and relying on the global default you should ensure that the new defaults are in line with your project's needs. You should also check for any code that accesses `django.conf.global_settings.MIDDLEWARE_CLASSES` directly.

Miscellaneous

- The *`django.core.files.uploadhandler.FileUploadHandler.new_file()`* method is now passed an additional `content_type_extra` parameter. If you have a custom *`FileUploadHandler`* that implements `new_file()`, be sure it accepts this new parameter.
- *`ModelFormSet`*s no longer delete instances when `save(commit=False)` is called. See *`can_delete`* for instructions on how to manually delete objects from deleted forms.
- Loading empty fixtures emits a `RuntimeWarning` rather than raising *`CommandError`*.
- *`django.contrib.staticfiles.views.serve()`* will now raise an *`Http404`* exception instead of *`ImproperlyConfigured`* when *`DEBUG`* is `False`. This change removes the need to conditionally add the view to your root URLconf, which in turn makes it safe to reverse by name. It also removes the ability for visitors to generate spurious HTTP 500 errors by requesting static files that don't exist or haven't been collected yet.

- The *django.db.models.Model.__eq__()* method is now defined in a way where instances of a proxy model and its base model are considered equal when primary keys match. Previously only instances of exact same class were considered equal on primary key match.
- The *django.db.models.Model.__eq__()* method has changed such that two `Model` instances without primary key values won't be considered equal (unless they are the same instance).
- The *django.db.models.Model.__hash__()* method will now raise `TypeError` when called on an instance without a primary key value. This is done to avoid mutable `__hash__` values in containers.
- *AutoField* columns in SQLite databases will now be created using the `AUTOINCREMENT` option, which guarantees monotonic increments. This will cause primary key numbering behavior to change on SQLite, becoming consistent with most other SQL databases. This will only apply to newly created tables. If you have a database created with an older version of Django, you will need to migrate it to take advantage of this feature. For example, you could do the following:
  1. Use *dumpdata* to save your data.
  2. Rename the existing database file (keep it as a backup).
  3. Run *migrate* to create the updated schema.
  4. Use *loaddata* to import the fixtures you exported in (1).
- `django.contrib.auth.models.AbstractUser` no longer defines a *get_absolute_url()* method. The old definition returned `"/users/%s/" % urlquote(self.username)` which was arbitrary since applications may or may not define such a url in `urlpatterns`. Define a `get_absolute_url()` method on your own custom user object or use *ABSOLUTE_URL_OVERRIDES* if you want a URL for your user.
- The static asset-serving functionality of the *django.test.LiveServerTestCase* class has been simplified: Now it's only able to serve content already present in *STATIC_ROOT* when tests are run. The ability to transparently serve all the static assets (similarly to what one gets with *DEBUG = True* at development-time) has been moved to a new class that lives in the `staticfiles` application (the one actually in charge of such feature): *django.contrib.staticfiles.testing.StaticLiveServerTestCase*. In other words, `LiveServerTestCase` itself is less powerful but at the same time has less magic.

  Rationale behind this is removal of dependency of non-contrib code on contrib applications.
- The old cache URI syntax (e.g. `"locmem://"`) is no longer supported. It still worked, even though it was not documented or officially supported. If you're still using it, please update to the current *CACHES* syntax.
- The default ordering of `Form` fields in case of inheritance has changed to follow normal Python MRO. Fields are now discovered by iterating through the MRO in reverse with the topmost class coming last. This only affects you if you relied on the default field ordering while having fields defined on both the current class *and* on a parent `Form`.
- The `required` argument of *SelectDateWidget* has been removed. This widget now respects the form field's `is_required` attribute like other widgets.
- `Widget.is_hidden` is now a read-only property, getting its value by introspecting the presence of `input_type == 'hidden'`.
- *select_related()* now chains in the same way as other similar calls like `prefetch_related`. That is, `select_related('foo', 'bar')` is equivalent to `select_related('foo').select_related('bar')`. Previously the latter would have been equivalent to `select_related('bar')`.
- GeoDjango dropped support for GEOS < 3.1.
- The `init_connection_state` method of database backends now executes in autocommit mode (unless you set *AUTOCOMMIT* to `False`). If you maintain a custom database backend, you should check that method.

- The `django.db.backends.BaseDatabaseFeatures.allows_primary_key_0` attribute has been renamed to `allows_auto_pk_0` to better describe it. It's `True` for all database backends included with Django except MySQL which does allow primary keys with value 0. It only forbids *autoincrement* primary keys with value 0.
- Shadowing model fields defined in a parent model has been forbidden as this creates ambiguity in the expected model behavior. In addition, clashing fields in the model inheritance hierarchy result in a system check error. For example, if you use multi-inheritance, you need to define custom primary key fields on parent models, otherwise the default `id` fields will clash. See *Multiple inheritance* for details.
- `django.utils.translation.parse_accept_lang_header()` now returns lowercase locales, instead of the case as it was provided. As locales should be treated case-insensitive this allows us to speed up locale detection.
- `django.utils.translation.get_language_from_path()` and `django.utils.translation.trans_real.get_supported_language_variant()` now no longer have a `supported` argument.
- The `shortcut` view in `django.contrib.contenttypes.views` now supports protocol-relative URLs (e.g. `//example.com`).
- *`GenericRelation`* now supports an optional `related_query_name` argument. Setting `related_query_name` adds a relation from the related object back to the content type for filtering, ordering and other query operations.
- When running tests on PostgreSQL, the *`USER`* will need read access to the built-in `postgres` database. This is in lieu of the previous behavior of connecting to the actual non-test database.
- As part of the System check framework, *fields, models, and model managers* all implement a `check()` method that is registered with the check framework. If you have an existing method called `check()` on one of these objects, you will need to rename it.
- As noted above in the "Cache" section of "Minor Features", defining the *`TIMEOUT`* argument of the *`CACHES`* setting as `None` will set the cache keys as "non-expiring". Previously, with the memcache backend, a *`TIMEOUT`* of `0` would set non-expiring keys, but this was inconsistent with the set-and-expire (i.e. no caching) behavior of `set("key", "value", timeout=0)`. If you want non-expiring keys, please update your settings to use `None` instead of `0` as the latter now designates set-and-expire in the settings as well.
- The `sql*` management commands now respect the `allow_migrate()` method of *`DATABASE_ROUTERS`*. If you have models synced to non-default databases, use the `--database` flag to get SQL for those models (previously they would always be included in the output).
- Decoding the query string from URLs now falls back to the ISO-8859-1 encoding when the input is not valid UTF-8.
- With the addition of the `django.contrib.auth.middleware.SessionAuthenticationMiddleware` to the default project template (pre-1.7.2 only), a database must be created before accessing a page using *`runserver`*.
- The addition of the `schemes` argument to `URLValidator` will appear as a backwards-incompatible change if you were previously using a custom regular expression to validate schemes. Any scheme not listed in `schemes` will fail validation, even if the regular expression matches the given URL.

Features deprecated in 1.7

`django.core.cache.get_cache` `django.core.cache.get_cache` has been supplanted by *`django.core.cache.caches`*.

django.utils.dictconfig/django.utils.importlib `django.utils.dictconfig` and `django.utils.importlib` were copies of respectively `logging.config` and `importlib` provided for Python versions prior to 2.7. They have been deprecated.

django.utils.module_loading.import_by_path The current `django.utils.module_loading.import_by_p` function catches `AttributeError`, `ImportError`, and `ValueError` exceptions, and re-raises *`ImproperlyConfigured`*. Such exception masking makes it needlessly hard to diagnose circular import problems, because it makes it look like the problem comes from inside Django. It has been deprecated in favor of *`import_string()`*.

django.utils.tzinfo `django.utils.tzinfo` provided two `tzinfo` subclasses, `LocalTimezone` and `FixedOffset`. They've been deprecated in favor of more correct alternatives provided by *`django.utils.timezone`*, *`django.utils.timezone.get_default_timezone()`* and *`django.utils.timezone.get_fixed_timezone()`*.

django.utils.unittest `django.utils.unittest` provided uniform access to the `unittest2` library on all Python versions. Since `unittest2` became the standard library's `unittest` module in Python 2.7, and Django 1.7 drops support for older Python versions, this module isn't useful anymore. It has been deprecated. Use `unittest` instead.

django.utils.datastructures.SortedDict As `OrderedDict` was added to the standard library in Python 2.7, `SortedDict` is no longer needed and has been deprecated.

The two additional, deprecated methods provided by `SortedDict` (`insert()` and `value_for_index()`) have been removed. If you relied on these methods to alter structures like form fields, you should now treat these `OrderedDicts` as immutable objects and override them to change their content.

For example, you might want to override `MyFormClass.base_fields` (although this attribute isn't considered a public API) to change the ordering of fields for all `MyFormClass` instances; or similarly, you could override `self.fields` from inside `MyFormClass.__init__()`, to change the fields for a particular form instance. For example (from Django itself):

```
PasswordChangeForm.base_fields = OrderedDict(
    (k, PasswordChangeForm.base_fields[k])
    for k in ['old_password', 'new_password1', 'new_password2']
)
```

Custom SQL location for models package Previously, if models were organized in a package (`myapp/models/`) rather than simply `myapp/models.py`, Django would look for initial SQL data in `myapp/models/sql/`. This bug has been fixed so that Django will search `myapp/sql/` as documented. After this issue was fixed, migrations were added which deprecates initial SQL data. Thus, while this change still exists, the deprecation is irrelevant as the entire feature will be removed in Django 1.9.

Reorganization of django.contrib.sites `django.contrib.sites` provides reduced functionality when it isn't in *`INSTALLED_APPS`*. The app-loading refactor adds some constraints in that situation. As a consequence, two objects were moved, and the old locations are deprecated:

- *`RequestSite`* now lives in `django.contrib.sites.requests`.
- *`get_current_site()`* now lives in `django.contrib.sites.shortcuts`.

`declared_fieldsets` attribute on `ModelAdmin` `ModelAdmin.declared_fieldsets` has been deprecated. Despite being a private API, it will go through a regular deprecation path. This attribute was mostly used by methods that bypassed `ModelAdmin.get_fieldsets()` but this was considered a bug and has been addressed.

Reorganization of `django.contrib.contenttypes` Since `django.contrib.contenttypes.generic` defined both admin and model related objects, an import of this module could trigger unexpected side effects. As a consequence, its contents were split into *contenttypes* submodules and the `django.contrib.contenttypes.generic` module is deprecated:

- *GenericForeignKey* and *GenericRelation* now live in *fields*.
- *BaseGenericInlineFormSet* and *generic_inlineformset_factory()* now live in *forms*.
- *GenericInlineModelAdmin*, *GenericStackedInline* and *GenericTabularInline* now live in *admin*.

`syncdb` The `syncdb` command has been deprecated in favor of the new *migrate* command. `migrate` takes the same arguments as `syncdb` used to plus a few more, so it's safe to just change the name you're calling and nothing else.

`util` modules renamed to `utils` The following instances of `util.py` in the Django codebase have been renamed to `utils.py` in an effort to unify all util and utils references:

- `django.contrib.admin.util`
- `django.contrib.gis.db.backends.util`
- `django.db.backends.util`
- `django.forms.util`

`get_formsets` method on `ModelAdmin` `ModelAdmin.get_formsets` has been deprecated in favor of the new *get_formsets_with_inlines()*, in order to better handle the case of selectively showing inlines on a `ModelAdmin`.

`IPAddressField` The `django.db.models.IPAddressField` and `django.forms.IPAddressField` fields have been deprecated in favor of *django.db.models.GenericIPAddressField* and *django.forms.GenericIPAddressField*.

`BaseMemcachedCache._get_memcache_timeout` method The `BaseMemcachedCache._get_memcache_timeout` method has been renamed to `get_backend_timeout()`. Despite being a private API, it will go through the normal deprecation.

Natural key serialization options The `--natural` and `-n` options for *dumpdata* have been deprecated. Use *dumpdata --natural-foreign* instead.

Similarly, the `use_natural_keys` argument for `serializers.serialize()` has been deprecated. Use `use_natural_foreign_keys` instead.

Merging of `POST` and `GET` arguments into `WSGIRequest.REQUEST` It was already strongly suggested that you use `GET` and `POST` instead of `REQUEST`, because the former are more explicit. The property `REQUEST` is deprecated and will be removed in Django 1.9.

`django.utils.datastructures.MergeDict` class `MergeDict` exists primarily to support merging `POST` and `GET` arguments into a `REQUEST` property on `WSGIRequest`. To merge dictionaries, use `dict.update()` instead. The class `MergeDict` is deprecated and will be removed in Django 1.9.

Language codes `zh-cn`, `zh-tw` and `fy-nl` The currently used language codes for Simplified Chinese `zh-cn`, Traditional Chinese `zh-tw` and (Western) Frysian `fy-nl` are deprecated and should be replaced by the language codes `zh-hans`, `zh-hant` and `fy` respectively. If you use these language codes, you should rename the locale directories and update your settings to reflect these changes. The deprecated language codes will be removed in Django 1.9.

`django.utils.functional.memoize` function The function `memoize` is deprecated and should be replaced by the `functools.lru_cache` decorator (available from Python 3.2 onwards).

Django ships a backport of this decorator for older Python versions and it's available at `django.utils.lru_cache.lru_cache`. The deprecated function will be removed in Django 1.9.

Geo Sitemaps Google has retired support for the Geo Sitemaps format. Hence Django support for Geo Sitemaps is deprecated and will be removed in Django 1.8.

Passing callable arguments to queryset methods Callable arguments for querysets were an undocumented feature that was unreliable. It's been deprecated and will be removed in Django 1.9.

Callable arguments were evaluated when a queryset was constructed rather than when it was evaluated, thus this feature didn't offer any benefit compared to evaluating arguments before passing them to queryset and created confusion that the arguments may have been evaluated at query time.

`ADMIN_FOR` setting The `ADMIN_FOR` feature, part of the admindocs, has been removed. You can remove the setting from your configuration at your convenience.

`SplitDateTimeWidget` with `DateTimeField` `SplitDateTimeWidget` support in *`DateTimeField`* is deprecated, use `SplitDateTimeWidget` with *`SplitDateTimeField`* instead.

`validate` The `validate` management command is deprecated in favor of the *`check`* command.

`django.core.management.BaseCommand` `requires_model_validation` is deprecated in favor of a new `requires_system_checks` flag. If the latter flag is missing, then the value of the former flag is used. Defining both `requires_system_checks` and `requires_model_validation` results in an error.

The `check()` method has replaced the old `validate()` method.

`ModelAdmin` validators The `ModelAdmin.validator_class` and `default_validator_class` attributes are deprecated in favor of the new `checks_class` attribute.

The `ModelAdmin.validate()` method is deprecated in favor of `ModelAdmin.check()`.

The `django.contrib.admin.validation` module is deprecated.

`django.db.backends.DatabaseValidation.validate_field` This method is deprecated in favor of a new `check_field` method. The functionality required by `check_field()` is the same as that provided by `validate_field()`, but the output format is different. Third-party database backends needing this functionality should provide an implementation of `check_field()`.

Loading `ssi` and `url` template tags from `future` library Django 1.3 introduced `{% load ssi from future %}` and `{% load url from future %}` syntax for forward compatibility of the `ssi` and *`url`* template tags. This syntax is now deprecated and will be removed in Django 1.9. You can simply remove the `{% load ... from future %}` tags.

`django.utils.text.javascript_quote` `javascript_quote()` was an undocumented function present in `django.utils.text`. It was used internally in the *javascript_catalog view* whose implementation was changed to make use of `json.dumps()` instead. If you were relying on this function to provide safe output from untrusted strings, you should use `django.utils.html.escapejs` or the *`escapejs`* template filter. If all you need is to generate valid JavaScript strings, you can simply use `json.dumps()`.

`fix_ampersands` utils method and template filter The `django.utils.html.fix_ampersands` method and the `fix_ampersands` template filter are deprecated, as the escaping of ampersands is already taken care of by Django's standard HTML escaping features. Combining this with `fix_ampersands` would either result in double escaping, or, if the output is assumed to be safe, a risk of introducing XSS vulnerabilities. Along with `fix_ampersands`, `django.utils.html.clean_html` is deprecated, an undocumented function that calls `fix_ampersands`. As this is an accelerated deprecation, `fix_ampersands` and `clean_html` will be removed in Django 1.8.

Reorganization of database test settings All database settings with a `TEST_` prefix have been deprecated in favor of entries in a *`TEST`* dictionary in the database settings. The old settings will be supported until Django 1.9. For backwards compatibility with older versions of Django, you can define both versions of the settings as long as they match.

FastCGI support FastCGI support via the `runfcgi` management command will be removed in Django 1.9. Please deploy your project using WSGI.

Moved objects in `contrib.sites` Following the app-loading refactor, two objects in `django.contrib.sites.models` needed to be moved because they must be available without importing `django.contrib.sites.models` when `django.contrib.sites` isn't installed. Import `RequestSite` from `django.contrib.sites.requests` and `get_current_site()` from `django.contrib.sites.shortcuts`. The old import locations will work until Django 1.9.

`django.forms.forms.get_declared_fields()` Django no longer uses this functional internally. Even though it's a private API, it'll go through the normal deprecation cycle.

Private Query Lookup APIs Private APIs `django.db.models.sql.where.WhereNode.make_atom()` and `django.db.models.sql.where.Constraint` are deprecated in favor of the new custom lookups API.

Features removed in 1.7

These features have reached the end of their deprecation cycle and are removed in Django 1.7. See *Features deprecated in 1.5* for details, including how to remove usage of these features.

- `django.utils.simplejson` is removed.
- `django.utils.itercompat.product` is removed.
- INSTALLED_APPS and TEMPLATE_DIRS are no longer corrected from a plain string into a tuple.
- *`HttpResponse`*, *`SimpleTemplateResponse`*, *`TemplateResponse`*, `render_to_response()`, *`index()`*, and *`sitemap()`* no longer take a `mimetype` argument
- *`HttpResponse`* immediately consumes its content if it's an iterator.
- The `AUTH_PROFILE_MODULE` setting, and the `get_profile()` method on the User model are removed.
- The `cleanup` management command is removed.
- The `daily_cleanup.py` script is removed.
- *`select_related()`* no longer has a `depth` keyword argument.
- The `get_warnings_state()`/`restore_warnings_state()` functions from *`django.test.utils`* and the `save_warnings_state()`/ `restore_warnings_state()` *django.test.*TestCase* are removed.
- The `check_for_test_cookie` method in *`AuthenticationForm`* is removed.
- The version of *`django.contrib.auth.views.password_reset_confirm()`* that supports base36 encoded user IDs (`django.contrib.auth.views.password_reset_confirm_uidb36`) is removed.
- The `django.utils.encoding.StrAndUnicode` mix-in is removed.

9.1.5 1.6 release

Django 1.6.11 release notes

March 18, 2015

Django 1.6.11 fixes two security issues in 1.6.10.

Denial-of-service possibility with `strip_tags()`

Last year *`strip_tags()`* was changed to work iteratively. The problem is that the size of the input it's processing can increase on each iteration which results in an infinite loop in `strip_tags()`. This issue only affects versions of Python that haven't received a bugfix in HTMLParser; namely Python < 2.7.7 and 3.3.5. Some operating system vendors have also backported the fix for the Python bug into their packages of earlier versions.

To remedy this issue, `strip_tags()` will now return the original input if it detects the length of the string it's processing increases. Remember that absolutely NO guarantee is provided about the results of `strip_tags()` being HTML safe. So NEVER mark safe the result of a `strip_tags()` call without escaping it first, for example with *`escape()`*.

Mitigated possible XSS attack via user-supplied redirect URLs

Django relies on user input in some cases (e.g. *`django.contrib.auth.views.login()`* and i18n) to redirect the user to an "on success" URL. The security checks for these redirects (namely `django.utils.http.is_safe_url()`) accepted URLs with leading control characters and so considered URLs like `\x08javascript:...` safe. This issue doesn't affect Django currently, since we only put this URL into the `Location` response header and browsers seem to ignore JavaScript there. Browsers we tested also treat

URLs prefixed with control characters such as `%08//example.com` as relative paths so redirection to an unsafe target isn't a problem either.

However, if a developer relies on `is_safe_url()` to provide safe redirect targets and puts such a URL into a link, they could suffer from an XSS attack as some browsers such as Google Chrome ignore control characters at the start of a URL in an anchor `href`.

Django 1.6.10 release notes

January 13, 2015

Django 1.6.10 fixes several security issues in 1.6.9.

WSGI header spoofing via underscore/dash conflation

When HTTP headers are placed into the WSGI environ, they are normalized by converting to uppercase, converting all dashes to underscores, and prepending *HTTP_*. For instance, a header `X-Auth-User` would become `HTTP_X_AUTH_USER` in the WSGI environ (and thus also in Django's `request.META` dictionary).

Unfortunately, this means that the WSGI environ cannot distinguish between headers containing dashes and headers containing underscores: `X-Auth-User` and `X-Auth_User` both become `HTTP_X_AUTH_USER`. This means that if a header is used in a security-sensitive way (for instance, passing authentication information along from a front-end proxy), even if the proxy carefully strips any incoming value for `X-Auth-User`, an attacker may be able to provide an `X-Auth_User` header (with underscore) and bypass this protection.

In order to prevent such attacks, both Nginx and Apache 2.4+ strip all headers containing underscores from incoming requests by default. Django's built-in development server now does the same. Django's development server is not recommended for production use, but matching the behavior of common production servers reduces the surface area for behavior changes during deployment.

Mitigated possible XSS attack via user-supplied redirect URLs

Django relies on user input in some cases (e.g. *`django.contrib.auth.views.login()`* and i18n) to redirect the user to an "on success" URL. The security checks for these redirects (namely `django.utils.http.is_safe_url()`) didn't strip leading whitespace on the tested URL and as such considered URLs like `\njavascript:...` safe. If a developer relied on `is_safe_url()` to provide safe redirect targets and put such a URL into a link, they could suffer from a XSS attack. This bug doesn't affect Django currently, since we only put this URL into the `Location` response header and browsers seem to ignore JavaScript there.

Denial-of-service attack against `django.views.static.serve`

In older versions of Django, the *`django.views.static.serve()`* view read the files it served one line at a time. Therefore, a big file with no newlines would result in memory usage equal to the size of that file. An attacker could exploit this and launch a denial-of-service attack by simultaneously requesting many large files. This view now reads the file in chunks to prevent large memory usage.

Note, however, that this view has always carried a warning that it is not hardened for production use and should be used only as a development aid. Now may be a good time to audit your project and serve your files in production using a real front-end web server if you are not doing so.

Database denial-of-service with `ModelMultipleChoiceField`

Given a form that uses `ModelMultipleChoiceField` and `show_hidden_initial=True` (not a documented API), it was possible for a user to cause an unreasonable number of SQL queries by submitting duplicate values for the field's data. The validation logic in `ModelMultipleChoiceField` now deduplicates submitted values to address this issue.

Django 1.6.9 release notes

January 2, 2015

Django 1.6.9 fixes a regression in the 1.6.6 security release.

Additionally, Django's vendored version of six, *`django.utils.six`*, has been upgraded to the latest release (1.9.0).

Bugfixes

- Fixed a regression with dynamically generated inlines and allowed field references in the admin (#23754).

Django 1.6.8 release notes

October 22, 2014

Django 1.6.8 fixes a couple regressions in the 1.6.6 security release.

Bugfixes

- Allowed related many-to-many fields to be referenced in the admin (#23604).
- Allowed inline and hidden references to admin fields (#23431).

Django 1.6.7 release notes

September 2, 2014

Django 1.6.7 fixes several bugs in 1.6.6, including a regression related to a security fix in that release.

Bugfixes

- Allowed inherited and m2m fields to be referenced in the admin (#23329).
- Fixed a crash when using `QuerySet.defer()` with `select_related()` (#23370).

Django 1.6.6 release notes

August 20, 2014

Django 1.6.6 fixes several security issues and bugs in 1.6.5.

`reverse()` could generate URLs pointing to other hosts

In certain situations, URL reversing could generate scheme-relative URLs (URLs starting with two slashes), which could unexpectedly redirect a user to a different host. An attacker could exploit this, for example, by redirecting users to a phishing site designed to ask for user's passwords.

To remedy this, URL reversing now ensures that no URL starts with two slashes (//), replacing the second slash with its URL encoded counterpart (%2F). This approach ensures that semantics stay the same, while making the URL relative to the domain and not to the scheme.

File upload denial-of-service

Before this release, Django's file upload handing in its default configuration may degrade to producing a huge number of `os.stat()` system calls when a duplicate filename is uploaded. Since `stat()` may invoke IO, this may produce a huge data-dependent slowdown that slowly worsens over time. The net result is that given enough time, a user with the ability to upload files can cause poor performance in the upload handler, eventually causing it to become very slow simply by uploading 0-byte files. At this point, even a slow network connection and few HTTP requests would be all that is necessary to make a site unavailable.

We've remedied the issue by changing the algorithm for generating file names if a file with the uploaded name already exists. *`Storage.get_available_name()`* now appends an underscore plus a random 7 character alphanumeric string (e.g. `"_x3a1gho"`), rather than iterating through an underscore followed by a number (e.g. `"_1"`, `"_2"`, etc.).

`RemoteUserMiddleware` session hijacking

When using the *`RemoteUserMiddleware`* and the `RemoteUserBackend`, a change to the `REMOTE_USER` header between requests without an intervening logout could result in the prior user's session being co-opted by the subsequent user. The middleware now logs the user out on a failed login attempt.

Data leakage via query string manipulation in `contrib.admin`

In older versions of Django it was possible to reveal any field's data by modifying the "popup" and "to_field" parameters of the query string on an admin change form page. For example, requesting a URL like `/admin/auth/user/?_popup=1&t=password` and viewing the page's HTML allowed viewing the password hash of each user. While the admin requires users to have permissions to view the change form pages in the first place, this could leak data if you rely on users having access to view only certain fields on a model.

To address the issue, an exception will now be raised if a `to_field` value that isn't a related field to a model that has been registered with the admin is specified.

Bugfixes

- Corrected email and URL validation to reject a trailing dash (#22579).
- Prevented indexes on PostgreSQL virtual fields (#22514).
- Prevented edge case where values of FK fields could be initialized with a wrong value when an inline model formset is created for a relationship defined to point to a field other than the PK (#13794).
- Restored `pre_delete` signals for `GenericRelation` cascade deletion (#22998).
- Fixed transaction handling when specifying non-default database in `createcachetable` and `flush` (#23089).

- Fixed the "ORA-01843: not a valid month" errors when using Unicode with older versions of Oracle server (#20292).
- Restored bug fix for sending unicode email with Python 2.6.5 and below (#19107).
- Prevented `UnicodeDecodeError` in `runserver` with non-UTF-8 and non-English locale (#23265).
- Fixed JavaScript errors while editing multi-geometry objects in the OpenLayers widget (#23137, #23293).
- Prevented a crash on Python 3 with query strings containing unencoded non-ASCII characters (#22996).

Django 1.6.5 release notes

May 14, 2014

Django 1.6.5 fixes two security issues and several bugs in 1.6.4.

Issue: Caches may incorrectly be allowed to store and serve private data

In certain situations, Django may allow caches to store private data related to a particular session and then serve that data to requests with a different session, or no session at all. This can lead to information disclosure and can be a vector for cache poisoning.

When using Django sessions, Django will set a `Vary: Cookie` header to ensure caches do not serve cached data to requests from other sessions. However, older versions of Internet Explorer (most likely only Internet Explorer 6, and Internet Explorer 7 if run on Windows XP or Windows Server 2003) are unable to handle the `Vary` header in combination with many content types. Therefore, Django would remove the header if the request was made by Internet Explorer.

To remedy this, the special behavior for these older Internet Explorer versions has been removed, and the `Vary` header is no longer stripped from the response. In addition, modifications to the `Cache-Control` header for all Internet Explorer requests with a `Content-Disposition` header have also been removed as they were found to have similar issues.

Issue: Malformed redirect URLs from user input not correctly validated

The validation for redirects did not correctly validate some malformed URLs, which are accepted by some browsers. This allows a user to be redirected to an unsafe URL unexpectedly.

Django relies on user input in some cases (e.g. *`django.contrib.auth.views.login()`*, `django.contrib.comments`, and i18n) to redirect the user to an "on success" URL. The security checks for these redirects (namely `django.utils.http.is_safe_url()`) did not correctly validate some malformed URLs, such as *http:\\\djangoproject.com*, which are accepted by some browsers with more liberal URL parsing.

To remedy this, the validation in `is_safe_url()` has been tightened to be able to handle and correctly validate these malformed URLs.

Bugfixes

- Made the `year_lookup_bounds_for_datetime_field` Oracle backend method Python 3 compatible (#22551).
- Fixed `pgettext_lazy` crash when receiving bytestring content on Python 2 (#22565).
- Fixed the SQL generated when filtering by a negated `Q` object that contains a `F` object. (#22429).

- Avoided overwriting data fetched by `select_related()` in certain cases which could cause minor performance regressions (#22508).

Django 1.6.4 release notes

April 28, 2014

Django 1.6.4 fixes several bugs in 1.6.3.

Bugfixes

- Added backwards compatibility support for the *`django.contrib.messages`* cookie format of Django 1.4 and earlier to facilitate upgrading to 1.6 from 1.4 (#22426).
- Restored the ability to `reverse()` views created using `functools.partial()` (#22486).
- Fixed the `object_id` of the `LogEntry` that's created after a user password change in the admin (#22515).

Django 1.6.3 release notes

April 21, 2014

Django 1.6.3 fixes several bugs in 1.6.2, including three security issues, and makes one backwards-incompatible change:

Unexpected code execution using `reverse()`

Django's URL handling is based on a mapping of regex patterns (representing the URLs) to callable views, and Django's own processing consists of matching a requested URL against those patterns to determine the appropriate view to invoke.

Django also provides a convenience function – `reverse()` – which performs this process in the opposite direction. The `reverse()` function takes information about a view and returns a URL which would invoke that view. Use of `reverse()` is encouraged for application developers, as the output of `reverse()` is always based on the current URL patterns, meaning developers do not need to change other code when making changes to URLs.

One argument signature for `reverse()` is to pass a dotted Python path to the desired view. In this situation, Django will import the module indicated by that dotted path as part of generating the resulting URL. If such a module has import-time side effects, those side effects will occur.

Thus it is possible for an attacker to cause unexpected code execution, given the following conditions:

1. One or more views are present which construct a URL based on user input (commonly, a "next" parameter in a querystring indicating where to redirect upon successful completion of an action).
2. One or more modules are known to an attacker to exist on the server's Python import path, which perform code execution with side effects on importing.

To remedy this, `reverse()` will now only accept and import dotted paths based on the view-containing modules listed in the project's URL pattern configuration, so as to ensure that only modules the developer intended to be imported in this fashion can or will be imported.

Caching of anonymous pages could reveal CSRF token

Django includes both a caching framework and a system for preventing cross-site request forgery (CSRF) attacks. The CSRF-protection system is based on a random nonce sent to the client in a cookie which must be sent by the client on future requests and, in forms, a hidden value which must be submitted back with the form.

The caching framework includes an option to cache responses to anonymous (i.e., unauthenticated) clients.

When the first anonymous request to a given page is by a client which did not have a CSRF cookie, the cache framework will also cache the CSRF cookie and serve the same nonce to other anonymous clients who do not have a CSRF cookie. This can allow an attacker to obtain a valid CSRF cookie value and perform attacks which bypass the check for the cookie.

To remedy this, the caching framework will no longer cache such responses. The heuristic for this will be:

1. If the incoming request did not submit any cookies, and
2. If the response did send one or more cookies, and
3. If the `Vary: Cookie` header is set on the response, then the response will not be cached.

MySQL typecasting

The MySQL database is known to "typecast" on certain queries; for example, when querying a table which contains string values, but using a query which filters based on an integer value, MySQL will first silently coerce the strings to integers and return a result based on that.

If a query is performed without first converting values to the appropriate type, this can produce unexpected results, similar to what would occur if the query itself had been manipulated.

Django's model field classes are aware of their own types and most such classes perform explicit conversion of query arguments to the correct database-level type before querying. However, three model field classes did not correctly convert their arguments:

- *FilePathField*
- *GenericIPAddressField*
- `IPAddressField`

These three fields have been updated to convert their arguments to the correct types before querying.

Additionally, developers of custom model fields are now warned via documentation to ensure their custom field classes will perform appropriate type conversions, and users of the *raw()* and *extra()* query methods – which allow the developer to supply raw SQL or SQL fragments – will be advised to ensure they perform appropriate manual type conversions prior to executing queries.

`select_for_update()` requires a transaction

Historically, queries that use *select_for_update()* could be executed in autocommit mode, outside of a transaction. Before Django 1.6, Django's automatic transactions mode allowed this to be used to lock records until the next write operation. Django 1.6 introduced database-level autocommit; since then, execution in such a context voids the effect of `select_for_update()`. It is, therefore, assumed now to be an error and raises an exception.

This change was made because such errors can be caused by including an app which expects global transactions (e.g. *ATOMIC_REQUESTS* set to `True`), or Django's old autocommit behavior, in a project which runs without them; and further, such errors may manifest as data-corruption bugs.

This change may cause test failures if you use `select_for_update()` in a test class which is a subclass of *`TransactionTestCase`* rather than *`TestCase`*.

Other bugfixes and changes

- Content retrieved from the GeoIP library is now properly decoded from its default `iso-8859-1` encoding (#21996).
- Fixed `AttributeError` when using *`bulk_create()`* with `ForeignObject` (#21566).
- Fixed crash of `QuerySets` that use `F() + timedelta()` when their query was compiled more once (#21643).
- Prevented custom `widget` class attribute of *`IntegerField`* subclasses from being overwritten by the code in their `__init__` method (#22245).
- Improved *`strip_tags()`* accuracy (but it still cannot guarantee an HTML-safe result, as stated in the documentation).
- Fixed a regression in the *`django.contrib.gis`* SQL compiler for non-concrete fields (#22250).
- Fixed *`ModelAdmin.preserve_filters`* when running a site with a URL prefix (#21795).
- Fixed a crash in the `find_command` management utility when the `PATH` environment variable wasn't set (#22256).
- Fixed *`changepassword`* on Windows (#22364).
- Avoided shadowing deadlock exceptions on MySQL (#22291).
- Wrapped database exceptions in `_set_autocommit` (#22321).
- Fixed atomicity when closing a database connection or when the database server disconnects (#21239 and #21202)
- Fixed regression in `prefetch_related` that caused the related objects query to include an unnecessary join (#21760).

Additionally, Django's vendored version of six, *`django.utils.six`* has been upgraded to the latest release (1.6.1).

Django 1.6.2 release notes

February 6, 2014

This is Django 1.6.2, a bugfix release for Django 1.6. Django 1.6.2 fixes several bugs in 1.6.1:

- Prevented the base geometry object of a prepared geometry to be garbage collected, which could lead to crash Django (#21662).
- Fixed a crash when executing the *`changepassword`* command when the user object representation contained non-ASCII characters (#21627).
- The *`collectstatic`* command will raise an error rather than default to using the current working directory if *`STATIC_ROOT`* is not set. Combined with the `--clear` option, the previous behavior could wipe anything below the current working directory (#21581).
- Fixed mail encoding on Python 3.3.3+ (#21093).
- Fixed an issue where when `settings.DATABASES['default']['AUTOCOMMIT'] = False`, the connection wasn't in autocommit mode but Django pretended it was.
- Fixed a regression in multiple-table inheritance `exclude()` queries (#21787).

- Added missing items to `django.utils.timezone.__all__` (#21880).
- Fixed a field misalignment issue with `select_related()` and model inheritance (#21413).
- Fixed join promotion for negated `AND` conditions (#21748).
- Oracle database introspection now works with boolean and float fields (#19884).
- Fixed an issue where lazy objects weren't actually marked as safe when passed through *`mark_safe()`* and could end up being double-escaped (#21882).

Additionally, Django's vendored version of six, *`django.utils.six`* has been upgraded to the latest release (1.5.2).

Django 1.6.1 release notes

December 12, 2013

This is Django 1.6.1, a bugfix release for Django 1.6. In addition to the bug fixes listed below, translations submitted since the 1.6 release are also included.

Bug fixes

- Fixed `BCryptSHA256PasswordHasher` with py-bcrypt and Python 3 (#21398).
- Fixed a regression that prevented a `ForeignKey` with a hidden reverse manager (`related_name` ending with '+') from being used as a lookup for `prefetch_related` (#21410).
- Fixed *`Queryset.datetimes`* raising `AttributeError` in some situations (#21432).
- Fixed *`ModelBackend`* raising `UnboundLocalError` if *`get_user_model()`* raised an error (#21439).
- Fixed a regression that prevented editable `GenericRelation` subclasses from working in `ModelForms` (#21428).
- Added missing `to_python` method for `ModelMultipleChoiceField` which is required in Django 1.6 to properly detect changes from initial values (#21568).
- Fixed `django.contrib.humanize` translations where the unicode sequence for the non-breaking space was returned verbatim (#21415).
- Fixed *`loaddata`* error when fixture file name contained any dots not related to file extensions (#21457) or when fixture path was relative but located in a subdirectory (#21551).
- Fixed display of inline instances in formsets when parent has 0 for primary key (#21472).
- Fixed a regression where custom querysets for foreign keys were overwritten if `ModelAdmin` had ordering set (#21405).
- Removed mention of a feature in the `--locale/-l` option of the `makemessages` and `compilemessages` commands that never worked as promised: Support of multiple locale names separated by commas. It's still possible to specify multiple locales in one run by using the option multiple times (#21488, #17181).
- Fixed a regression that unnecessarily triggered settings configuration when importing `get_wsgi_application` (#21486).
- Fixed test client `logout()` method when using the cookie-based session backend (#21448).
- Fixed a crash when a `GeometryField` uses a non-geometric widget (#21496).
- Fixed password hash upgrade when changing the iteration count (#21535).
- Fixed a bug in the debug view when the URLconf only contains one element (#21530).

- Re-added missing search result count and reset link in changelist admin view (#21510).
- The current language is no longer saved to the session by `LocaleMiddleware` on every response, but rather only after a logout (#21473).
- Fixed a crash when executing `runserver` on non-English systems and when the formatted date in its output contained non-ASCII characters (#21358).
- Fixed a crash in the debug view after an exception occurred on Python $\geq$ 3.3 (#21443).
- Fixed a crash in *`ImageField`* on some platforms (Homebrew and RHEL6 reported) (#21355).
- Fixed a regression when using generic relations in `ModelAdmin.list_filter` (#21431).

Django 1.6 release notes

: Dedicated to Malcolm Tredinnick

On March 17, 2013, the Django project and the free software community lost a very dear friend and developer.

Malcolm was a long-time contributor to Django, a model community member, a brilliant mind, and a friend. His contributions to Django — and to many other open source projects — are nearly impossible to enumerate. Many on the core Django team had their first patches reviewed by him; his mentorship enriched us. His consideration, patience, and dedication will always be an inspiration to us.

This release of Django is for Malcolm.

– The Django Developers

November 6, 2013

Welcome to Django 1.6!

These release notes cover the *new features*, as well as some *backwards incompatible changes* you'll want to be aware of when upgrading from Django 1.5 or older versions. We've also dropped some features, which are detailed in *our deprecation plan*, and we've *begun the deprecation process for some features*.

Python compatibility

Django 1.6, like Django 1.5, requires Python 2.6.5 or above. Python 3 is also officially supported. We **highly recommend** the latest minor release for each supported Python series (2.6.X, 2.7.X, 3.2.X, and 3.3.X).

Django 1.6 will be the final release series to support Python 2.6; beginning with Django 1.7, the minimum supported Python version will be 2.7.

Python 3.4 is not supported, but support will be added in Django 1.7.

What's new in Django 1.6

Simplified default project and app templates The default templates used by *`startproject`* and *`startapp`* have been simplified and modernized. The admin is now enabled by default in new projects; the sites framework no longer is. *clickjacking prevention* is now on and the database defaults to SQLite.

If the default templates don't suit your tastes, you can use *custom project and app templates*.

Improved transaction management Django's transaction management was overhauled. Database-level autocommit is now turned on by default. This makes transaction handling more explicit and should improve performance. The existing APIs were deprecated, and new APIs were introduced, as described in the transaction management docs.

Persistent database connections Django now supports reusing the same database connection for several requests. This avoids the overhead of re-establishing a connection at the beginning of each request. For backwards compatibility, this feature is disabled by default. See *Persistent connections* for details.

Discovery of tests in any test module Django 1.6 ships with a new test runner that allows more flexibility in the location of tests. The previous runner (`django.test.simple.DjangoTestSuiteRunner`) found tests only in the `models.py` and `tests.py` modules of a Python package in *`INSTALLED_APPS`*.

The new runner (`django.test.runner.DiscoverRunner`) uses the test discovery features built into `unittest2` (the version of `unittest` in the Python 2.7+ standard library, and bundled with Django). With test discovery, tests can be located in any module whose name matches the pattern `test*.py`.

In addition, the test labels provided to `./manage.py test` to nominate specific tests to run must now be full Python dotted paths (or directory paths), rather than `applabel.TestCase.test_method_name` pseudo-paths. This allows running tests located anywhere in your codebase, rather than only in *`INSTALLED_APPS`*. For more details, see Testing in Django.

This change is backwards-incompatible; see the *backwards-incompatibility notes*.

Time zone aware aggregation The support for time zones introduced in Django 1.4 didn't work well with *`QuerySet.dates()`*: aggregation was always performed in UTC. This limitation was lifted in Django 1.6. Use *`QuerySet.datetimes()`* to perform time zone aware aggregation on a *`DateTimeField`*.

Support for savepoints in SQLite Django 1.6 adds support for savepoints in SQLite, with some *limitations*.

`BinaryField` model field A new *`django.db.models.BinaryField`* model field allows storage of raw binary data in the database.

GeoDjango form widgets GeoDjango now provides form fields and widgets for its geo-specialized fields. They are OpenLayers-based by default, but they can be customized to use any other JS framework.

`check` management command added for verifying compatibility A *`check`* management command was added, enabling you to verify if your current configuration (currently oriented at settings) is compatible with the current version of Django.

`Model.save()` algorithm changed The *`Model.save()`* method now tries to directly `UPDATE` the database if the instance has a primary key value. Previously `SELECT` was performed to determine if `UPDATE` or `INSERT` were needed. The new algorithm needs only one query for updating an existing row while the old algorithm needed two. See *`Model.save()`* for more details.

In some rare cases the database doesn't report that a matching row was found when doing an `UPDATE`. An example is the PostgreSQL `ON UPDATE` trigger which returns `NULL`. In such cases it is possible to set *`django.db.models.Options.select_on_save`* flag to force saving to use the old algorithm.

Minor features

- Authentication backends can raise `PermissionDenied` to immediately fail the authentication chain.
- The `HttpOnly` flag can be set on the CSRF cookie with *`CSRF_COOKIE_HTTPONLY`*.
- The *`assertQuerysetEqual()`* now checks for undefined order and raises `ValueError` if undefined order is spotted. The order is seen as undefined if the given `QuerySet` isn't ordered and there are more than one ordered values to compare against.
- Added *`earliest()`* for symmetry with *`latest()`*.
- In addition to *`year`*, *`month`* and *`day`*, the ORM now supports *`hour`*, *`minute`* and *`second`* lookups.
- Django now wraps all PEP-249 exceptions.
- The default widgets for *`EmailField`*, *`URLField`*, *`IntegerField`*, *`FloatField`* and *`DecimalField`* use the new type attributes available in HTML5 (`type='email'`, `type='url'`, `type='number'`). Note that due to erratic support of the `number` input type with localized numbers in current browsers, Django only uses it when numeric fields are not localized.
- The `number` argument for *lazy plural translations* can be provided at translation time rather than at definition time.
- For custom management commands: Verification of the presence of valid settings in commands that ask for it by using the *`can_import_settings`* internal option is now performed independently from handling of the locale that should be active during the execution of the command. The latter can now be influenced by the new *`leave_locale_alone`* internal option. See *Management commands and locales* for more details.
- The *`success_url`* of *`DeletionMixin`* is now interpolated with its `object`'s `__dict__`.
- *`HttpResponseRedirect`* and *`HttpResponsePermanentRedirect`* now provide an `url` attribute (equivalent to the URL the response will redirect to).
- The `MemcachedCache` cache backend now uses the latest `pickle` protocol available.
- Added *`SuccessMessageMixin`* which provides a `success_message` attribute for *`FormView`* based classes.
- Added the *`django.db.models.ForeignKey.db_constraint`* and *`django.db.models.ManyToManyField.db_constraint`* options.
- The jQuery library embedded in the admin has been upgraded to version 1.9.1.
- Syndication feeds (*`django.contrib.syndication`*) can now pass extra context through to feed templates using a new *`Feed.get_context_data()`* callback.
- The admin list columns have a `column-<field_name>` class in the HTML so the columns header can be styled with CSS, e.g. to set a column width.
- The *isolation level* can be customized under PostgreSQL.
- The *`blocktrans`* template tag now respects `TEMPLATE_STRING_IF_INVALID` for variables not present in the context, just like other template constructs.
- `SimpleLazyObjects` will now present more helpful representations in shell debugging situations.
- Generic *`GeometryField`* is now editable with the OpenLayers widget in the admin.
- The documentation contains a deployment checklist.
- The *`diffsettings`* command gained a `--all` option.
- `django.forms.fields.Field.__init__` now calls `super()`, allowing field mixins to implement `__init__()` methods that will reliably be called.

- The `validate_max` parameter was added to `BaseFormSet` and *`formset_factory()`*, and `ModelForm` and inline versions of the same. The behavior of validation for formsets with `max_num` was clarified. The previously undocumented behavior that hardened formsets against memory exhaustion attacks was documented, and the undocumented limit of the higher of 1000 or `max_num` forms was changed so it is always 1000 more than `max_num`.
- Added `BCryptSHA256PasswordHasher` to resolve the password truncation issue with bcrypt.
- Pillow is now the preferred image manipulation library to use with Django. PIL is pending deprecation (support to be removed in Django 1.8). To upgrade, you should **first** uninstall PIL, **then** install Pillow.
- *`ModelForm`* accepts several new `Meta` options.
  - Fields included in the `localized_fields` list will be localized (by setting `localize` on the form field).
  - The `labels`, `help_texts` and `error_messages` options may be used to customize the default fields, see *Overriding the default fields* for details.
- The `choices` argument to model fields now accepts an iterable of iterables instead of requiring an iterable of lists or tuples.
- The reason phrase can be customized in HTTP responses using *`reason_phrase`*.
- When giving the URL of the next page for *`logout()`*, *`password_reset()`*, *`password_reset_confirm()`*, and *`password_change()`*, you can now pass URL names and they will be resolved.
- The new *`dumpdata --pks`* option specifies the primary keys of objects to dump. This option can only be used with one model.
- Added `QuerySet` methods *`first()`* and *`last()`* which are convenience methods returning the first or last object matching the filters. Returns `None` if there are no objects matching.
- *`View`* and *`RedirectView`* now support HTTP `PATCH` method.
- `GenericForeignKey` now takes an optional `for_concrete_model` argument, which when set to `False` allows the field to reference proxy models. The default is `True` to retain the old behavior.
- The *`LocaleMiddleware`* now stores the active language in session if it is not present there. This prevents loss of language settings after session flush, e.g. logout.
- *`SuspiciousOperation`* has been differentiated into a number of subclasses, and each will log to a matching named logger under the `django.security` logging hierarchy. Along with this change, a `handler400` mechanism and default view are used whenever a `SuspiciousOperation` reaches the WSGI handler to return an `HttpResponseBadRequest`.
- The *`DoesNotExist`* exception now includes a message indicating the name of the attribute used for the lookup.
- The *`get_or_create()`* method no longer requires at least one keyword argument.
- The *`SimpleTestCase`* class includes a new assertion helper for testing formset errors: *`assertFormsetError()`*.
- The list of related fields added to a *`QuerySet`* by *`select_related()`* can be cleared using `select_related(None)`.
- The *`get_extra()`* and *`get_max_num()`* methods on *`InlineModelAdmin`* may be overridden to customize the extra and maximum number of inline forms.
- Formsets now have a *`total_error_count()`* method.

- *ModelForm* fields can now override error messages defined in model fields by using the *error_messages* argument of a `Field`'s constructor. To take advantage of this new feature with your custom fields, *see the updated recommendation* for raising a `ValidationError`.
- *ModelAdmin* now preserves filters on the list view after creating, editing or deleting an object. It's possible to restore the previous behavior of clearing filters by setting the *preserve_filters* attribute to `False`.
- Added *FormMixin.get_prefix* (which returns *FormMixin.prefix* by default) to allow customizing the *prefix* of the form.
- Raw queries (`Manager.raw()` or `cursor.execute()`) can now use the "pyformat" parameter style, where placeholders in the query are given as `'%(name)s'` and the parameters are passed as a dictionary rather than a list (except on SQLite). This has long been possible (but not officially supported) on MySQL and PostgreSQL, and is now also available on Oracle.
- The default iteration count for the PBKDF2 password hasher has been increased by 20%. This backwards compatible change will not affect existing passwords or users who have subclassed `django.contrib.auth.hashers.PBKDF2PasswordHasher` to change the default value. Passwords *will be upgraded* to use the new iteration count as necessary.

Backwards incompatible changes in 1.6

: In addition to the changes outlined in this section, be sure to review the *deprecation plan* for any features that have been removed. If you haven't updated your code within the deprecation timeline for a given feature, its removal may appear as a backwards incompatible change.

New transaction management model

Behavior changes Database-level autocommit is enabled by default in Django 1.6. While this doesn't change the general spirit of Django's transaction management, there are a few backwards-incompatibilities.

Savepoints and `assertNumQueries` The changes in transaction management may result in additional statements to create, release or rollback savepoints. This is more likely to happen with SQLite, since it didn't support savepoints until this release.

If tests using *assertNumQueries()* fail because of a higher number of queries than expected, check that the extra queries are related to savepoints, and adjust the expected number of queries accordingly.

Autocommit option for PostgreSQL In previous versions, database-level autocommit was only an option for PostgreSQL, and it was disabled by default. This option is now ignored and can be removed.

New test runner In order to maintain greater consistency with Python's unittest module, the new test runner (`django.test.runner.DiscoverRunner`) does not automatically support some types of tests that were supported by the previous runner:

- Tests in `models.py` and `tests/__init__.py` files will no longer be found and run. Move them to a file whose name begins with `test`.
- Doctests will no longer be automatically discovered. To integrate doctests in your test suite, follow the recommendations in the Python documentation.

Django bundles a modified version of the doctest module from the Python standard library (in django.test._doctest) and includes some additional doctest utilities. These utilities are deprecated and will be removed in Django 1.8; doctest suites should be updated to work with the standard library's doctest module (or converted to unittest-compatible tests).

If you wish to delay updates to your test suite, you can set your *TEST_RUNNER* setting to django.test.simple.DjangoTestSuiteRunner to fully restore the old test behavior. DjangoTestSuiteRunner is deprecated but will not be removed from Django until version 1.8.

Removal of django.contrib.gis.tests.GeoDjangoTestSuiteRunner GeoDjango custom test runner This is for developers working on the GeoDjango application itself and related to the item above about changes in the test runners:

The django.contrib.gis.tests.GeoDjangoTestSuiteRunner test runner has been removed and the standalone GeoDjango tests execution setup it implemented isn't supported anymore. To run the GeoDjango tests simply use the new DiscoverRunner and specify the django.contrib.gis app.

Custom User models in tests The introduction of the new test runner has also slightly changed the way that test models are imported. As a result, any test that overrides AUTH_USER_MODEL to test behavior with one of Django's test user models (django.contrib.auth.tests.custom_user.CustomUser and django.contrib.auth.tests.custom_user.ExtensionUser) must now explicitly import the User model in your test module:

```
from django.contrib.auth.tests.custom_user import CustomUser

@override_settings(AUTH_USER_MODEL='auth.CustomUser')
class CustomUserFeatureTests(TestCase):
    def test_something(self):
        # Test code here ...
```

This import forces the custom user model to be registered. Without this import, the test will be unable to swap in the custom user model, and you will get an error reporting:

```
ImproperlyConfigured: AUTH_USER_MODEL refers to model 'auth.CustomUser' that has not been installed
```

Time zone-aware day, month, and week_day lookups Django 1.6 introduces time zone support for *day*, *month*, and *week_day* lookups when *USE_TZ* is True. These lookups were previously performed in UTC regardless of the current time zone.

This requires *time zone definitions in the database*. If you're using SQLite, you must install pytz. If you're using MySQL, you must install pytz and load the time zone tables with mysql_tzinfo_to_sql.

Addition of QuerySet.datetimes() When the time zone support added in Django 1.4 was active, *QuerySet.dates()* lookups returned unexpected results, because the aggregation was performed in UTC. To fix this, Django 1.6 introduces a new API, *QuerySet.datetimes()*. This requires a few changes in your code.

QuerySet.dates() returns date objects *QuerySet.dates()* now returns a list of date. It used to return a list of datetime.

QuerySet.datetimes() returns a list of datetime.

QuerySet.dates() no longer usable on DateTimeField *QuerySet.dates()* raises an error if it's used on *DateTimeField* when time zone support is active. Use *QuerySet.datetimes()* instead.

`date_hierarchy` requires time zone definitions The *`date_hierarchy`* feature of the admin now relies on *`QuerySet.datetimes()`* when it's used on a *`DateTimeField`*.

This requires time zone definitions in the database when *`USE_TZ`* is `True`. *Learn more*.

`date_list` in generic views requires time zone definitions For the same reason, accessing `date_list` in the context of a date-based generic view requires time zone definitions in the database when the view is based on a *`DateTimeField`* and *`USE_TZ`* is `True`. *Learn more*.

New lookups may clash with model fields Django 1.6 introduces `hour`, `minute`, and `second` lookups on *`DateTimeField`*. If you had model fields called `hour`, `minute`, or `second`, the new lookups will clash with you field names. Append an explicit *`exact`* lookup if this is an issue.

`BooleanField` no longer defaults to `False` When a *`BooleanField`* doesn't have an explicit *`default`*, the implicit default value is `None`. In previous version of Django, it was `False`, but that didn't represent accurately the lack of a value.

Code that relies on the default value being `False` may raise an exception when saving new model instances to the database, because `None` isn't an acceptable value for a *`BooleanField`*. You should either specify `default=False` in the field definition, or ensure the field is set to `True` or `False` before saving the object.

Translations and comments in templates

Extraction of translations after comments Extraction of translatable literals from templates with the *`makemessages`* command now correctly detects i18n constructs when they are located after a `{# / #}`-type comment on the same line. E.g.:

```
{# A comment #}{% trans "This literal was incorrectly ignored. Not anymore" %}
```

Location of translator comments *Comments for translators in templates* specified using `{# / #}` need to be at the end of a line. If they are not, the comments are ignored and *`makemessages`* will generate a warning. For example:

```
{# Translators: This is ignored #}{% trans "Translate me" %}
{{ title }}{# Translators: Extracted and associated with 'Welcome' below #}
<h1>{% trans "Welcome" %}</h1>
```

Quoting in `reverse()` When reversing URLs, Django didn't apply *`urlquote()`* to arguments before interpolating them in URL patterns. This bug is fixed in Django 1.6. If you worked around this bug by applying URL quoting before passing arguments to `reverse()`, this may result in double-quoting. If this happens, simply remove the URL quoting from your code. You will also have to replace special characters in URLs used in *`assertRedirects()`* with their encoded versions.

Storage of IP addresses in the comments app The comments app now uses a `GenericIPAddressField` for storing commenters' IP addresses, to support comments submitted from IPv6 addresses. Until now, it stored them in an `IPAddressField`, which is only meant to support IPv4. When saving a comment made from an IPv6 address, the address would be silently truncated on MySQL databases, and raise an exception on Oracle. You will need to change the column type in your database to benefit from this change.

For MySQL, execute this query on your project's database:

```
ALTER TABLE django_comments MODIFY ip_address VARCHAR(39);
```

For Oracle, execute this query:

```
ALTER TABLE DJANGO_COMMENTS MODIFY (ip_address VARCHAR2(39));
```

If you do not apply this change, the behavior is unchanged: on MySQL, IPv6 addresses are silently truncated; on Oracle, an exception is generated. No database change is needed for SQLite or PostgreSQL databases.

Percent literals in `cursor.execute` queries When you are running raw SQL queries through the *cursor.execute* method, the rule about doubling percent literals (`%`) inside the query has been unified. Past behavior depended on the database backend. Now, across all backends, you only need to double literal percent characters if you are also providing replacement parameters. For example:

```
# No parameters, no percent doubling
cursor.execute("SELECT foo FROM bar WHERE baz = '30%'")

# Parameters passed, non-placeholders have to be doubled
cursor.execute("SELECT foo FROM bar WHERE baz = '30%%' and id = %s", [self.id])
```

`SQLite` users need to check and update such queries.

Help text of model form fields for ManyToManyField fields HTML rendering of model form fields corresponding to *`ManyToManyField`* model fields used to get the hard-coded sentence:

> *Hold down "Control", or "Command" on a Mac, to select more than one.*

(or its translation to the active locale) imposed as the help legend shown along them if neither *`model`* nor *`form`* `help_text` attributes were specified by the user (or this string was appended to any `help_text` that was provided).

Since this happened at the model layer, there was no way to prevent the text from appearing in cases where it wasn't applicable such as form fields that implement user interactions that don't involve a keyboard and/or a mouse.

Starting with Django 1.6, as an ad-hoc temporary backward-compatibility provision, the logic to add the "Hold down..." sentence has been moved to the model form field layer and modified to add the text only when the associated widget is *`SelectMultiple`* or selected subclasses.

The change can affect you in a backward incompatible way if you employ custom model form fields and/or widgets for `ManyToManyField` model fields whose UIs do rely on the automatic provision of the mentioned hard-coded sentence. These form field implementations need to adapt to the new scenario by providing their own handling of the `help_text` attribute.

Applications that use Django model form facilities together with Django built-in form fields and widgets aren't affected but need to be aware of what's described in *Munging of help text of model form fields for ManyToManyField fields* below.

QuerySet iteration The `QuerySet` iteration was changed to immediately convert all fetched rows to `Model` objects. In Django 1.5 and earlier the fetched rows were converted to `Model` objects in chunks of 100.

Existing code will work, but the amount of rows converted to objects might change in certain use cases. Such usages include partially looping over a queryset or any usage which ends up doing `__bool__` or `__contains__`.

Notably most database backends did fetch all the rows in one go already in 1.5.

It is still possible to convert the fetched rows to `Model` objects lazily by using the *`iterator()`* method.

`BoundField.label_tag` now includes the form's `label_suffix` This is consistent with how methods like *`Form.as_p`* and *`Form.as_ul`* render labels.

If you manually render `label_tag` in your templates:

```
{{ form.my_field.label_tag }}: {{ form.my_field }}
```

you'll want to remove the colon (or whatever other separator you may be using) to avoid duplicating it when upgrading to Django 1.6. The following template in Django 1.6 will render identically to the above template in Django 1.5, except that the colon will appear inside the `<label>` element.

```
{{ form.my_field.label_tag }} {{ form.my_field }}
```

will render something like:

```
<label for="id_my_field">My Field:</label> <input id="id_my_field" type="text" name="my_field" />
```

If you want to keep the current behavior of rendering `label_tag` without the `label_suffix`, instantiate the form `label_suffix=''`. You can also customize the `label_suffix` on a per-field basis using the new `label_suffix` parameter on *`label_tag()`*.

Admin views `_changelist_filters` GET parameter To achieve preserving and restoring list view filters, admin views now pass around the *_changelist_filters* GET parameter. It's important that you account for that change if you have custom admin templates or if your tests rely on the previous URLs. If you want to revert to the original behavior you can set the *`preserve_filters`* attribute to `False`.

`django.contrib.auth` password reset uses base 64 encoding of `User` PK Past versions of Django used base 36 encoding of the `User` primary key in the password reset views and URLs (*`django.contrib.auth.views.password_reset_confirm()`*). Base 36 encoding is sufficient if the user primary key is an integer, however, with the introduction of custom user models in Django 1.5, that assumption may no longer be true.

`django.contrib.auth.views.password_reset_confirm()` has been modified to take a `uidb64` parameter instead of `uidb36`. If you are reversing this view, for example in a custom `password_reset_email.html` template, be sure to update your code.

A temporary shim for *`django.contrib.auth.views.password_reset_confirm()`* that will allow password reset links generated prior to Django 1.6 to continue to work has been added to provide backwards compatibility; this will be removed in Django 1.7. Thus, as long as your site has been running Django 1.6 for more than *`PASSWORD_RESET_TIMEOUT_DAYS`*, this change will have no effect. If not (for example, if you upgrade directly from Django 1.5 to Django 1.7), then any password reset links generated before you upgrade to Django 1.7 or later won't work after the upgrade.

In addition, if you have any custom password reset URLs, you will need to update them by replacing `uidb36` with `uidb64` and the dash that follows that pattern with a slash. Also add `_\-` to the list of characters that may match the `uidb64` pattern.

For example:

```
url(r'^reset/(?P<uidb36>[0-9A-Za-z]+)-(?P<token>.+)/$',
    'django.contrib.auth.views.password_reset_confirm',
    name='password_reset_confirm'),
```

becomes:

```
url(r'^reset/(?P<uidb64>[0-9A-Za-z_\-]+)/(?P<token>.+)/$',
    'django.contrib.auth.views.password_reset_confirm',
    name='password_reset_confirm'),
```

You may also want to add the shim to support the old style reset links. Using the example above, you would modify the existing url by replacing `django.contrib.auth.views.password_reset_confirm` with `django.contrib.auth.views.password_reset_confirm_uidb36` and also remove the `name` argument so it doesn't conflict with the new url:

```
url(r'^reset/(?P<uidb36>[0-9A-Za-z]+)-(?P<token>.+)/$',
    'django.contrib.auth.views.password_reset_confirm_uidb36'),
```

You can remove this URL pattern after your app has been deployed with Django 1.6 for *PASSWORD_RESET_TIMEOUT_DAYS*.

Default session serialization switched to JSON Historically, *django.contrib.sessions* used `pickle` to serialize session data before storing it in the backend. If you're using the *signed cookie session backend* and *SECRET_KEY* is known by an attacker (there isn't an inherent vulnerability in Django that would cause it to leak), the attacker could insert a string into his session which, when unpickled, executes arbitrary code on the server. The technique for doing so is simple and easily available on the internet. Although the cookie session storage signs the cookie-stored data to prevent tampering, a *SECRET_KEY* leak immediately escalates to a remote code execution vulnerability.

This attack can be mitigated by serializing session data using JSON rather than `pickle`. To facilitate this, Django 1.5.3 introduced a new setting, *SESSION_SERIALIZER*, to customize the session serialization format. For backwards compatibility, this setting defaulted to using `pickle` in Django 1.5.3, but we've changed the default to JSON in 1.6. If you upgrade and switch from pickle to JSON, sessions created before the upgrade will be lost. While JSON serialization does not support all Python objects like `pickle` does, we highly recommend using JSON-serialized sessions. Be aware of the following when checking your code to determine if JSON serialization will work for your application:

- JSON requires string keys, so you will likely run into problems if you are using non-string keys in `request.session`.
- Setting session expiration by passing `datetime` values to *set_expiry()* will not work as `datetime` values are not serializable in JSON. You can use integer values instead.

See the *Session serialization* documentation for more details.

Object Relational Mapper changes Django 1.6 contains many changes to the ORM. These changes fall mostly in three categories:

1. Bug fixes (e.g. proper join clauses for generic relations, query combining, join promotion, and join trimming fixes)
2. Preparation for new features. For example the ORM is now internally ready for multicolumn foreign keys.
3. General cleanup.

These changes can result in some compatibility problems. For example, some queries will now generate different table aliases. This can affect *QuerySet.extra()*. In addition some queries will now produce different results. An example is *exclude(condition)* where the condition is a complex one (referencing multijoins inside *Q objects*). In many cases the affected queries didn't produce correct results in Django 1.5 but do now. Unfortunately there are also cases that produce different results, but neither Django 1.5 nor 1.6 produce correct results.

Finally, there have been many changes to the ORM internal APIs.

Miscellaneous

- The `django.db.models.query.EmptyQuerySet` can't be instantiated any more - it is only usable as a marker class for checking if *none()* has been called: `isinstance(qs.none(), EmptyQuerySet)`

- If your CSS/JavaScript code used to access HTML input widgets by type, you should review it as `type='text'` widgets might be now output as `type='email'`, `type='url'` or `type='number'` depending on their corresponding field type.
- Form field's *`error_messages`* that contain a placeholder should now always use a named placeholder (`"Value '%(value)s' is too big"` instead of `"Value '%s' is too big"`). See the corresponding field documentation for details about the names of the placeholders. The changes in 1.6 particularly affect *`DecimalField`* and *`ModelMultipleChoiceField`*.
- Some *`error_messages`* for *`IntegerField`*, *`EmailField`*, `IPAddressField`, *`GenericIPAddressField`*, and *`SlugField`* have been suppressed because they duplicated error messages already provided by validators tied to the fields.
- Due to a change in the form validation workflow, *`TypedChoiceField`* `coerce` method should always return a value present in the `choices` field attribute. That limitation should be lift again in Django 1.7.
- There have been changes in the way timeouts are handled in cache backends. Explicitly passing in `timeout=None` no longer results in using the default timeout. It will now set a non-expiring timeout. Passing 0 into the memcache backend no longer uses the default timeout, and now will set-and-expire-immediately the value.
- The `django.contrib.flatpages` app used to set custom HTTP headers for debugging purposes. This functionality was not documented and made caching ineffective so it has been removed, along with its generic implementation, previously available in `django.core.xheaders`.
- The `XViewMiddleware` has been moved from `django.middleware.doc` to `django.contrib.admindocs.middleware` because it is an implementation detail of admindocs, proven not to be reusable in general.
- *`GenericIPAddressField`* will now only allow `blank` values if `null` values are also allowed. Creating a `GenericIPAddressField` where `blank` is allowed but `null` is not will trigger a model validation error because `blank` values are always stored as `null`. Previously, storing a `blank` value in a field which did not allow `null` would cause a database exception at runtime.
- If a `NoReverseMatch` exception is raised from a method when rendering a template, it is not silenced. For example, `{{ obj.view_href }}` will cause template rendering to fail if `view_href()` raises `NoReverseMatch`. There is no change to the *`{% url %}`* tag, it causes template rendering to fail like always when `NoReverseMatch` is raised.
- *`django.test.Client.logout()`* now calls *`django.contrib.auth.logout()`* which will send the *`user_logged_out()`* signal.
- *Authentication views* are now reversed by name, not their locations in `django.contrib.auth.views`. If you are using the views without a `name`, you should update your `urlpatterns` to use *`url()`* with the `name` parameter. For example:

```
(r'^reset/done/$', 'django.contrib.auth.views.password_reset_complete')
```

  becomes:

```
url(r'^reset/done/$', 'django.contrib.auth.views.password_reset_complete', name='password_reset_
```

- *`RedirectView`* now has a *pattern_name* attribute which allows it to choose the target by reversing the URL.
- In Django 1.4 and 1.5, a blank string was unintentionally not considered to be a valid password. This meant *`set_password()`* would save a blank password as an unusable password like *`set_unusable_password()`* does, and thus *`check_password()`* always returned `False` for blank passwords. This has been corrected in this release: blank passwords are now valid.
- The admin *`changelist_view`* previously accepted a `pop` GET parameter to signify it was to be displayed in a popup. This parameter has been renamed to `_popup` to be consistent with the rest of the admin views. You

should update your custom templates if they use the previous parameter name.

- *validate_email()* now accepts email addresses with `localhost` as the domain.
- The new *makemessages --keep-pot* option prevents deleting the temporary .pot file generated before creating the .po file.
- The undocumented `django.core.servers.basehttp.WSGIServerException` has been removed. Use `socket.error` provided by the standard library instead. This change was also released in Django 1.5.5.
- The signature of *django.views.generic.base.RedirectView.get_redirect_url()* has changed and now accepts positional arguments as well (`*args, **kwargs`). Any unnamed captured group will now be passed to `get_redirect_url()` which may result in a `TypeError` if you don't update the signature of your custom method.

Features deprecated in 1.6

Transaction management APIs Transaction management was completely overhauled in Django 1.6, and the current APIs are deprecated:

- `django.middleware.transaction.TransactionMiddleware`
- `django.db.transaction.autocommit`
- `django.db.transaction.commit_on_success`
- `django.db.transaction.commit_manually`
- the `TRANSACTIONS_MANAGED` setting

`django.contrib.comments` Django's comment framework has been deprecated and is no longer supported. It will be available in Django 1.6 and 1.7, and removed in Django 1.8. Most users will be better served with a custom solution, or a hosted product like Disqus.

The code formerly known as `django.contrib.comments` is still available in an external repository.

Support for PostgreSQL versions older than 8.4 The end of upstream support periods was reached in December 2011 for PostgreSQL 8.2 and in February 2013 for 8.3. As a consequence, Django 1.6 sets 8.4 as the minimum PostgreSQL version it officially supports.

You're strongly encouraged to use the most recent version of PostgreSQL available, because of performance improvements and to take advantage of the native streaming replication available in PostgreSQL 9.x.

Changes to `cycle` and `firstof` The template system generally escapes all variables to avoid XSS attacks. However, due to an accident of history, the *cycle* and *firstof* tags render their arguments as-is.

Django 1.6 starts a process to correct this inconsistency. The `future` template library provides alternate implementations of *cycle* and *firstof* that autoescape their inputs. If you're using these tags, you're encouraged to include the following line at the top of your templates to enable the new behavior:

```
{% load cycle from future %}
```

or:

```
{% load firstof from future %}
```

The tags implementing the old behavior have been deprecated, and in Django 1.8, the old behavior will be replaced with the new behavior. To ensure compatibility with future versions of Django, existing templates should be modified to use the `future` versions.

If necessary, you can temporarily disable auto-escaping with *`mark_safe()`* or *`{% autoescape off %}`*.

`CACHE_MIDDLEWARE_ANONYMOUS_ONLY` setting `CacheMiddleware` and `UpdateCacheMiddleware` used to provide a way to cache requests only if they weren't made by a logged-in user. This mechanism was largely ineffective because the middleware correctly takes into account the `Vary: Cookie` HTTP header, and this header is being set on a variety of occasions, such as:

- accessing the session, or
- using CSRF protection, which is turned on by default, or
- using a client-side library which sets cookies, like Google Analytics.

This makes the cache effectively work on a per-session basis regardless of the `CACHE_MIDDLEWARE_ANONYMOUS_ONLY` setting.

`SEND_BROKEN_LINK_EMAILS` setting *`CommonMiddleware`* used to provide basic reporting of broken links by email when `SEND_BROKEN_LINK_EMAILS` is set to `True`.

Because of intractable ordering problems between *`CommonMiddleware`* and *`LocaleMiddleware`*, this feature was split out into a new middleware: *`BrokenLinkEmailsMiddleware`*.

If you're relying on this feature, you should add `'django.middleware.common.BrokenLinkEmailsMiddleware'` to your *`MIDDLEWARE_CLASSES`* setting and remove `SEND_BROKEN_LINK_EMAILS` from your settings.

`_has_changed` method on widgets If you defined your own form widgets and defined the `_has_changed` method on a widget, you should now define this method on the form field itself.

`module_name` model `_meta` attribute `Model._meta.module_name` was renamed to `model_name`. Despite being a private API, it will go through a regular deprecation path.

`get_(add|change|delete)_permission` model `_meta` methods `Model._meta.get_(add|change|delete)_per` methods were deprecated. Even if they were not part of the public API they'll also go through a regular deprecation path. You can replace them with `django.contrib.auth.get_permission_codename('action', Model._meta)` where `'action'` is `'add'`, `'change'`, or `'delete'`.

`get_query_set` and similar methods renamed to `get_queryset` Methods that return a `QuerySet` such as `Manager.get_query_set` or `ModelAdmin.queryset` have been renamed to `get_queryset`.

If you are writing a library that implements, for example, a `Manager.get_query_set` method, and you need to support old Django versions, you should rename the method and conditionally add an alias with the old name:

```
class CustomManager(models.Manager):
    def get_queryset(self):
        pass # ...

    if django.VERSION < (1, 6):
        get_query_set = get_queryset

    # For Django >= 1.6, models.Manager provides a get_query_set fallback
    # that emits a warning when used.
```

If you are writing a library that needs to call the `get_queryset` method and must support old Django versions, you should write:

```
get_queryset = (some_manager.get_query_set
                if hasattr(some_manager, 'get_query_set')
                else some_manager.get_queryset)
return get_queryset() # etc
```

In the general case of a custom manager that both implements its own `get_queryset` method and calls that method, and needs to work with older Django versions, and libraries that have not been updated yet, it is useful to define a `get_queryset_compat` method as below and use it internally to your manager:

```
class YourCustomManager(models.Manager):
    def get_queryset(self):
        return YourCustomQuerySet() # for example

    if django.VERSION < (1, 6):
        get_query_set = get_queryset

    def active(self): # for example
        return self.get_queryset_compat().filter(active=True)

    def get_queryset_compat(self):
        get_queryset = (self.get_query_set
                        if hasattr(self, 'get_query_set')
                        else self.get_queryset)
        return get_queryset()
```

This helps to minimize the changes that are needed, but also works correctly in the case of subclasses (such as `RelatedManagers` from Django 1.5) which might override either `get_query_set` or `get_queryset`.

`shortcut` view and URLconf The `shortcut` view was moved from `django.views.defaults` to `django.contrib.contenttypes.views` shortly after the 1.0 release, but the old location was never deprecated. This oversight was corrected in Django 1.6 and you should now use the new location.

The URLconf `django.conf.urls.shortcut` was also deprecated. If you're including it in an URLconf, simply replace:

```
(r'^prefix/', include('django.conf.urls.shortcut')),
```

with:

```
(r'^prefix/(?P<content_type_id>\d+)/(?P<object_id>.*)/$', 'django.contrib.contenttypes.views.shortcut
```

`ModelForm` without `fields` or `exclude` Previously, if you wanted a *`ModelForm`* to use all fields on the model, you could simply omit the `Meta.fields` attribute, and all fields would be used.

This can lead to security problems where fields are added to the model and, unintentionally, automatically become editable by end users. In some cases, particular with boolean fields, it is possible for this problem to be completely invisible. This is a form of Mass assignment vulnerability.

For this reason, this behavior is deprecated, and using the `Meta.exclude` option is strongly discouraged. Instead, all fields that are intended for inclusion in the form should be listed explicitly in the `fields` attribute.

If this security concern really does not apply in your case, there is a shortcut to explicitly indicate that all fields should be used - use the special value `"__all__"` for the fields attribute:

```
class MyModelForm(ModelForm):
    class Meta:
        fields = "__all__"
        model = MyModel
```

If you have custom `ModelForms` that only need to be used in the admin, there is another option. The admin has its own methods for defining fields (`fieldsets` etc.), and so adding a list of fields to the `ModelForm` is redundant. Instead, simply omit the `Meta` inner class of the `ModelForm`, or omit the `Meta.model` attribute. Since the `ModelAdmin` subclass knows which model it is for, it can add the necessary attributes to derive a functioning `ModelForm`. This behavior also works for earlier Django versions.

`UpdateView` and `CreateView` without explicit fields The generic views *`CreateView`* and *`UpdateView`*, and anything else derived from *`ModelFormMixin`*, are vulnerable to the security problem described in the section above, because they can automatically create a `ModelForm` that uses all fields for a model.

For this reason, if you use these views for editing models, you must also supply the `fields` attribute (new in Django 1.6), which is a list of model fields and works in the same way as the *`ModelForm`* `Meta.fields` attribute. Alternatively, you can set the `form_class` attribute to a `ModelForm` that explicitly defines the fields to be used. Defining an `UpdateView` or `CreateView` subclass to be used with a model but without an explicit list of fields is deprecated.

Munging of help text of model form fields for `ManyToManyField` fields All special handling of the `help_text` attribute of `ManyToManyField` model fields performed by standard model or model form fields as described in *Help text of model form fields for ManyToManyField fields* above is deprecated and will be removed in Django 1.8.

Help text of these fields will need to be handled either by applications, custom form fields or widgets, just like happens with the rest of the model field types.

9.1.6 1.5 release

Django 1.5.12 release notes

January 2, 2015

Django 1.5.12 fixes a regression in the 1.5.9 security release.

Bugfixes

- Fixed a regression with dynamically generated inlines and allowed field references in the admin (#23754).

Django 1.5.11 release notes

October 22, 2014

Django 1.5.11 fixes a couple regressions in the 1.5.9 security release.

Bugfixes

- Allowed related many-to-many fields to be referenced in the admin (#23604).
- Allowed inline and hidden references to admin fields (#23431).

Django 1.5.10 release notes

September 2, 2014

Django 1.5.10 fixes a regression in the 1.5.9 security release.

Bugfixes

- Allowed inherited and m2m fields to be referenced in the admin (#22486)

Django 1.5.9 release notes

August 20, 2014

Django 1.5.9 fixes several security issues in 1.5.8.

`reverse()` could generate URLs pointing to other hosts

In certain situations, URL reversing could generate scheme-relative URLs (URLs starting with two slashes), which could unexpectedly redirect a user to a different host. An attacker could exploit this, for example, by redirecting users to a phishing site designed to ask for user's passwords.

To remedy this, URL reversing now ensures that no URL starts with two slashes (//), replacing the second slash with its URL encoded counterpart (%2F). This approach ensures that semantics stay the same, while making the URL relative to the domain and not to the scheme.

File upload denial-of-service

Before this release, Django's file upload handing in its default configuration may degrade to producing a huge number of `os.stat()` system calls when a duplicate filename is uploaded. Since `stat()` may invoke IO, this may produce a huge data-dependent slowdown that slowly worsens over time. The net result is that given enough time, a user with the ability to upload files can cause poor performance in the upload handler, eventually causing it to become very slow simply by uploading 0-byte files. At this point, even a slow network connection and few HTTP requests would be all that is necessary to make a site unavailable.

We've remedied the issue by changing the algorithm for generating file names if a file with the uploaded name already exists. *`Storage.get_available_name()`* now appends an underscore plus a random 7 character alphanumeric string (e.g. `"_x3a1gho"`), rather than iterating through an underscore followed by a number (e.g. `"_1"`, `"_2"`, etc.).

`RemoteUserMiddleware` session hijacking

When using the *`RemoteUserMiddleware`* and the `RemoteUserBackend`, a change to the `REMOTE_USER` header between requests without an intervening logout could result in the prior user's session being co-opted by the subsequent user. The middleware now logs the user out on a failed login attempt.

Data leakage via query string manipulation in `contrib.admin`

In older versions of Django it was possible to reveal any field's data by modifying the "popup" and "to_field" parameters of the query string on an admin change form page. For example, requesting a URL like `/admin/auth/user/?pop=1&t=password` and viewing the page's HTML allowed viewing the password hash

of each user. While the admin requires users to have permissions to view the change form pages in the first place, this could leak data if you rely on users having access to view only certain fields on a model.

To address the issue, an exception will now be raised if a `to_field` value that isn't a related field to a model that has been registered with the admin is specified.

Django 1.5.8 release notes

May 14, 2014

Django 1.5.8 fixes two security issues in 1.5.8.

Caches may incorrectly be allowed to store and serve private data

In certain situations, Django may allow caches to store private data related to a particular session and then serve that data to requests with a different session, or no session at all. This can lead to information disclosure and can be a vector for cache poisoning.

When using Django sessions, Django will set a `Vary: Cookie` header to ensure caches do not serve cached data to requests from other sessions. However, older versions of Internet Explorer (most likely only Internet Explorer 6, and Internet Explorer 7 if run on Windows XP or Windows Server 2003) are unable to handle the `Vary` header in combination with many content types. Therefore, Django would remove the header if the request was made by Internet Explorer.

To remedy this, the special behavior for these older Internet Explorer versions has been removed, and the `Vary` header is no longer stripped from the response. In addition, modifications to the `Cache-Control` header for all Internet Explorer requests with a `Content-Disposition` header have also been removed as they were found to have similar issues.

Malformed redirect URLs from user input not correctly validated

The validation for redirects did not correctly validate some malformed URLs, which are accepted by some browsers. This allows a user to be redirected to an unsafe URL unexpectedly.

Django relies on user input in some cases (e.g. *`django.contrib.auth.views.login()`*, `django.contrib.comments`, and i18n) to redirect the user to an "on success" URL. The security checks for these redirects (namely `django.utils.http.is_safe_url()`) did not correctly validate some malformed URLs, such as *http:\\\djangoproject.com*, which are accepted by some browsers with more liberal URL parsing.

To remedy this, the validation in `is_safe_url()` has been tightened to be able to handle and correctly validate these malformed URLs.

Django 1.5.7 release notes

April 28, 2014

Django 1.5.7 fixes a regression in the 1.5.6 security release.

Bugfixes

- Restored the ability to `reverse()` views created using `functools.partial()` (#22486).

Django 1.5.6 release notes

April 21, 2014

Django 1.5.6 fixes several bugs in 1.5.5, including three security issues.

Unexpected code execution using `reverse()`

Django's URL handling is based on a mapping of regex patterns (representing the URLs) to callable views, and Django's own processing consists of matching a requested URL against those patterns to determine the appropriate view to invoke.

Django also provides a convenience function – `reverse()` – which performs this process in the opposite direction. The `reverse()` function takes information about a view and returns a URL which would invoke that view. Use of `reverse()` is encouraged for application developers, as the output of `reverse()` is always based on the current URL patterns, meaning developers do not need to change other code when making changes to URLs.

One argument signature for `reverse()` is to pass a dotted Python path to the desired view. In this situation, Django will import the module indicated by that dotted path as part of generating the resulting URL. If such a module has import-time side effects, those side effects will occur.

Thus it is possible for an attacker to cause unexpected code execution, given the following conditions:

1. One or more views are present which construct a URL based on user input (commonly, a "next" parameter in a querystring indicating where to redirect upon successful completion of an action).
2. One or more modules are known to an attacker to exist on the server's Python import path, which perform code execution with side effects on importing.

To remedy this, `reverse()` will now only accept and import dotted paths based on the view-containing modules listed in the project's URL pattern configuration, so as to ensure that only modules the developer intended to be imported in this fashion can or will be imported.

Caching of anonymous pages could reveal CSRF token

Django includes both a caching framework and a system for preventing cross-site request forgery (CSRF) attacks. The CSRF-protection system is based on a random nonce sent to the client in a cookie which must be sent by the client on future requests and, in forms, a hidden value which must be submitted back with the form.

The caching framework includes an option to cache responses to anonymous (i.e., unauthenticated) clients.

When the first anonymous request to a given page is by a client which did not have a CSRF cookie, the cache framework will also cache the CSRF cookie and serve the same nonce to other anonymous clients who do not have a CSRF cookie. This can allow an attacker to obtain a valid CSRF cookie value and perform attacks which bypass the check for the cookie.

To remedy this, the caching framework will no longer cache such responses. The heuristic for this will be:

1. If the incoming request did not submit any cookies, and
2. If the response did send one or more cookies, and
3. If the `Vary: Cookie` header is set on the response, then the response will not be cached.

MySQL typecasting

The MySQL database is known to "typecast" on certain queries; for example, when querying a table which contains string values, but using a query which filters based on an integer value, MySQL will first silently coerce the strings to integers and return a result based on that.

If a query is performed without first converting values to the appropriate type, this can produce unexpected results, similar to what would occur if the query itself had been manipulated.

Django's model field classes are aware of their own types and most such classes perform explicit conversion of query arguments to the correct database-level type before querying. However, three model field classes did not correctly convert their arguments:

- *FilePathField*
- *GenericIPAddressField*
- IPAddressField

These three fields have been updated to convert their arguments to the correct types before querying.

Additionally, developers of custom model fields are now warned via documentation to ensure their custom field classes will perform appropriate type conversions, and users of the *raw()* and *extra()* query methods – which allow the developer to supply raw SQL or SQL fragments – will be advised to ensure they perform appropriate manual type conversions prior to executing queries.

Bugfixes

- Fixed *ModelBackend* raising UnboundLocalError if *get_user_model()* raised an error (#21439).

Additionally, Django's vendored version of six, *django.utils.six*, has been upgraded to the latest release (1.6.1).

Django 1.5.5 release notes

October 23, 2013

Django 1.5.5 fixes a couple security-related bugs and several other bugs in the 1.5 series.

Readdressed denial-of-service via password hashers

Django 1.5.4 imposes a 4096-byte limit on passwords in order to mitigate a denial-of-service attack through submission of bogus but extremely large passwords. In Django 1.5.5, we've reverted this change and instead improved the speed of our PBKDF2 algorithm by not rehashing the key on every iteration.

Properly rotate CSRF token on login

This behavior introduced as a security hardening measure in Django 1.5.2 did not work properly and is now fixed.

Bugfixes

- Fixed a data corruption bug with datetime_safe.datetime.combine (#21256).
- Fixed a Python 3 incompatibility in django.utils.text.unescape_entities() (#21185).

- Fixed a couple data corruption issues with `QuerySet` edge cases under Oracle and MySQL (#21203, #21126).
- Fixed crashes when using combinations of `annotate()`, `select_related()`, and `only()` (#16436).

Backwards incompatible changes

- The undocumented `django.core.servers.basehttp.WSGIServerException` has been removed. Use `socket.error` provided by the standard library instead.

Django 1.5.4 release notes

September 14, 2013

This is Django 1.5.4, the fourth release in the Django 1.5 series. It addresses two security issues and one bug.

Denial-of-service via password hashers

In previous versions of Django, no limit was imposed on the plaintext length of a password. This allowed a denial-of-service attack through submission of bogus but extremely large passwords, tying up server resources performing the (expensive, and increasingly expensive with the length of the password) calculation of the corresponding hash.

As of 1.5.4, Django's authentication framework imposes a 4096-byte limit on passwords, and will fail authentication with any submitted password of greater length.

Corrected usage of `sensitive_post_parameters()` in `django.contrib.auth`'s admin

The decoration of the `add_view` and `user_change_password` user admin views with *`sensitive_post_parameters()`* did not include *`method_decorator()`* (required since the views are methods) resulting in the decorator not being properly applied. This usage has been fixed and *`sensitive_post_parameters()`* will now throw an exception if it's improperly used.

Bugfixes

- Fixed a bug that prevented a `QuerySet` that uses *`prefetch_related()`* from being pickled and unpickled more than once (the second pickling attempt raised an exception) (#21102).

Django 1.5.3 release notes

September 10, 2013

This is Django 1.5.3, the third release in the Django 1.5 series. It addresses one security issue and also contains an opt-in feature to enhance the security of *`django.contrib.sessions`*.

Directory traversal vulnerability in `ssi` template tag

In previous versions of Django it was possible to bypass the `ALLOWED_INCLUDE_ROOTS` setting used for security with the `ssi` template tag by specifying a relative path that starts with one of the allowed roots. For example, if `ALLOWED_INCLUDE_ROOTS = ("/var/www",)` the following would be possible:

```
{% ssi "/var/www/../../etc/passwd" %}
```

In practice this is not a very common problem, as it would require the template author to put the `ssi` file in a user-controlled variable, but it's possible in principle.

Mitigating a remote-code execution vulnerability in `django.contrib.sessions`

`django.contrib.sessions` currently uses `pickle` to serialize session data before storing it in the backend. If you're using the *signed cookie session backend* and *`SECRET_KEY`* is known by an attacker (there isn't an inherent vulnerability in Django that would cause it to leak), the attacker could insert a string into his session which, when unpickled, executes arbitrary code on the server. The technique for doing so is simple and easily available on the internet. Although the cookie session storage signs the cookie-stored data to prevent tampering, a *`SECRET_KEY`* leak immediately escalates to a remote code execution vulnerability.

This attack can be mitigated by serializing session data using JSON rather than `pickle`. To facilitate this, Django 1.5.3 introduces a new setting, *`SESSION_SERIALIZER`*, to customize the session serialization format. For backwards compatibility, this setting defaults to using `pickle`. While JSON serialization does not support all Python objects like `pickle` does, we highly recommend switching to JSON-serialized values. Also, as JSON requires string keys, you will likely run into problems if you are using non-string keys in `request.session`. See the *Session serialization* documentation for more details.

Django 1.5.2 release notes

August 13, 2013

This is Django 1.5.2, a bugfix and security release for Django 1.5.

Mitigated possible XSS attack via user-supplied redirect URLs

Django relies on user input in some cases (e.g. *`django.contrib.auth.views.login()`*, `django.contrib.comments`, and i18n) to redirect the user to an "on success" URL. The security checks for these redirects (namely `django.utils.http.is_safe_url()`) didn't check if the scheme is `http(s)` and as such allowed `javascript:...` URLs to be entered. If a developer relied on `is_safe_url()` to provide safe redirect targets and put such a URL into a link, they could suffer from a XSS attack. This bug doesn't affect Django currently, since we only put this URL into the `Location` response header and browsers seem to ignore JavaScript there.

XSS vulnerability in `django.contrib.admin`

If a *`URLField`* is used in Django 1.5, it displays the current value of the field and a link to the target on the admin change page. The display routine of this widget was flawed and allowed for XSS.

Bugfixes

- Fixed a crash with *`prefetch_related()`* (#19607) as well as some `pickle` regressions with `prefetch_related` (#20157 and #20257).
- Fixed a regression in *`django.contrib.gis`* in the Google Map output on Python 3 (#20773).
- Made `DjangoTestSuiteRunner.setup_databases` properly handle aliases for the default database (#19940) and prevented `teardown_databases` from attempting to tear down aliases (#20681).

- Fixed the `django.core.cache.backends.memcached.MemcachedCache` backend's `get_many()` method on Python 3 (#20722).
- Fixed *`django.contrib.humanize`* translation syntax errors. Affected languages: Mexican Spanish, Mongolian, Romanian, Turkish (#20695).
- Added support for wheel packages (#19252).
- The CSRF token now rotates when a user logs in.
- Some Python 3 compatibility fixes including #20212 and #20025.
- Fixed some rare cases where *`get()`* exceptions recursed infinitely (#20278).
- *`makemessages`* no longer crashes with `UnicodeDecodeError` (#20354).
- Fixed `geojson` detection with SpatiaLite.
- *`assertContains()`* once again works with binary content (#20237).
- Fixed *`ManyToManyField`* if it has a unicode `name` parameter (#20207).
- Ensured that the WSGI request's path is correctly based on the `SCRIPT_NAME` environment variable or the *`FORCE_SCRIPT_NAME`* setting, regardless of whether or not either has a trailing slash (#20169).
- Fixed an obscure bug with the *`override_settings()`* decorator. If you hit an `AttributeError: 'Settings' object has no attribute '_original_allowed_hosts'` exception, it's probably fixed (#20636).

Django 1.5.1 release notes

March 28, 2013

This is Django 1.5.1, a bugfix release for Django 1.5. It's completely backwards compatible with Django 1.5, but includes a handful of fixes.

The biggest fix is for a memory leak introduced in Django 1.5. Under certain circumstances, repeated iteration over querysets could leak memory - sometimes quite a bit of it. If you'd like more information, the details are in our ticket tracker (and in a related issue in Python itself).

If you've noticed memory problems under Django 1.5, upgrading to 1.5.1 should fix those issues.

Django 1.5.1 also includes a couple smaller fixes:

- Module-level warnings emitted during tests are no longer silently hidden (#18985).
- Prevented filtering on password hashes in the user admin (#20078).

Django 1.5 release notes

February 26, 2013

Welcome to Django 1.5!

These release notes cover the *new features*, as well as some *backwards incompatible changes* you'll want to be aware of when upgrading from Django 1.4 or older versions. We've also dropped some features, which are detailed in *our deprecation plan*, and we've *begun the deprecation process for some features*.

Overview

The biggest new feature in Django 1.5 is the *configurable User model*. Before Django 1.5, applications that wanted to use Django's auth framework (`django.contrib.auth`) were forced to use Django's definition of a "user". In Django 1.5, you can now swap out the `User` model for one that you write yourself. This could be a simple extension to the existing `User` model – for example, you could add a Twitter or Facebook ID field – or you could completely replace the `User` with one totally customized for your site.

Django 1.5 is also the first release with *Python 3 support*! We're labeling this support "experimental" because we don't yet consider it production-ready, but everything's in place for you to start porting your apps to Python 3. Our next release, Django 1.6, will support Python 3 without reservations.

Other notable new features in Django 1.5 include:

- *Support for saving a subset of model's fields* - `Model.save()` now accepts an `update_fields` argument, letting you specify which fields are written back to the database when you call `save()`. This can help in high-concurrency operations, and can improve performance.
- Better *support for streaming responses* via the new `StreamingHttpResponse` response class.
- *GeoDjango* now supports PostGIS 2.0.
- ... and more; *see below*.

Wherever possible we try to introduce new features in a backwards-compatible manner per our API stability policy. However, as with previous releases, Django 1.5 ships with some minor *backwards incompatible changes*; people upgrading from previous versions of Django should read that list carefully.

One deprecated feature worth noting is the shift to "new-style" `url` tag. Prior to Django 1.3, syntax like `{% url myview %}` was interpreted incorrectly (Django considered `"myview"` to be a literal name of a view, not a template variable named `myview`). Django 1.3 and above introduced the `{% load url from future %}` syntax to bring in the corrected behavior where `myview` was seen as a variable.

The upshot of this is that if you are not using `{% load url from future %}` in your templates, you'll need to change tags like `{% url myview %}` to `{% url "myview" %}`. If you *were* using `{% load url from future %}` you can simply remove that line under Django 1.5

Python compatibility

Django 1.5 requires Python 2.6.5 or above, though we **highly recommend** Python 2.7.3 or above. Support for Python 2.5 and below has been dropped.

This change should affect only a small number of Django users, as most operating-system vendors today are shipping Python 2.6 or newer as their default version. If you're still using Python 2.5, however, you'll need to stick to Django 1.4 until you can upgrade your Python version. Per our support policy, Django 1.4 will continue to receive security support until the release of Django 1.6.

Django 1.5 does not run on a Jython final release, because Jython's latest release doesn't currently support Python 2.6. However, Jython currently does offer an alpha release featuring 2.7 support, and Django 1.5 supports that alpha release.

Python 3 support Django 1.5 introduces support for Python 3 - specifically, Python 3.2 and above. This comes in the form of a **single** codebase; you don't need to install a different version of Django on Python 3. This means that you can write applications targeted for just Python 2, just Python 3, or single applications that support both platforms.

However, we're labeling this support "experimental" for now: although it's received extensive testing via our automated test suite, it's received very little real-world testing. We've done our best to eliminate bugs, but we can't be sure we covered all possible uses of Django.

Some features of Django aren't available because they depend on third-party software that hasn't been ported to Python 3 yet, including:

- the MySQL database backend (depends on MySQLdb)
- *ImageField* (depends on PIL)
- *LiveServerTestCase* (depends on Selenium WebDriver)

Further, Django's more than a web framework; it's an ecosystem of pluggable components. At this point, very few third-party applications have been ported to Python 3, so it's unlikely that a real-world application will have all its dependencies satisfied under Python 3.

Thus, we're recommending that Django 1.5 not be used in production under Python 3. Instead, use this opportunity to begin porting applications to Python 3. If you're an author of a pluggable component, we encourage you to start porting now.

We plan to offer first-class, production-ready support for Python 3 in our next release, Django 1.6.

What's new in Django 1.5

Configurable User model In Django 1.5, you can now use your own model as the store for user-related data. If your project needs a username with more than 30 characters, or if you want to store user's names in a format other than first name/last name, or you want to put custom profile information onto your User object, you can now do so.

If you have a third-party reusable application that references the User model, you may need to make some changes to the way you reference User instances. You should also document any specific features of the User model that your application relies upon.

See the *documentation on custom User models* for more details.

Support for saving a subset of model's fields The method *Model.save()* has a new keyword argument `update_fields`. By using this argument it is possible to save only a select list of model's fields. This can be useful for performance reasons or when trying to avoid overwriting concurrent changes.

Deferred instances (those loaded by `.only()` or `.defer()`) will automatically save just the loaded fields. If any field is set manually after load, that field will also get updated on save.

See the *Model.save()* documentation for more details.

Caching of related model instances When traversing relations, the ORM will avoid re-fetching objects that were previously loaded. For example, with the tutorial's models:

```
>>> first_poll = Poll.objects.all()[0]
>>> first_choice = first_poll.choice_set.all()[0]
>>> first_choice.poll is first_poll
True
```

In Django 1.5, the third line no longer triggers a new SQL query to fetch `first_choice.poll`; it was set by the second line.

For one-to-one relationships, both sides can be cached. For many-to-one relationships, only the single side of the relationship can be cached. This is particularly helpful in combination with `prefetch_related`.

Explicit support for streaming responses Before Django 1.5, it was possible to create a streaming response by passing an iterator to *HttpResponse*. But this was unreliable: any middleware that accessed the *content* attribute would consume the iterator prematurely.

You can now explicitly generate a streaming response with the new *StreamingHttpResponse* class. This class exposes a *streaming_content* attribute which is an iterator.

Since *StreamingHttpResponse* does not have a `content` attribute, middleware that needs access to the response content must test for streaming responses and behave accordingly.

`{% verbatim %}` template tag To make it easier to deal with JavaScript templates which collide with Django's syntax, you can now use the *verbatim* block tag to avoid parsing the tag's content.

Retrieval of `ContentType` instances associated with proxy models The methods *ContentTypeManager.get_for_model()* and *ContentTypeManager.get_for_models()* have a new keyword argument – respectively `for_concrete_model` and `for_concrete_models`. By passing `False` using this argument it is now possible to retrieve the *ContentType* associated with proxy models.

New `view` variable in class-based views context In all generic class-based views (or any class-based view inheriting from `ContextMixin`), the context dictionary contains a `view` variable that points to the `View` instance.

GeoDjango

- *LineString* and *MultiLineString* GEOS objects now support the *interpolate()* and *project()* methods (so-called linear referencing).
- The `wkb` and `hex` properties of *GEOSGeometry* objects preserve the Z dimension.
- Support for PostGIS 2.0 has been added and support for GDAL < 1.5 has been dropped.

New tutorials Additions to the docs include a revamped Tutorial 3 and a new tutorial on testing. A new section, "Advanced Tutorials", offers How to write reusable apps as well as a step-by-step guide for new contributors in Writing your first patch for Django.

Minor features Django 1.5 also includes several smaller improvements worth noting:

- The template engine now interprets `True`, `False` and `None` as the corresponding Python objects.
- *django.utils.timezone* provides a helper for converting aware datetimes between time zones. See *localtime()*.
- The generic views support OPTIONS requests.
- Management commands do not raise `SystemExit` any more when called by code from *call_command()*. Any exception raised by the command (mostly *CommandError*) is propagated.

  Moreover, when you output errors or messages in your custom commands, you should now use `self.stdout.write('message')` and `self.stderr.write('error')` (see the note on *management commands output*).
- The dumpdata management command outputs one row at a time, preventing out-of-memory errors when dumping large datasets.
- In the localflavor for Canada, "pq" was added to the acceptable codes for Quebec. It's an old abbreviation.
- The *receiver* decorator is now able to connect to more than one signal by supplying a list of signals.
- In the admin, you can now filter users by groups which they are members of.
- *QuerySet.bulk_create()* now has a batch_size argument. By default the batch_size is unlimited except for SQLite where single batch is limited so that 999 parameters per query isn't exceeded.

- The *LOGIN_URL* and *LOGIN_REDIRECT_URL* settings now also accept view function names and *named URL patterns*. This allows you to reduce configuration duplication. More information can be found in the *login_required()* documentation.
- Django now provides a mod_wsgi auth handler.
- The *QuerySet.delete()* and *Model.delete()* can now take fast-path in some cases. The fast-path allows for less queries and less objects fetched into memory. See *QuerySet.delete()* for details.
- An instance of `ResolverMatch` is stored on the request as `resolver_match`.
- By default, all logging messages reaching the `django` logger when *DEBUG* is `True` are sent to the console (unless you redefine the logger in your *LOGGING* setting).
- When using *RequestContext*, it is now possible to look up permissions by using `{% if 'someapp.someperm' in perms %}` in templates.
- It's not required any more to have `404.html` and `500.html` templates in the root templates directory. Django will output some basic error messages for both situations when those templates are not found. Of course, it's still recommended as good practice to provide those templates in order to present pretty error pages to the user.
- *django.contrib.auth* provides a new signal that is emitted whenever a user fails to login successfully. See *user_login_failed*
- The new *loaddata --ignorenonexistent* option ignore data for fields that no longer exist.
- *assertXMLEqual()* and *assertXMLNotEqual()* new assertions allow you to test equality for XML content at a semantic level, without caring for syntax differences (spaces, attribute order, etc.).
- RemoteUserMiddleware now forces logout when the REMOTE_USER header disappears during the same browser session.
- The *cache-based session backend* can store session data in a non-default cache.
- Multi-column indexes can now be created on models. Read the *index_together* documentation for more information.
- During Django's logging configuration verbose Deprecation warnings are enabled and warnings are captured into the logging system. Logged warnings are routed through the `console` logging handler, which by default requires *DEBUG* to be True for output to be generated. The result is that DeprecationWarnings should be printed to the console in development environments the way they have been in Python versions < 2.7.
- The API for *django.contrib.admin.ModelAdmin.message_user()* method has been modified to accept additional arguments adding capabilities similar to *django.contrib.messages.add_message()*. This is useful for generating error messages from admin actions.
- The admin's list filters can now be customized per-request thanks to the new *django.contrib.admin.ModelAdmin.get_list_filter()* method.

Backwards incompatible changes in 1.5

: In addition to the changes outlined in this section, be sure to review the *deprecation plan* for any features that have been removed. If you haven't updated your code within the deprecation timeline for a given feature, its removal may appear as a backwards incompatible change.

ALLOWED_HOSTS required in production The new *ALLOWED_HOSTS* setting validates the request's `Host` header and protects against host-poisoning attacks. This setting is now required whenever *DEBUG* is `False`, or else *django.http.HttpRequest.get_host()* will raise *SuspiciousOperation*. For more details see the *full documentation* for the new setting.

Managers on abstract models Abstract models are able to define a custom manager, and that manager *will be inherited by any concrete models extending the abstract model*. However, if you try to use the abstract model to call a method on the manager, an exception will now be raised. Previously, the call would have been permitted, but would have failed as soon as any database operation was attempted (usually with a "table does not exist" error from the database).

If you have functionality on a manager that you have been invoking using the abstract class, you should migrate that logic to a Python `staticmethod` or `classmethod` on the abstract class.

Context in year archive class-based views For consistency with the other date-based generic views, *YearArchiveView* now passes `year` in the context as a `datetime.date` rather than a string. If you are using `{{ year }}` in your templates, you must replace it with `{{ year|date:"Y" }}`.

`next_year` and `previous_year` were also added in the context. They are calculated according to `allow_empty` and `allow_future`.

Context in year and month archive class-based views *YearArchiveView* and *MonthArchiveView* were documented to provide a `date_list` sorted in ascending order in the context, like their function-based predecessors, but it actually was in descending order. In 1.5, the documented order was restored. You may want to add (or remove) the `reversed` keyword when you're iterating on `date_list` in a template:

```
{% for date in date_list reversed %}
```

ArchiveIndexView still provides a `date_list` in descending order.

Context in TemplateView For consistency with the design of the other generic views, *TemplateView* no longer passes a `params` dictionary into the context, instead passing the variables from the URLconf directly into the context.

Non-form data in HTTP requests *request.POST* will no longer include data posted via HTTP requests with non form-specific content-types in the header. In prior versions, data posted with content-types other than `multipart/form-data` or `application/x-www-form-urlencoded` would still end up represented in the *request.POST* attribute. Developers wishing to access the raw POST data for these cases, should use the *request.body* attribute instead.

request_finished signal Django used to send the *request_finished* signal as soon as the view function returned a response. This interacted badly with *streaming responses* that delay content generation.

This signal is now sent after the content is fully consumed by the WSGI gateway. This might be backwards incompatible if you rely on the signal being fired before sending the response content to the client. If you do, you should consider using middleware instead.

: Some WSGI servers and middleware do not always call `close` on the response object after handling a request, most notably uWSGI prior to 1.2.6 and Sentry's error reporting middleware up to 2.0.7. In those cases the `request_finished` signal isn't sent at all. This can result in idle connections to database and memcache servers.

OPTIONS, PUT and DELETE requests in the test client Unlike GET and POST, these HTTP methods aren't implemented by web browsers. Rather, they're used in APIs, which transfer data in various formats such as JSON or XML. Since such requests may contain arbitrary data, Django doesn't attempt to decode their body.

However, the test client used to build a query string for OPTIONS and DELETE requests like for GET, and a request body for PUT requests like for POST. This encoding was arbitrary and inconsistent with Django's behavior when it receives the requests, so it was removed in Django 1.5.

If you were using the `data` parameter in an OPTIONS or a DELETE request, you must convert it to a query string and append it to the `path` parameter.

If you were using the `data` parameter in a PUT request without a `content_type`, you must encode your data before passing it to the test client and set the `content_type` argument.

System version of `simplejson` no longer used *As explained below*, Django 1.5 deprecates `django.utils.simplejson` in favor of Python 2.6's built-in `json` module. In theory, this change is harmless. Unfortunately, because of incompatibilities between versions of `simplejson`, it may trigger errors in some circumstances.

JSON-related features in Django 1.4 always used `django.utils.simplejson`. This module was actually:

- A system version of `simplejson`, if one was available (ie. `import simplejson` works), if it was more recent than Django's built-in copy or it had the C speedups, or
- The `json` module from the standard library, if it was available (ie. Python 2.6 or greater), or
- A built-in copy of version 2.0.7 of `simplejson`.

In Django 1.5, those features use Python's `json` module, which is based on version 2.0.9 of `simplejson`.

There are no known incompatibilities between Django's copy of version 2.0.7 and Python's copy of version 2.0.9. However, there are some incompatibilities between other versions of `simplejson`:

- While the `simplejson` API is documented as always returning unicode strings, the optional C implementation can return a byte string. This was fixed in Python 2.7.
- `simplejson.JSONEncoder` gained a `namedtuple_as_object` keyword argument in version 2.2.

More information on these incompatibilities is available in ticket #18023.

The net result is that, if you have installed `simplejson` and your code uses Django's serialization internals directly – for instance `django.core.serializers.json.DjangoJSONEncoder`, the switch from `simplejson` to `json` could break your code. (In general, changes to internals aren't documented; we're making an exception here.)

At this point, the maintainers of Django believe that using `json` from the standard library offers the strongest guarantee of backwards-compatibility. They recommend to use it from now on.

String types of hasher method parameters If you have written a *custom password hasher*, your `encode()`, `verify()` or `safe_summary()` methods should accept Unicode parameters (`password`, `salt` or `encoded`). If any of the hashing methods need byte strings, you can use the *`force_bytes()`* utility to encode the strings.

Validation of previous_page_number and next_page_number When using object pagination, the `previous_page_number()` and `next_page_number()` methods of the *`Page`* object did not check if the returned number was inside the existing page range. It does check it now and raises an *`InvalidPage`* exception when the number is either too low or too high.

Behavior of autocommit database option on PostgreSQL changed PostgreSQL's autocommit option didn't work as advertised previously. It did work for single transaction block, but after the first block was left the autocommit behavior was never restored. This bug is now fixed in 1.5. While this is only a bug fix, it is worth checking your applications behavior if you are using PostgreSQL together with the autocommit option.

Session not saved on 500 responses Django's session middleware will skip saving the session data if the response's status code is 500.

Email checks on failed admin login Prior to Django 1.5, if you attempted to log into the admin interface and mistakenly used your email address instead of your username, the admin interface would provide a warning advising that your email address was not your username. In Django 1.5, the introduction of *custom User models* has required the removal of this warning. This doesn't change the login behavior of the admin site; it only affects the warning message that is displayed under one particular mode of login failure.

Changes in tests execution Some changes have been introduced in the execution of tests that might be backward-incompatible for some testing setups:

Database flushing in `django.test.TransactionTestCase` Previously, the test database was truncated *before* each test run in a `TransactionTestCase`.

In order to be able to run unit tests in any order and to make sure they are always isolated from each other, `TransactionTestCase` will now reset the database *after* each test run instead.

No more implicit DB sequences reset `TransactionTestCase` tests used to reset primary key sequences automatically together with the database flushing actions described above.

This has been changed so no sequences are implicitly reset. This can cause `TransactionTestCase` tests that depend on hard-coded primary key values to break.

The new `reset_sequences` attribute can be used to force the old behavior for `TransactionTestCase` that might need it.

Ordering of tests In order to make sure all `TestCase` code starts with a clean database, tests are now executed in the following order:

- First, all unittests (including `unittest.TestCase`, `SimpleTestCase`, `TestCase` and `TransactionTestCase`) are run with no particular ordering guaranteed nor enforced among them.
- Then any other tests (e.g. doctests) that may alter the database without restoring it to its original state are run.

This should not cause any problems unless you have existing doctests which assume a `TransactionTestCase` executed earlier left some database state behind or unit tests that rely on some form of state being preserved after the execution of other tests. Such tests are already very fragile, and must now be changed to be able to run independently.

***cleaned_data* dictionary kept for invalid forms** The `cleaned_data` dictionary is now always present after form validation. When the form doesn't validate, it contains only the fields that passed validation. You should test the success of the validation with the `is_valid()` method and not with the presence or absence of the `cleaned_data` attribute on the form.

Behavior of `syncdb` with multiple databases `syncdb` now queries the database routers to determine if content types (when *`contenttypes`* is enabled) and permissions (when *`auth`* is enabled) should be created in the target database. Previously, it created them in the default database, even when another database was specified with the `--database` option.

If you use `syncdb` on multiple databases, you should ensure that your routers allow synchronizing content types and permissions to only one of them. See the docs on the *behavior of contrib apps with multiple databases* for more information.

XML deserializer will not parse documents with a DTD In order to prevent exposure to denial-of-service attacks related to external entity references and entity expansion, the XML model deserializer now refuses to parse XML documents containing a DTD (DOCTYPE definition). Since the XML serializer does not output a DTD, this will not impact typical usage, only cases where custom-created XML documents are passed to Django's model deserializer.

Formsets default `max_num` A (default) value of `None` for the `max_num` argument to a formset factory no longer defaults to allowing any number of forms in the formset. Instead, in order to prevent memory-exhaustion attacks, it now defaults to a limit of 1000 forms. This limit can be raised by explicitly setting a higher value for `max_num`.

Miscellaneous

- *`django.forms.ModelMultipleChoiceField`* now returns an empty `QuerySet` as the empty value instead of an empty list.
- *`int_to_base36()`* properly raises a `TypeError` instead of `ValueError` for non-integer inputs.
- The `slugify` template filter is now available as a standard python function at *`django.utils.text.slugify()`*. Similarly, `remove_tags` is available at `django.utils.html.remove_tags()`.
- Uploaded files are no longer created as executable by default. If you need them to be executable change *`FILE_UPLOAD_PERMISSIONS`* to your needs. The new default value is `0o666` (octal) and the current umask value is first masked out.
- The *`F expressions`* supported bitwise operators by `&` and `|`. These operators are now available using `.bitand()` and `.bitor()` instead. The removal of `&` and `|` was done to be consistent with *`Q() expressions`* and `QuerySet` combining where the operators are used as boolean AND and OR operators.
- In a `filter()` call, when *`F expressions`* contained lookups spanning multi-valued relations, they didn't always reuse the same relations as other lookups along the same chain. This was changed, and now F() expressions will always use the same relations as other lookups within the same `filter()` call.
- The *`csrf_token`* template tag is no longer enclosed in a div. If you need HTML validation against pre-HTML5 Strict DTDs, you should add a div around it in your pages.
- The template tags library `adminmedia`, which only contained the deprecated template tag `{% admin_media_prefix %}`, was removed. Attempting to load it with `{% load adminmedia %}` will fail. If your templates still contain that line you must remove it.
- Because of an implementation oversight, it was possible to use django.contrib.redirects without enabling django.contrib.sites. This isn't allowed any longer. If you're using `django.contrib.redirects`, make sure *`INSTALLED_APPS`* contains `django.contrib.sites`.
- *`BoundField.label_tag`* now escapes its `contents` argument. To avoid the HTML escaping, use *`django.utils.safestring.mark_safe()`* on the argument before passing it.
- Accessing reverse one-to-one relations fetched via *`select_related()`* now raises *`DoesNotExist`* instead of returning `None`.

Features deprecated in 1.5

`django.contrib.localflavor` The localflavor contrib app has been split into separate packages. `django.contrib.localflavor` itself will be removed in Django 1.6, after an accelerated deprecation.

The new packages are available on GitHub. The core team cannot efficiently maintain these packages in the long term — it spans just a dozen countries at this time; similar to translations, maintenance will be handed over to interested members of the community.

`django.contrib.markup` The markup contrib module has been deprecated and will follow an accelerated deprecation schedule. Direct use of Python markup libraries or 3rd party tag libraries is preferred to Django maintaining this functionality in the framework.

`AUTH_PROFILE_MODULE` With the introduction of *custom User models*, there is no longer any need for a built-in mechanism to store user profile data.

You can still define user profiles models that have a one-to-one relation with the User model - in fact, for many applications needing to associate data with a User account, this will be an appropriate design pattern to follow. However, the `AUTH_PROFILE_MODULE` setting, and the `django.contrib.auth.models.User.get_profile()` method for accessing the user profile model, should not be used any longer.

Streaming behavior of `HttpResponse` Django 1.5 deprecates the ability to stream a response by passing an iterator to *`HttpResponse`*. If you rely on this behavior, switch to *`StreamingHttpResponse`*. See *Explicit support for streaming responses* above.

In Django 1.7 and above, the iterator will be consumed immediately by *`HttpResponse`*.

`django.utils.simplejson` Since Django 1.5 drops support for Python 2.5, we can now rely on the `json` module being available in Python's standard library, so we've removed our own copy of `simplejson`. You should now import `json` instead of `django.utils.simplejson`.

Unfortunately, this change might have unwanted side-effects, because of incompatibilities between versions of `simplejson` – see the *backwards-incompatible changes* section. If you rely on features added to `simplejson` after it became Python's `json`, you should import `simplejson` explicitly.

`django.utils.encoding.StrAndUnicode` The `django.utils.encoding.StrAndUnicode` mix-in has been deprecated. Define a `__str__` method and apply the *`python_2_unicode_compatible()`* decorator instead.

`django.utils.itercompat.product` The `django.utils.itercompat.product` function has been deprecated. Use the built-in `itertools.product()` instead.

`cleanup` management command The `cleanup` management command has been deprecated and replaced by *`clearsessions`*.

`daily_cleanup.py` script The undocumented `daily_cleanup.py` script has been deprecated. Use the *`clearsessions`* management command instead.

`depth` keyword argument in `select_related` The `depth` keyword argument in *`select_related()`* has been deprecated. You should use field names instead.

9.1.7 1.4 release

Django 1.4.22 release notes

August 18, 2015

Django 1.4.22 fixes a security issue in 1.4.21.

It also fixes support with pip 7+ by disabling wheel support. Older versions of 1.4 would silently build a broken wheel when installed with those versions of pip.

Denial-of-service possibility in `logout()` view by filling session store

Previously, a session could be created when anonymously accessing the *`django.contrib.auth.views.logout()`* view (provided it wasn't decorated with *`login_required()`* as done in the admin). This could allow an attacker to easily create many new session records by sending repeated requests, potentially filling up the session store or causing other users' session records to be evicted.

The *`SessionMiddleware`* has been modified to no longer create empty session records, including when *`SESSION_SAVE_EVERY_REQUEST`* is active.

Additionally, the `contrib.sessions.backends.base.SessionBase.flush()` and `cache_db.SessionStore.flush()` methods have been modified to avoid creating a new empty session. Maintainers of third-party session backends should check if the same vulnerability is present in their backend and correct it if so.

Django 1.4.21 release notes

July 8, 2015

Django 1.4.21 fixes several security issues in 1.4.20.

Denial-of-service possibility by filling session store

In previous versions of Django, the session backends created a new empty record in the session storage anytime `request.session` was accessed and there was a session key provided in the request cookies that didn't already have a session record. This could allow an attacker to easily create many new session records simply by sending repeated requests with unknown session keys, potentially filling up the session store or causing other users' session records to be evicted.

The built-in session backends now create a session record only if the session is actually modified; empty session records are not created. Thus this potential DoS is now only possible if the site chooses to expose a session-modifying view to anonymous users.

As each built-in session backend was fixed separately (rather than a fix in the core sessions framework), maintainers of third-party session backends should check whether the same vulnerability is present in their backend and correct it if so.

Header injection possibility since validators accept newlines in input

Some of Django's built-in validators (*`EmailValidator`*, most seriously) didn't prohibit newline characters (due to the usage of `$` instead of `\Z` in the regular expressions). If you use values with newlines in HTTP response or email headers, you can suffer from header injection attacks. Django itself isn't vulnerable because *`HttpResponse`* and the mail sending utilities in *`django.core.mail`* prohibit newlines in HTTP and SMTP headers, respectively. While

the validators have been fixed in Django, if you're creating HTTP responses or email messages in other ways, it's a good idea to ensure that those methods prohibit newlines as well. You might also want to validate that any existing data in your application doesn't contain unexpected newlines.

validate_ipv4_address(), *validate_slug()*, and *URLValidator* and their usage in the corresponding form fields `GenericIPAddresseField`, `IPAddressField`, `SlugField`, and `URLField` are also affected.

The undocumented, internally unused `validate_integer()` function is now stricter as it validates using a regular expression instead of simply casting the value using `int()` and checking if an exception was raised.

Django 1.4.20 release notes

March 18, 2015

Django 1.4.20 fixes one security issue in 1.4.19.

Mitigated possible XSS attack via user-supplied redirect URLs

Django relies on user input in some cases (e.g. *django.contrib.auth.views.login()* and i18n) to redirect the user to an "on success" URL. The security checks for these redirects (namely `django.utils.http.is_safe_url()`) accepted URLs with leading control characters and so considered URLs like `\x08javascript:...` safe. This issue doesn't affect Django currently, since we only put this URL into the `Location` response header and browsers seem to ignore JavaScript there. Browsers we tested also treat URLs prefixed with control characters such as `%08//example.com` as relative paths so redirection to an unsafe target isn't a problem either.

However, if a developer relies on `is_safe_url()` to provide safe redirect targets and puts such a URL into a link, they could suffer from an XSS attack as some browsers such as Google Chrome ignore control characters at the start of a URL in an anchor `href`.

Django 1.4.19 release notes

January 27, 2015

Django 1.4.19 fixes a regression in the 1.4.18 security release.

Bugfixes

- `GZipMiddleware` now supports streaming responses. As part of the 1.4.18 security release, the `django.views.static.serve()` function was altered to stream the files it serves. Unfortunately, the `GZipMiddleware` consumed the stream prematurely and prevented files from being served properly (#24158).

Django 1.4.18 release notes

January 13, 2015

Django 1.4.18 fixes several security issues in 1.4.17 as well as a regression on Python 2.5 in the 1.4.17 release.

WSGI header spoofing via underscore/dash conflation

When HTTP headers are placed into the WSGI environ, they are normalized by converting to uppercase, converting all dashes to underscores, and prepending *HTTP_*. For instance, a header `X-Auth-User` would become `HTTP_X_AUTH_USER` in the WSGI environ (and thus also in Django's `request.META` dictionary).

Unfortunately, this means that the WSGI environ cannot distinguish between headers containing dashes and headers containing underscores: `X-Auth-User` and `X-Auth_User` both become `HTTP_X_AUTH_USER`. This means that if a header is used in a security-sensitive way (for instance, passing authentication information along from a front-end proxy), even if the proxy carefully strips any incoming value for `X-Auth-User`, an attacker may be able to provide an `X-Auth_User` header (with underscore) and bypass this protection.

In order to prevent such attacks, both Nginx and Apache 2.4+ strip all headers containing underscores from incoming requests by default. Django's built-in development server now does the same. Django's development server is not recommended for production use, but matching the behavior of common production servers reduces the surface area for behavior changes during deployment.

Mitigated possible XSS attack via user-supplied redirect URLs

Django relies on user input in some cases (e.g. *django.contrib.auth.views.login()* and i18n) to redirect the user to an "on success" URL. The security checks for these redirects (namely `django.utils.http.is_safe_url()`) didn't strip leading whitespace on the tested URL and as such considered URLs like `\njavascript:...` safe. If a developer relied on `is_safe_url()` to provide safe redirect targets and put such a URL into a link, they could suffer from a XSS attack. This bug doesn't affect Django currently, since we only put this URL into the `Location` response header and browsers seem to ignore JavaScript there.

Denial-of-service attack against `django.views.static.serve`

In older versions of Django, the *django.views.static.serve()* view read the files it served one line at a time. Therefore, a big file with no newlines would result in memory usage equal to the size of that file. An attacker could exploit this and launch a denial-of-service attack by simultaneously requesting many large files. This view now reads the file in chunks to prevent large memory usage.

Note, however, that this view has always carried a warning that it is not hardened for production use and should be used only as a development aid. Now may be a good time to audit your project and serve your files in production using a real front-end web server if you are not doing so.

Bugfixes

- To maintain compatibility with Python 2.5, Django's vendored version of six, *django.utils.six*, has been downgraded to 1.8.0 which is the last version to support Python 2.5.

Django 1.4.17 release notes

January 2, 2015

Django 1.4.17 fixes a regression in the 1.4.14 security release.

Additionally, Django's vendored version of six, *django.utils.six*, has been upgraded to the latest release (1.9.0).

Bugfixes

- Fixed a regression with dynamically generated inlines and allowed field references in the admin (#23754).

Django 1.4.16 release notes

October 22, 2014

Django 1.4.16 fixes a couple regressions in the 1.4.14 security release and a bug preventing the use of some GEOS versions with GeoDjango.

Bugfixes

- Allowed related many-to-many fields to be referenced in the admin (#23604).
- Allowed inline and hidden references to admin fields (#23431).
- Fixed parsing of the GEOS version string (#20036).

Django 1.4.15 release notes

September 2, 2014

Django 1.4.15 fixes a regression in the 1.4.14 security release.

Bugfixes

- Allowed inherited and m2m fields to be referenced in the admin (#22486)

Django 1.4.14 release notes

August 20, 2014

Django 1.4.14 fixes several security issues in 1.4.13.

`reverse()` could generate URLs pointing to other hosts

In certain situations, URL reversing could generate scheme-relative URLs (URLs starting with two slashes), which could unexpectedly redirect a user to a different host. An attacker could exploit this, for example, by redirecting users to a phishing site designed to ask for user's passwords.

To remedy this, URL reversing now ensures that no URL starts with two slashes (//), replacing the second slash with its URL encoded counterpart (%2F). This approach ensures that semantics stay the same, while making the URL relative to the domain and not to the scheme.

File upload denial-of-service

Before this release, Django's file upload handing in its default configuration may degrade to producing a huge number of `os.stat()` system calls when a duplicate filename is uploaded. Since `stat()` may invoke IO, this may produce a huge data-dependent slowdown that slowly worsens over time. The net result is that given enough time, a user with the ability to upload files can cause poor performance in the upload handler, eventually causing it to become very slow

simply by uploading 0-byte files. At this point, even a slow network connection and few HTTP requests would be all that is necessary to make a site unavailable.

We've remedied the issue by changing the algorithm for generating file names if a file with the uploaded name already exists. *Storage.get_available_name()* now appends an underscore plus a random 7 character alphanumeric string (e.g. "_x3a1gho"), rather than iterating through an underscore followed by a number (e.g. "_1", "_2", etc.).

RemoteUserMiddleware session hijacking

When using the *RemoteUserMiddleware* and the RemoteUserBackend, a change to the REMOTE_USER header between requests without an intervening logout could result in the prior user's session being co-opted by the subsequent user. The middleware now logs the user out on a failed login attempt.

Data leakage via query string manipulation in contrib.admin

In older versions of Django it was possible to reveal any field's data by modifying the "popup" and "to_field" parameters of the query string on an admin change form page. For example, requesting a URL like /admin/auth/user/?pop=1&t=password and viewing the page's HTML allowed viewing the password hash of each user. While the admin requires users to have permissions to view the change form pages in the first place, this could leak data if you rely on users having access to view only certain fields on a model.

To address the issue, an exception will now be raised if a to_field value that isn't a related field to a model that has been registered with the admin is specified.

Django 1.4.13 release notes

May 14, 2014

Django 1.4.13 fixes two security issues in 1.4.12.

Caches may incorrectly be allowed to store and serve private data

In certain situations, Django may allow caches to store private data related to a particular session and then serve that data to requests with a different session, or no session at all. This can lead to information disclosure and can be a vector for cache poisoning.

When using Django sessions, Django will set a Vary: Cookie header to ensure caches do not serve cached data to requests from other sessions. However, older versions of Internet Explorer (most likely only Internet Explorer 6, and Internet Explorer 7 if run on Windows XP or Windows Server 2003) are unable to handle the Vary header in combination with many content types. Therefore, Django would remove the header if the request was made by Internet Explorer.

To remedy this, the special behavior for these older Internet Explorer versions has been removed, and the Vary header is no longer stripped from the response. In addition, modifications to the Cache-Control header for all Internet Explorer requests with a Content-Disposition header have also been removed as they were found to have similar issues.

Malformed redirect URLs from user input not correctly validated

The validation for redirects did not correctly validate some malformed URLs, which are accepted by some browsers. This allows a user to be redirected to an unsafe URL unexpectedly.

Django relies on user input in some cases (e.g. *django.contrib.auth.views.login()*, `django.contrib.comments`, and i18n) to redirect the user to an "on success" URL. The security checks for these redirects (namely `django.utils.http.is_safe_url()`) did not correctly validate some malformed URLs, such as *http:\\\djangoproject.com*, which are accepted by some browsers with more liberal URL parsing.

To remedy this, the validation in `is_safe_url()` has been tightened to be able to handle and correctly validate these malformed URLs.

Django 1.4.12 release notes

April 28, 2014

Django 1.4.12 fixes a regression in the 1.4.11 security release.

Bugfixes

- Restored the ability to `reverse()` views created using `functools.partial()` (#22486).

Django 1.4.11 release notes

April 21, 2014

Django 1.4.11 fixes three security issues in 1.4.10. Additionally, Django's vendored version of six, *django.utils.six*, has been upgraded to the latest release (1.6.1).

Unexpected code execution using `reverse()`

Django's URL handling is based on a mapping of regex patterns (representing the URLs) to callable views, and Django's own processing consists of matching a requested URL against those patterns to determine the appropriate view to invoke.

Django also provides a convenience function – `reverse()` – which performs this process in the opposite direction. The `reverse()` function takes information about a view and returns a URL which would invoke that view. Use of `reverse()` is encouraged for application developers, as the output of `reverse()` is always based on the current URL patterns, meaning developers do not need to change other code when making changes to URLs.

One argument signature for `reverse()` is to pass a dotted Python path to the desired view. In this situation, Django will import the module indicated by that dotted path as part of generating the resulting URL. If such a module has import-time side effects, those side effects will occur.

Thus it is possible for an attacker to cause unexpected code execution, given the following conditions:

1. One or more views are present which construct a URL based on user input (commonly, a "next" parameter in a querystring indicating where to redirect upon successful completion of an action).
2. One or more modules are known to an attacker to exist on the server's Python import path, which perform code execution with side effects on importing.

To remedy this, `reverse()` will now only accept and import dotted paths based on the view-containing modules listed in the project's URL pattern configuration, so as to ensure that only modules the developer intended to be imported in this fashion can or will be imported.

Caching of anonymous pages could reveal CSRF token

Django includes both a caching framework and a system for preventing cross-site request forgery (CSRF) attacks. The CSRF-protection system is based on a random nonce sent to the client in a cookie which must be sent by the client on future requests and, in forms, a hidden value which must be submitted back with the form.

The caching framework includes an option to cache responses to anonymous (i.e., unauthenticated) clients.

When the first anonymous request to a given page is by a client which did not have a CSRF cookie, the cache framework will also cache the CSRF cookie and serve the same nonce to other anonymous clients who do not have a CSRF cookie. This can allow an attacker to obtain a valid CSRF cookie value and perform attacks which bypass the check for the cookie.

To remedy this, the caching framework will no longer cache such responses. The heuristic for this will be:

1. If the incoming request did not submit any cookies, and
2. If the response did send one or more cookies, and
3. If the `Vary: Cookie` header is set on the response, then the response will not be cached.

MySQL typecasting

The MySQL database is known to "typecast" on certain queries; for example, when querying a table which contains string values, but using a query which filters based on an integer value, MySQL will first silently coerce the strings to integers and return a result based on that.

If a query is performed without first converting values to the appropriate type, this can produce unexpected results, similar to what would occur if the query itself had been manipulated.

Django's model field classes are aware of their own types and most such classes perform explicit conversion of query arguments to the correct database-level type before querying. However, three model field classes did not correctly convert their arguments:

- *`FilePathField`*
- *`GenericIPAddressField`*
- `IPAddressField`

These three fields have been updated to convert their arguments to the correct types before querying.

Additionally, developers of custom model fields are now warned via documentation to ensure their custom field classes will perform appropriate type conversions, and users of the *`raw()`* and *`extra()`* query methods – which allow the developer to supply raw SQL or SQL fragments – will be advised to ensure they perform appropriate manual type conversions prior to executing queries.

Django 1.4.10 release notes

November 6, 2013

Django 1.4.10 fixes a Python-compatibility bug in the 1.4 series.

Python compatibility

Django 1.4.9 inadvertently introduced issues with Python 2.5 compatibility. Django 1.4.10 restores Python 2.5 compatibility. This was issue #21362 in Django's Trac.

Django 1.4.9 release notes

October 23, 2013

Django 1.4.9 fixes a security-related bug in the 1.4 series and one other data corruption bug.

Readdressed denial-of-service via password hashers

Django 1.4.8 imposes a 4096-byte limit on passwords in order to mitigate a denial-of-service attack through submission of bogus but extremely large passwords. In Django 1.4.9, we've reverted this change and instead improved the speed of our PBKDF2 algorithm by not rehashing the key on every iteration.

Bugfixes

- Fixed a data corruption bug with `datetime_safe.datetime.combine` (#21256).

Django 1.4.8 release notes

September 14, 2013

Django 1.4.8 fixes two security issues present in previous Django releases in the 1.4 series.

Denial-of-service via password hashers

In previous versions of Django, no limit was imposed on the plaintext length of a password. This allowed a denial-of-service attack through submission of bogus but extremely large passwords, tying up server resources performing the (expensive, and increasingly expensive with the length of the password) calculation of the corresponding hash.

As of 1.4.8, Django's authentication framework imposes a 4096-byte limit on passwords and will fail authentication with any submitted password of greater length.

Corrected usage of `sensitive_post_parameters()` in `django.contrib.auth`'s admin

The decoration of the `add_view` and `user_change_password` user admin views with *`sensitive_post_parameters()`* did not include *`method_decorator()`* (required since the views are methods) resulting in the decorator not being properly applied. This usage has been fixed and *`sensitive_post_parameters()`* will now throw an exception if it's improperly used.

Django 1.4.7 release notes

September 10, 2013

Django 1.4.7 fixes one security issue present in previous Django releases in the 1.4 series.

Directory traversal vulnerability in `ssi` template tag

In previous versions of Django it was possible to bypass the `ALLOWED_INCLUDE_ROOTS` setting used for security with the `ssi` template tag by specifying a relative path that starts with one of the allowed roots. For example, if `ALLOWED_INCLUDE_ROOTS = ("/var/www",)` the following would be possible:

```
{% ssi "/var/www/../../etc/passwd" %}
```

In practice this is not a very common problem, as it would require the template author to put the `ssi` file in a user-controlled variable, but it's possible in principle.

Django 1.4.6 release notes

August 13, 2013

Django 1.4.6 fixes one security issue present in previous Django releases in the 1.4 series, as well as one other bug.

This is the sixth bugfix/security release in the Django 1.4 series.

Mitigated possible XSS attack via user-supplied redirect URLs

Django relies on user input in some cases (e.g. *django.contrib.auth.views.login()*, `django.contrib.comments`, and i18n) to redirect the user to an "on success" URL. The security checks for these redirects (namely `django.utils.http.is_safe_url()`) didn't check if the scheme is `http(s)` and as such allowed `javascript:...` URLs to be entered. If a developer relied on `is_safe_url()` to provide safe redirect targets and put such a URL into a link, they could suffer from a XSS attack. This bug doesn't affect Django currently, since we only put this URL into the `Location` response header and browsers seem to ignore JavaScript there.

Bugfixes

- Fixed an obscure bug with the *override_settings()* decorator. If you hit an `AttributeError: 'Settings' object has no attribute '_original_allowed_hosts'` exception, it's probably fixed (#20636).

Django 1.4.5 release notes

February 20, 2013

Django 1.4.5 corrects a packaging problem with yesterday's 1.4.4 release.

The release contained stray `.pyc` files that caused "bad magic number" errors when running with some versions of Python. This releases corrects this, and also fixes a bad documentation link in the project template `settings.py` file generated by `manage.py startproject`.

Django 1.4.4 release notes

February 19, 2013

Django 1.4.4 fixes four security issues present in previous Django releases in the 1.4 series, as well as several other bugs and numerous documentation improvements.

This is the fourth bugfix/security release in the Django 1.4 series.

Host header poisoning

Some parts of Django – independent of end-user-written applications – make use of full URLs, including domain name, which are generated from the HTTP Host header. Django's documentation has for some time contained notes advising users on how to configure Web servers to ensure that only valid Host headers can reach the Django application. However, it has been reported to us that even with the recommended Web server configurations there are still techniques available for tricking many common Web servers into supplying the application with an incorrect and possibly malicious Host header.

For this reason, Django 1.4.4 adds a new setting, `ALLOWED_HOSTS`, containing an explicit list of valid host/domain names for this site. A request with a Host header not matching an entry in this list will raise `SuspiciousOperation` if `request.get_host()` is called. For full details see the documentation for the *`ALLOWED_HOSTS`* setting.

The default value for this setting in Django 1.4.4 is `['*']` (matching any host), for backwards-compatibility, but we strongly encourage all sites to set a more restrictive value.

This host validation is disabled when `DEBUG` is `True` or when running tests.

XML deserialization

The XML parser in the Python standard library is vulnerable to a number of attacks via external entities and entity expansion. Django uses this parser for deserializing XML-formatted database fixtures. This deserializer is not intended for use with untrusted data, but in order to err on the side of safety in Django 1.4.4 the XML deserializer refuses to parse an XML document with a DTD (DOCTYPE definition), which closes off these attack avenues.

These issues in the Python standard library are CVE-2013-1664 and CVE-2013-1665. More information available from the Python security team.

Django's XML serializer does not create documents with a DTD, so this should not cause any issues with the typical round-trip from `dumpdata` to `loaddata`, but if you feed your own XML documents to the `loaddata` management command, you will need to ensure they do not contain a DTD.

Formset memory exhaustion

Previous versions of Django did not validate or limit the form-count data provided by the client in a formset's management form, making it possible to exhaust a server's available memory by forcing it to create very large numbers of forms.

In Django 1.4.4, all formsets have a strictly-enforced maximum number of forms (1000 by default, though it can be set higher via the `max_num` formset factory argument).

Admin history view information leakage

In previous versions of Django, an admin user without change permission on a model could still view the unicode representation of instances via their admin history log. Django 1.4.4 now limits the admin history log view for an object to users with change permission for that model.

Other bugfixes and changes

- Prevented transaction state from leaking from one request to the next (#19707).
- Changed an SQL command syntax to be MySQL 4 compatible (#19702).

- Added backwards-compatibility with old unsalted MD5 passwords (#18144).
- Numerous documentation improvements and fixes.

Django 1.4.3 release notes

December 10, 2012

Django 1.4.3 addresses two security issues present in previous Django releases in the 1.4 series.

Please be aware that this security release is slightly different from previous ones. Both issues addressed here have been dealt with in prior security updates to Django. In one case, we have received ongoing reports of problems, and in the other we've chosen to take further steps to tighten up Django's code in response to independent discovery of potential problems from multiple sources.

Host header poisoning

Several earlier Django security releases focused on the issue of poisoning the HTTP Host header, causing Django to generate URLs pointing to arbitrary, potentially-malicious domains.

In response to further input received and reports of continuing issues following the previous release, we're taking additional steps to tighten Host header validation. Rather than attempt to accommodate all features HTTP supports here, Django's Host header validation attempts to support a smaller, but far more common, subset:

- Hostnames must consist of characters `[A-Za-z0-9]` plus hyphen ('-') or dot ('.').
- IP addresses – both IPv4 and IPv6 – are permitted.
- Port, if specified, is numeric.

Any deviation from this will now be rejected, raising the exception *`django.core.exceptions.SuspiciousOperation`*.

Redirect poisoning

Also following up on a previous issue: in July of this year, we made changes to Django's HTTP redirect classes, performing additional validation of the scheme of the URL to redirect to (since, both within Django's own supplied applications and many third-party applications, accepting a user-supplied redirect target is a common pattern).

Since then, two independent audits of the code turned up further potential problems. So, similar to the Host-header issue, we are taking steps to provide tighter validation in response to reported problems (primarily with third-party applications, but to a certain extent also within Django itself). This comes in two parts:

1. A new utility function, `django.utils.http.is_safe_url`, is added; this function takes a URL and a hostname, and checks that the URL is either relative, or if absolute matches the supplied hostname. This function is intended for use whenever user-supplied redirect targets are accepted, to ensure that such redirects cannot lead to arbitrary third-party sites.

2. All of Django's own built-in views – primarily in the authentication system – which allow user-supplied redirect targets now use `is_safe_url` to validate the supplied URL.

Django 1.4.2 release notes

October 17, 2012

This is the second security release in the Django 1.4 series.

Host header poisoning

Some parts of Django – independent of end-user-written applications – make use of full URLs, including domain name, which are generated from the HTTP Host header. Some attacks against this are beyond Django's ability to control, and require the web server to be properly configured; Django's documentation has for some time contained notes advising users on such configuration.

Django's own built-in parsing of the Host header is, however, still vulnerable, as was reported to us recently. The Host header parsing in Django 1.3.3 and Django 1.4.1 – specifically, `django.http.HttpRequest.get_host()` – was incorrectly handling username/password information in the header. Thus, for example, the following Host header would be accepted by Django when running on "validsite.com":

```
Host: validsite.com:random@evilsite.com
```

Using this, an attacker can cause parts of Django – particularly the password-reset mechanism – to generate and display arbitrary URLs to users.

To remedy this, the parsing in `HttpRequest.get_host()` is being modified; Host headers which contain potentially dangerous content (such as username/password pairs) now raise the exception *`django.core.exceptions.SuspiciousOperation`*.

Details of this issue were initially posted online as a security advisory.

Backwards incompatible changes

- The newly introduced *`GenericIPAddressField`* constructor arguments have been adapted to match those of all other model fields. The first two keyword arguments are now verbose_name and name.

Other bugfixes and changes

- Subclass HTMLParser only for appropriate Python versions (#18239).
- Added batch_size argument to qs.bulk_create() (#17788).
- Fixed a small regression in the admin filters where wrongly formatted dates passed as url parameters caused an unhandled ValidationError (#18530).
- Fixed an endless loop bug when accessing permissions in templates (#18979)
- Fixed some Python 2.5 compatibility issues
- Fixed an issue with quoted filenames in Content-Disposition header (#19006)
- Made the context option in `trans` and `blocktrans` tags accept literals wrapped in single quotes (#18881).
- Numerous documentation improvements and fixes.

Django 1.4.1 release notes

July 30, 2012

This is the first security release in the Django 1.4 series, fixing several security issues in Django 1.4. Django 1.4.1 is a recommended upgrade for all users of Django 1.4.

For a full list of issues addressed in this release, see the security advisory.

Django 1.4 release notes

March 23, 2012

Welcome to Django 1.4!

These release notes cover the *new features*, as well as some *backwards incompatible changes* you'll want to be aware of when upgrading from Django 1.3 or older versions. We've also dropped some features, which are detailed in *our deprecation plan*, and we've *begun the deprecation process for some features*.

Overview

The biggest new feature in Django 1.4 is *support for time zones* when handling date/times. When enabled, this Django will store date/times in UTC, use timezone-aware objects internally, and translate them to users' local timezones for display.

If you're upgrading an existing project to Django 1.4, switching to the timezone aware mode may take some care: the new mode disallows some rather sloppy behavior that used to be accepted. We encourage anyone who's upgrading to check out the *timezone migration guide* and the *timezone FAQ* for useful pointers.

Other notable new features in Django 1.4 include:

- A number of ORM improvements, including *SELECT FOR UPDATE support*, the ability to *bulk insert* large datasets for improved performance, and *QuerySet.prefetch_related*, a method to batch-load related objects in areas where `select_related()` doesn't work.
- Some nice security additions, including *improved password hashing* (featuring PBKDF2 and bcrypt support), new *tools for cryptographic signing*, several *CSRF improvements*, and *simple clickjacking protection*.
- An *updated default project layout and manage.py* that removes the "magic" from prior versions. And for those who don't like the new layout, you can use *custom project and app templates* instead!
- *Support for in-browser testing frameworks* (like Selenium).
- ... and a whole lot more; *see below*!

Wherever possible we try to introduce new features in a backwards-compatible manner per our API stability policy policy. However, as with previous releases, Django 1.4 ships with some minor *backwards incompatible changes*; people upgrading from previous versions of Django should read that list carefully.

Python compatibility

Django 1.4 has dropped support for Python 2.4. Python 2.5 is now the minimum required Python version. Django is tested and supported on Python 2.5, 2.6 and 2.7.

This change should affect only a small number of Django users, as most operating-system vendors today are shipping Python 2.5 or newer as their default version. If you're still using Python 2.4, however, you'll need to stick to Django 1.3 until you can upgrade. Per our support policy, Django 1.3 will continue to receive security support until the release of Django 1.5.

Django does not support Python 3.x at this time. At some point before the release of Django 1.4, we plan to publish a document outlining our full timeline for deprecating Python 2.x and moving to Python 3.x.

What's new in Django 1.4

Support for time zones In previous versions, Django used "naive" date/times (that is, date/times without an associated time zone), leaving it up to each developer to interpret what a given date/time "really means". This can cause all sorts of subtle timezone-related bugs.

In Django 1.4, you can now switch Django into a more correct, time-zone aware mode. In this mode, Django stores date and time information in UTC in the database, uses time-zone-aware datetime objects internally and translates them to the end user's time zone in templates and forms. Reasons for using this feature include:

- Customizing date and time display for users around the world.
- Storing datetimes in UTC for database portability and interoperability. (This argument doesn't apply to PostgreSQL, because it already stores timestamps with time zone information in Django 1.3.)
- Avoiding data corruption problems around DST transitions.

Time zone support is enabled by default in new projects created with *`startproject`*. If you want to use this feature in an existing project, read the *migration guide*. If you encounter problems, there's a helpful *FAQ*.

Support for in-browser testing frameworks Django 1.4 supports integration with in-browser testing frameworks like Selenium. The new *`django.test.LiveServerTestCase`* base class lets you test the interactions between your site's front and back ends more comprehensively. See the *`documentation`* for more details and concrete examples.

Updated default project layout and `manage.py` Django 1.4 ships with an updated default project layout and `manage.py` file for the *`startproject`* management command. These fix some issues with the previous `manage.py` handling of Python import paths that caused double imports, trouble moving from development to deployment, and other difficult-to-debug path issues.

The previous `manage.py` called functions that are now deprecated, and thus projects upgrading to Django 1.4 should update their `manage.py`. (The old-style `manage.py` will continue to work as before until Django 1.6. In 1.5 it will raise `DeprecationWarning`).

The new recommended `manage.py` file should look like this:

```
#!/usr/bin/env python
import os, sys

if __name__ == "__main__":
    os.environ.setdefault("DJANGO_SETTINGS_MODULE", "{{ project_name }}.settings")

    from django.core.management import execute_from_command_line

    execute_from_command_line(sys.argv)
```

`{{ project_name }}` should be replaced with the Python package name of the actual project.

If settings, URLconfs and apps within the project are imported or referenced using the project name prefix (e.g. `myproject.settings`, `ROOT_URLCONF = "myproject.urls"`, etc.), the new `manage.py` will need to be moved one directory up, so it is outside the project package rather than adjacent to `settings.py` and `urls.py`.

For instance, with the following layout:

```
manage.py
mysite/
    __init__.py
    settings.py
    urls.py
    myapp/
```

```
        __init__.py
        models.py
```

You could import `mysite.settings`, `mysite.urls`, and `mysite.myapp`, but not `settings`, `urls`, or `myapp` as top-level modules.

Anything imported as a top-level module can be placed adjacent to the new `manage.py`. For instance, to decouple "myapp" from the project module and import it as just `myapp`, place it outside the `mysite/` directory:

```
manage.py
myapp/
    __init__.py
    models.py
mysite/
    __init__.py
    settings.py
    urls.py
```

If the same code is imported inconsistently (some places with the project prefix, some places without it), the imports will need to be cleaned up when switching to the new `manage.py`.

Custom project and app templates The *`startapp`* and *`startproject`* management commands now have a `--template` option for specifying a path or URL to a custom app or project template.

For example, Django will use the `/path/to/my_project_template` directory when you run the following command:

```
django-admin.py startproject --template=/path/to/my_project_template myproject
```

You can also now provide a destination directory as the second argument to both *`startapp`* and *`startproject`*:

```
django-admin.py startapp myapp /path/to/new/app
django-admin.py startproject myproject /path/to/new/project
```

For more information, see the *`startapp`* and *`startproject`* documentation.

Improved WSGI support The *`startproject`* management command now adds a `wsgi.py` module to the initial project layout, containing a simple WSGI application that can be used for deploying with WSGI app servers.

The *`built-in development server`* now supports using an externally-defined WSGI callable, which makes it possible to run runserver with the same WSGI configuration that is used for deployment. The new *`WSGI_APPLICATION`* setting lets you configure which WSGI callable *`runserver`* uses.

(The `runfcgi` management command also internally wraps the WSGI callable configured via *`WSGI_APPLICATION`*.)

`SELECT FOR UPDATE` support Django 1.4 includes a *`QuerySet.select_for_update()`* method, which generates a `SELECT ... FOR UPDATE` SQL query. This will lock rows until the end of the transaction, meaning other transactions cannot modify or delete rows matched by a `FOR UPDATE` query.

For more details, see the documentation for *`select_for_update()`*.

`Model.objects.bulk_create` in the ORM This method lets you create multiple objects more efficiently. It can result in significant performance increases if you have many objects.

Django makes use of this internally, meaning some operations (such as database setup for test suites) have seen a performance benefit as a result.

See the *bulk_create()* docs for more information.

QuerySet.prefetch_related Similar to *select_related()* but with a different strategy and broader scope, *prefetch_related()* has been added to *QuerySet*. This method returns a new QuerySet that will prefetch each of the specified related lookups in a single batch as soon as the query begins to be evaluated. Unlike select_related, it does the joins in Python, not in the database, and supports many-to-many relationships, GenericForeignKey and more. This allows you to fix a very common performance problem in which your code ends up doing O(n) database queries (or worse) if objects on your primary QuerySet each have many related objects that you also need to fetch.

Improved password hashing Django's auth system (django.contrib.auth) stores passwords using a one-way algorithm. Django 1.3 uses the SHA1 algorithm, but increasing processor speeds and theoretical attacks have revealed that SHA1 isn't as secure as we'd like. Thus, Django 1.4 introduces a new password storage system: by default Django now uses the PBKDF2 algorithm (as recommended by NIST). You can also easily choose a different algorithm (including the popular bcrypt algorithm). For more details, see *How Django stores passwords*.

HTML5 doctype We've switched the admin and other bundled templates to use the HTML5 doctype. While Django will be careful to maintain compatibility with older browsers, this change means that you can use any HTML5 features you need in admin pages without having to lose HTML validity or override the provided templates to change the doctype.

List filters in admin interface Prior to Django 1.4, the *admin* app let you specify change list filters by specifying a field lookup, but it didn't allow you to create custom filters. This has been rectified with a simple API (previously used internally and known as "FilterSpec"). For more details, see the documentation for *list_filter*.

Multiple sort in admin interface The admin change list now supports sorting on multiple columns. It respects all elements of the *ordering* attribute, and sorting on multiple columns by clicking on headers is designed to mimic the behavior of desktop GUIs. We also added a *get_ordering()* method for specifying the ordering dynamically (i.e., depending on the request).

New ModelAdmin methods We added a *save_related()* method to *ModelAdmin* to ease customization of how related objects are saved in the admin.

Two other new *ModelAdmin* methods, *get_list_display()* and *get_list_display_links()* enable dynamic customization of fields and links displayed on the admin change list.

Admin inlines respect user permissions Admin inlines now only allow those actions for which the user has permission. For ManyToMany relationships with an auto-created intermediate model (which does not have its own permissions), the change permission for the related model determines if the user has the permission to add, change or delete relationships.

Tools for cryptographic signing Django 1.4 adds both a low-level API for signing values and a high-level API for setting and reading signed cookies, one of the most common uses of signing in Web applications.

See the cryptographic signing docs for more information.

Cookie-based session backend Django 1.4 introduces a cookie-based session backend that uses the tools for cryptographic signing to store the session data in the client's browser.

: Session data is signed and validated by the server, but it's not encrypted. This means a user can view any data stored in the session but cannot change it. Please read the documentation for further clarification before using this backend.

See the *cookie-based session backend* docs for more information.

New form wizard The previous `FormWizard` from `django.contrib.formtools` has been replaced with a new implementation based on the class-based views introduced in Django 1.3. It features a pluggable storage API and doesn't require the wizard to pass around hidden fields for every previous step.

Django 1.4 ships with a session-based storage backend and a cookie-based storage backend. The latter uses the tools for cryptographic signing also introduced in Django 1.4 to store the wizard's state in the user's cookies.

`reverse_lazy` A lazily evaluated version of `reverse()` was added to allow using URL reversals before the project's URLconf gets loaded.

Translating URL patterns Django can now look for a language prefix in the URLpattern when using the new *`i18n_patterns()`* helper function. It's also now possible to define translatable URL patterns using *`ugettext_lazy()`*. See *Internationalization: in URL patterns* for more information about the language prefix and how to internationalize URL patterns.

Contextual translation support for `{% trans %}` and `{% blocktrans %}` The *contextual translation* support introduced in Django 1.3 via the `pgettext` function has been extended to the *`trans`* and *`blocktrans`* template tags using the new `context` keyword.

Customizable `SingleObjectMixin` URLConf kwargs Two new attributes, *`pk_url_kwarg`* and *`slug_url_kwarg`*, have been added to *`SingleObjectMixin`* to enable the customization of URLconf keyword arguments used for single object generic views.

Assignment template tags A new `assignment_tag` helper function was added to `template.Library` to ease the creation of template tags that store data in a specified context variable.

`*args` and `**kwargs` support for template tag helper functions The *simple_tag*, *inclusion_tag* and newly introduced `assignment_tag` template helper functions may now accept any number of positional or keyword arguments. For example:

```
@register.simple_tag
def my_tag(a, b, *args, **kwargs):
    warning = kwargs['warning']
    profile = kwargs['profile']
    ...
    return ...
```

Then, in the template, any number of arguments may be passed to the template tag. For example:

```
{% my_tag 123 "abcd" book.title warning=message|lower profile=user.profile %}
```

No wrapping of exceptions in `TEMPLATE_DEBUG` mode In previous versions of Django, whenever the `TEMPLATE_DEBUG` setting was `True`, any exception raised during template rendering (even exceptions unrelated to template syntax) were wrapped in `TemplateSyntaxError` and re-raised. This was done in order to provide detailed template source location information in the debug 500 page.

In Django 1.4, exceptions are no longer wrapped. Instead, the original exception is annotated with the source information. This means that catching exceptions from template rendering is now consistent regardless of the value of `TEMPLATE_DEBUG`, and there's no need to catch and unwrap `TemplateSyntaxError` in order to catch other errors.

`truncatechars` template filter This new filter truncates a string to be no longer than the specified number of characters. Truncated strings end with a translatable ellipsis sequence ("..."). See the documentation for *`truncatechars`* for more details.

`static` template tag The *`staticfiles`* contrib app has a new `static` template tag to refer to files saved with the *`STATICFILES_STORAGE`* storage backend. It uses the storage backend's `url` method and therefore supports advanced features such as *serving files from a cloud service*.

`CachedStaticFilesStorage` storage backend The *`staticfiles`* contrib app now has a *`CachedStaticFilesStorage`* backend that caches the files it saves (when running the *`collectstatic`* management command) by appending the MD5 hash of the file's content to the filename. For example, the file `css/styles.css` would also be saved as `css/styles.55e7cbb9ba48.css`

See the *`CachedStaticFilesStorage`* docs for more information.

Simple clickjacking protection We've added a middleware to provide easy protection against clickjacking using the `X-Frame-Options` header. It's not enabled by default for backwards compatibility reasons, but you'll almost certainly want to enable it to help plug that security hole for browsers that support the header.

CSRF improvements We've made various improvements to our CSRF features, including the *`ensure_csrf_cookie()`* decorator, which can help with AJAX-heavy sites; protection for PUT and DELETE requests; and the *`CSRF_COOKIE_SECURE`* and *`CSRF_COOKIE_PATH`* settings, which can improve the security and usefulness of CSRF protection. See the CSRF docs for more information.

Error report filtering We added two function decorators, *`sensitive_variables()`* and *`sensitive_post_parameters()`*, to allow designating the local variables and POST parameters that may contain sensitive information and should be filtered out of error reports.

All POST parameters are now systematically filtered out of error reports for certain views (`login`, `password_reset_confirm`, `password_change` and `add_view` in *`django.contrib.auth.views`*, as well as `user_change_password` in the admin app) to prevent the leaking of sensitive information such as user passwords.

You can override or customize the default filtering by writing a *custom filter*. For more information see the docs on *Filtering error reports*.

Extended IPv6 support Django 1.4 can now better handle IPv6 addresses with the new *`GenericIPAddressField`* model field, *`GenericIPAddressField`* form field and the validators *`validate_ipv46_address`* and *`validate_ipv6_address`*.

HTML comparisons in tests The base classes in *django.test* now have some helpers to compare HTML without tripping over irrelevant differences in whitespace, argument quoting/ordering and closing of self-closing tags. You can either compare HTML directly with the new *assertHTMLEqual()* and *assertHTMLNotEqual()* assertions, or use the `html=True` flag with *assertContains()* and *assertNotContains()* to test whether the client's response contains a given HTML fragment. See the *assertions documentation* for more.

Two new date format strings Two new *date* formats were added for use in template filters, template tags and Format localization:

- `e` – the name of the timezone of the given datetime object
- `o` – the ISO 8601 year number

Please make sure to update your *custom format files* if they contain either `e` or `o` in a format string. For example a Spanish localization format previously only escaped the `d` format character:

```
DATE_FORMAT = r'j \de F \de Y'
```

But now it needs to also escape `e` and `o`:

```
DATE_FORMAT = r'j \d\e F \d\e Y'
```

For more information, see the *date* documentation.

Minor features Django 1.4 also includes several smaller improvements worth noting:

- A more usable stacktrace in the technical 500 page. Frames in the stack trace that reference Django's framework code are dimmed out, while frames in application code are slightly emphasized. This change makes it easier to scan a stacktrace for issues in application code.
- Tablespace support in PostgreSQL.
- Customizable names for *simple_tag()*.
- In the documentation, a helpful security overview page.
- The `django.contrib.auth.models.check_password` function has been moved to the *django.contrib.auth.hashers* module. Importing it from the old location will still work, but you should update your imports.
- The *collectstatic* management command now has a `--clear` option to delete all files at the destination before copying or linking the static files.
- It's now possible to load fixtures containing forward references when using MySQL with the InnoDB database engine.
- A new 403 response handler has been added as `'django.views.defaults.permission_denied'`. You can set your own handler by setting the value of *django.conf.urls.handler403*. See the documentation about *the 403 (HTTP Forbidden) view* for more information.
- The *makemessages* command uses a new and more accurate lexer, JsLex, for extracting translatable strings from JavaScript files.
- The *trans* template tag now takes an optional `as` argument to be able to retrieve a translation string without displaying it but setting a template context variable instead.
- The *if* template tag now supports `{% elif %}` clauses.
- If your Django app is behind a proxy, you might find the new *SECURE_PROXY_SSL_HEADER* setting useful. It solves the problem of your proxy "eating" the fact that a request came in via HTTPS. But only use this setting if you know what you're doing.

- A new, plain-text, version of the HTTP 500 status code internal error page served when *DEBUG* is `True` is now sent to the client when Django detects that the request has originated in JavaScript code. (*is_ajax()* is used for this.)

  Like its HTML counterpart, it contains a collection of different pieces of information about the state of the application.

  This should make it easier to read when debugging interaction with client-side JavaScript.

- Added the *makemessages --no-location* option.
- Changed the `locmem` cache backend to use `pickle.HIGHEST_PROTOCOL` for better compatibility with the other cache backends.
- Added support in the ORM for generating `SELECT` queries containing `DISTINCT ON`.

  The `distinct()` `QuerySet` method now accepts an optional list of model field names. If specified, then the `DISTINCT` statement is limited to these fields. This is only supported in PostgreSQL.

  For more details, see the documentation for *distinct()*.

- The admin login page will add a password reset link if you include a URL with the name *'admin_password_reset'* in your urls.py, so plugging in the built-in password reset mechanism and making it available is now much easier. For details, see *Adding a password-reset feature*.
- The MySQL database backend can now make use of the savepoint feature implemented by MySQL version 5.0.3 or newer with the InnoDB storage engine.
- It's now possible to pass initial values to the model forms that are part of both model formsets and inline model formsets as returned from factory functions `modelformset_factory` and `inlineformset_factory` respectively just like with regular formsets. However, initial values only apply to extra forms, i.e. those which are not bound to an existing model instance.
- The sitemaps framework can now handle HTTPS links using the new *Sitemap.protocol* class attribute.
- A new *django.test.SimpleTestCase* subclass of *unittest.TestCase* that's lighter than *django.test.TestCase* and company. It can be useful in tests that don't need to hit a database. See *Hierarchy of Django unit testing classes*.

Backwards incompatible changes in 1.4

SECRET_KEY setting is required Running Django with an empty or known *SECRET_KEY* disables many of Django's security protections and can lead to remote-code-execution vulnerabilities. No Django site should ever be run without a *SECRET_KEY*.

In Django 1.4, starting Django with an empty *SECRET_KEY* will raise a *DeprecationWarning*. In Django 1.5, it will raise an exception and Django will refuse to start. This is slightly accelerated from the usual deprecation path due to the severity of the consequences of running Django with no *SECRET_KEY*.

django.contrib.admin The included administration app `django.contrib.admin` has for a long time shipped with a default set of static files such as JavaScript, images and stylesheets. Django 1.3 added a new contrib app `django.contrib.staticfiles` to handle such files in a generic way and defined conventions for static files included in apps.

Starting in Django 1.4, the admin's static files also follow this convention, to make the files easier to deploy. In previous versions of Django, it was also common to define an `ADMIN_MEDIA_PREFIX` setting to point to the URL where the admin's static files live on a Web server. This setting has now been deprecated and replaced by the more general setting *STATIC_URL*. Django will now expect to find the admin static files under the URL `<STATIC_URL>/admin/`.

If you've previously used a URL path for ADMIN_MEDIA_PREFIX (e.g. /media/) simply make sure *STATIC_URL* and *STATIC_ROOT* are configured and your Web server serves those files correctly. The development server continues to serve the admin files just like before. Read the static files howto for more details.

If your ADMIN_MEDIA_PREFIX is set to an specific domain (e.g. http://media.example.com/admin/), make sure to also set your *STATIC_URL* setting to the correct URL – for example, http://media.example.com/.

: If you're implicitly relying on the path of the admin static files within Django's source code, you'll need to update that path. The files were moved from django/contrib/admin/media/ to django/contrib/admin/static/admin/.

Supported browsers for the admin Django hasn't had a clear policy on which browsers are supported by the admin app. Our new policy formalizes existing practices: YUI's A-grade browsers should provide a fully-functional admin experience, with the notable exception of Internet Explorer 6, which is no longer supported.

Released over 10 years ago, IE6 imposes many limitations on modern Web development. The practical implications of this policy are that contributors are free to improve the admin without consideration for these limitations.

Obviously, this new policy **has no impact** on sites you develop using Django. It only applies to the Django admin. Feel free to develop apps compatible with any range of browsers.

Removed admin icons As part of an effort to improve the performance and usability of the admin's change-list sorting interface and *horizontal* and *vertical* "filter" widgets, some icon files were removed and grouped into two sprite files.

Specifically: selector-add.gif, selector-addall.gif, selector-remove.gif, selector-removeall.gif, selector_stacked-add.gif and selector_stacked-remove.gif were combined into selector-icons.gif; and arrow-up.gif and arrow-down.gif were combined into sorting-icons.gif.

If you used those icons to customize the admin, then you'll need to replace them with your own icons or get the files from a previous release.

CSS class names in admin forms To avoid conflicts with other common CSS class names (e.g. "button"), we added a prefix ("field-") to all CSS class names automatically generated from the form field names in the main admin forms, stacked inline forms and tabular inline cells. You'll need to take that prefix into account in your custom style sheets or JavaScript files if you previously used plain field names as selectors for custom styles or JavaScript transformations.

Compatibility with old signed data Django 1.3 changed the cryptographic signing mechanisms used in a number of places in Django. While Django 1.3 kept fallbacks that would accept hashes produced by the previous methods, these fallbacks are removed in Django 1.4.

So, if you upgrade to Django 1.4 directly from 1.2 or earlier, you may lose/invalidate certain pieces of data that have been cryptographically signed using an old method. To avoid this, use Django 1.3 first for a period of time to allow the signed data to expire naturally. The affected parts are detailed below, with 1) the consequences of ignoring this advice and 2) the amount of time you need to run Django 1.3 for the data to expire or become irrelevant.

- contrib.sessions data integrity check
  - Consequences: The user will be logged out, and session data will be lost.
  - Time period: Defined by *SESSION_COOKIE_AGE*.
- contrib.auth password reset hash

  - Consequences: Password reset links from before the upgrade will not work.
  - Time period: Defined by *PASSWORD_RESET_TIMEOUT_DAYS*.

Form-related hashes: these have a are much shorter lifetime and are relevant only for the short window where a user might fill in a form generated by the pre-upgrade Django instance and try to submit it to the upgraded Django instance:

- `contrib.comments` form security hash
  - Consequences: The user will see the validation error "Security hash failed."
  - Time period: The amount of time you expect users to take filling out comment forms.
- `FormWizard` security hash
  - Consequences: The user will see an error about the form having expired and will be sent back to the first page of the wizard, losing the data entered so far.
  - Time period: The amount of time you expect users to take filling out the affected forms.
- CSRF check
  - Note: This is actually a Django 1.1 fallback, not Django 1.2, and it applies only if you're upgrading from 1.1.
  - Consequences: The user will see a 403 error with any CSRF-protected POST form.
  - Time period: The amount of time you expect user to take filling out such forms.
- `contrib.auth` user password hash-upgrade sequence
  - Consequences: Each user's password will be updated to a stronger password hash when it's written to the database in 1.4. This means that if you upgrade to 1.4 and then need to downgrade to 1.3, version 1.3 won't be able to read the updated passwords.
  - Remedy: Set *PASSWORD_HASHERS* to use your original password hashing when you initially upgrade to 1.4. After you confirm your app works well with Django 1.4 and you won't have to roll back to 1.3, enable the new password hashes.

django.contrib.flatpages Starting in 1.4, the *FlatpageFallbackMiddleware* only adds a trailing slash and redirects if the resulting URL refers to an existing flatpage. For example, requesting `/notaflatpageoravalidurl` in a previous version would redirect to `/notaflatpageoravalidurl/`, which would subsequently raise a 404. Requesting `/notaflatpageoravalidurl` now will immediately raise a 404.

Also, redirects returned by flatpages are now permanent (with 301 status code), to match the behavior of *CommonMiddleware*.

Serialization of `datetime` and `time` As a consequence of time-zone support, and according to the ECMA-262 specification, we made changes to the JSON serializer:

- It includes the time zone for aware datetime objects. It raises an exception for aware time objects.
- It includes milliseconds for datetime and time objects. There is still some precision loss, because Python stores microseconds (6 digits) and JSON only supports milliseconds (3 digits). However, it's better than discarding microseconds entirely.

We changed the XML serializer to use the ISO8601 format for datetimes. The letter `T` is used to separate the date part from the time part, instead of a space. Time zone information is included in the `[+-]HH:MM` format.

Though the serializers now use these new formats when creating fixtures, they can still load fixtures that use the old format.

`supports_timezone` changed to `False` for SQLite The database feature `supports_timezone` used to be `True` for SQLite. Indeed, if you saved an aware datetime object, SQLite stored a string that included an UTC offset. However, this offset was ignored when loading the value back from the database, which could corrupt the data.

In the context of time-zone support, this flag was changed to `False`, and datetimes are now stored without time-zone information in SQLite. When *`USE_TZ`* is `False`, if you attempt to save an aware datetime object, Django raises an exception.

MySQLdb-specific exceptions The MySQL backend historically has raised `MySQLdb.OperationalError` when a query triggered an exception. We've fixed this bug, and we now raise *`django.db.DatabaseError`* instead. If you were testing for `MySQLdb.OperationalError`, you'll need to update your `except` clauses.

Database connection's thread-locality `DatabaseWrapper` objects (i.e. the connection objects referenced by `django.db.connection` and `django.db.connections["some_alias"]`) used to be thread-local. They are now global objects in order to be potentially shared between multiple threads. While the individual connection objects are now global, the `django.db.connections` dictionary referencing those objects is still thread-local. Therefore if you just use the ORM or `DatabaseWrapper.cursor()` then the behavior is still the same as before. Note, however, that `django.db.connection` does not directly reference the default `DatabaseWrapper` object anymore and is now a proxy to access that object's attributes. If you need to access the actual `DatabaseWrapper` object, use `django.db.connections[DEFAULT_DB_ALIAS]` instead.

As part of this change, all underlying SQLite connections are now enabled for potential thread-sharing (by passing the `check_same_thread=False` attribute to pysqlite). `DatabaseWrapper` however preserves the previous behavior by disabling thread-sharing by default, so this does not affect any existing code that purely relies on the ORM or on `DatabaseWrapper.cursor()`.

Finally, while it's now possible to pass connections between threads, Django doesn't make any effort to synchronize access to the underlying backend. Concurrency behavior is defined by the underlying backend implementation. Check their documentation for details.

***COMMENTS_BANNED_USERS_GROUP* setting** Django's comments has historically supported excluding the comments of a special user group, but we've never documented the feature properly and didn't enforce the exclusion in other parts of the app such as the template tags. To fix this problem, we removed the code from the feed class.

If you rely on the feature and want to restore the old behavior, use a custom comment model manager to exclude the user group, like this:

```
from django.conf import settings
from django.contrib.comments.managers import CommentManager

class BanningCommentManager(CommentManager):
    def get_query_set(self):
        qs = super(BanningCommentManager, self).get_query_set()
        if getattr(settings, 'COMMENTS_BANNED_USERS_GROUP', None):
            where = ['user_id NOT IN (SELECT user_id FROM auth_user_groups WHERE group_id = %s)']
            params = [settings.COMMENTS_BANNED_USERS_GROUP]
            qs = qs.extra(where=where, params=params)
        return qs
```

Save this model manager in your custom comment app (e.g., in `my_comments_app/managers.py`) and add it your custom comment app model:

```
from django.db import models
from django.contrib.comments.models import Comment

from my_comments_app.managers import BanningCommentManager
```

```
class CommentWithTitle(Comment):
    title = models.CharField(max_length=300)

    objects = BanningCommentManager()
```

***IGNORABLE_404_STARTS* and *IGNORABLE_404_ENDS* settings** Until Django 1.3, it was possible to exclude some URLs from Django's 404 error reporting by adding prefixes to `IGNORABLE_404_STARTS` and suffixes to `IGNORABLE_404_ENDS`.

In Django 1.4, these two settings are superseded by *`IGNORABLE_404_URLS`*, which is a list of compiled regular expressions. Django won't send an email for 404 errors on URLs that match any of them.

Furthermore, the previous settings had some rather arbitrary default values:

```
IGNORABLE_404_STARTS = ('/cgi-bin/', '/_vti_bin', '/_vti_inf')
IGNORABLE_404_ENDS = ('mail.pl', 'mailform.pl', 'mail.cgi', 'mailform.cgi',
                      'favicon.ico', '.php')
```

It's not Django's role to decide if your website has a legacy `/cgi-bin/` section or a `favicon.ico`. As a consequence, the default values of *`IGNORABLE_404_URLS`*, `IGNORABLE_404_STARTS`, and `IGNORABLE_404_ENDS` are all now empty.

If you have customized `IGNORABLE_404_STARTS` or `IGNORABLE_404_ENDS`, or if you want to keep the old default value, you should add the following lines in your settings file:

```
import re
IGNORABLE_404_URLS = (
    # for each <prefix> in IGNORABLE_404_STARTS
    re.compile(r'^<prefix>'),
    # for each <suffix> in IGNORABLE_404_ENDS
    re.compile(r'<suffix>$'),
)
```

Don't forget to escape characters that have a special meaning in a regular expression, such as periods.

CSRF protection extended to PUT and DELETE Previously, Django's CSRF protection provided protection only against POST requests. Since use of PUT and DELETE methods in AJAX applications is becoming more common, we now protect all methods not defined as safe by **RFC 2616** – i.e., we exempt GET, HEAD, OPTIONS and TRACE, and we enforce protection on everything else.

If you're using PUT or DELETE methods in AJAX applications, please see the *instructions about using AJAX and CSRF*.

Password reset view now accepts `subject_template_name` The `password_reset` view in `django.contrib.auth` now accepts a `subject_template_name` parameter, which is passed to the password save form as a keyword argument. If you are using this view with a custom password reset form, then you will need to ensure your form's `save()` method accepts this keyword argument.

`django.core.template_loaders` This was an alias to `django.template.loader` since 2005, and we've removed it without emitting a warning due to the length of the deprecation. If your code still referenced this, please use `django.template.loader` instead.

`django.db.models.fields.URLField.verify_exists` This functionality has been removed due to intractable performance and security issues. Any existing usage of `verify_exists` should be removed.

`django.core.files.storage.Storage.open` The `open` method of the base Storage class used to take an obscure parameter `mixin` that allowed you to dynamically change the base classes of the returned file object. This has been removed. In the rare case you relied on the `mixin` parameter, you can easily achieve the same by overriding the `open` method, like this:

```
from django.core.files import File
from django.core.files.storage import FileSystemStorage

class Spam(File):
    """
    Spam, spam, spam, spam and spam.
    """
    def ham(self):
        return 'eggs'

class SpamStorage(FileSystemStorage):
    """
    A custom file storage backend.
    """
    def open(self, name, mode='rb'):
        return Spam(open(self.path(name), mode))
```

YAML deserializer now uses `yaml.safe_load` `yaml.load` is able to construct any Python object, which may trigger arbitrary code execution if you process a YAML document that comes from an untrusted source. This feature isn't necessary for Django's YAML deserializer, whose primary use is to load fixtures consisting of simple objects. Even though fixtures are trusted data, the YAML deserializer now uses `yaml.safe_load` for additional security.

Session cookies now have the `httponly` flag by default Session cookies now include the `httponly` attribute by default to help reduce the impact of potential XSS attacks. As a consequence of this change, session cookie data, including sessionid, is no longer accessible from JavaScript in many browsers. For strict backwards compatibility, use `SESSION_COOKIE_HTTPONLY = False` in your settings file.

The `urlize` filter no longer escapes every URL When a URL contains a `%xx` sequence, where `xx` are two hexadecimal digits, *`urlize`* now assumes that the URL is already escaped and doesn't apply URL escaping again. This is wrong for URLs whose unquoted form contains a `%xx` sequence, but such URLs are very unlikely to happen in the wild, because they would confuse browsers too.

`assertTemplateUsed` and `assertTemplateNotUsed` as context manager It's now possible to check whether a template was used within a block of code with *`assertTemplateUsed()`* and *`assertTemplateNotUsed()`*. And they can be used as a context manager:

```
with self.assertTemplateUsed('index.html'):
    render_to_string('index.html')
with self.assertTemplateNotUsed('base.html'):
    render_to_string('index.html')
```

See the *assertion documentation* for more.

Database connections after running the test suite The default test runner no longer restores the database connections after tests' execution. This prevents the production database from being exposed to potential threads that would still be running and attempting to create new connections.

If your code relied on connections to the production database being created after tests' execution, then you can restore the previous behavior by subclassing `DjangoTestRunner` and overriding its `teardown_databases()` method.

Output of `manage.py help` *`manage.py help`* now groups available commands by application. If you depended on the output of this command – if you parsed it, for example – then you'll need to update your code. To get a list of all available management commands in a script, use *`manage.py help --commands`* instead.

`extends` template tag Previously, the *`extends`* tag used a buggy method of parsing arguments, which could lead to it erroneously considering an argument as a string literal when it wasn't. It now uses `parser.compile_filter`, like other tags.

The internals of the tag aren't part of the official stable API, but in the interests of full disclosure, the `ExtendsNode.__init__` definition has changed, which may break any custom tags that use this class.

Loading some incomplete fixtures no longer works Prior to 1.4, a default value was inserted for fixture objects that were missing a specific date or datetime value when auto_now or auto_now_add was set for the field. This was something that should not have worked, and in 1.4 loading such incomplete fixtures will fail. Because fixtures are a raw import, they should explicitly specify all field values, regardless of field options on the model.

Development Server Multithreading The development server is now is multithreaded by default. Use the *`runserver --nothreading`* option to disable the use of threading in the development server:

```
django-admin.py runserver --nothreading
```

Attributes disabled in markdown when safe mode set Prior to Django 1.4, attributes were included in any markdown output regardless of safe mode setting of the filter. With version > 2.1 of the Python-Markdown library, an enable_attributes option was added. When the safe argument is passed to the markdown filter, both the `safe_mode=True` and `enable_attributes=False` options are set. If using a version of the Python-Markdown library less than 2.1, a warning is issued that the output is insecure.

FormMixin get_initial returns an instance-specific dictionary In Django 1.3, the `get_initial` method of the *`django.views.generic.edit.FormMixin`* class was returning the class `initial` dictionary. This has been fixed to return a copy of this dictionary, so form instances can modify their initial data without messing with the class variable.

Features deprecated in 1.4

Old styles of calling `cache_page` decorator Some legacy ways of calling *`cache_page()`* have been deprecated. Please see the documentation for the correct way to use this decorator.

Support for PostgreSQL versions older than 8.2 Django 1.3 dropped support for PostgreSQL versions older than 8.0, and we suggested using a more recent version because of performance improvements and, more importantly, the end of upstream support periods for 8.0 and 8.1 was near (November 2010).

Django 1.4 takes that policy further and sets 8.2 as the minimum PostgreSQL version it officially supports.

Request exceptions are now always logged When we added logging support in Django in 1.3, the admin error email support was moved into the *django.utils.log.AdminEmailHandler*, attached to the 'django.request' logger. In order to maintain the established behavior of error emails, the 'django.request' logger was called only when *DEBUG* was False.

To increase the flexibility of error logging for requests, the 'django.request' logger is now called regardless of the value of *DEBUG*, and the default settings file for new projects now includes a separate filter attached to *django.utils.log.AdminEmailHandler* to prevent admin error emails in DEBUG mode:

```
'filters': {
    'require_debug_false': {
        '()': 'django.utils.log.RequireDebugFalse'
    }
},
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'filters': ['require_debug_false'],
        'class': 'django.utils.log.AdminEmailHandler'
    }
},
```

If your project was created prior to this change, your *LOGGING* setting will not include this new filter. In order to maintain backwards-compatibility, Django will detect that your 'mail_admins' handler configuration includes no 'filters' section and will automatically add this filter for you and issue a pending-deprecation warning. This will become a deprecation warning in Django 1.5, and in Django 1.6 the backwards-compatibility shim will be removed entirely.

The existence of any 'filters' key under the 'mail_admins' handler will disable this backward-compatibility shim and deprecation warning.

django.conf.urls.defaults Until Django 1.3, the functions *include()*, patterns() and *url()* plus *handler404*, *handler500* were located in a django.conf.urls.defaults module.

In Django 1.4, they live in *django.conf.urls*.

django.contrib.databrowse Databrowse has not seen active development for some time, and this does not show any sign of changing. There had been a suggestion for a GSOC project to integrate the functionality of databrowse into the admin, but no progress was made. While Databrowse has been deprecated, an enhancement of django.contrib.admin providing a similar feature set is still possible.

The code that powers Databrowse is licensed under the same terms as Django itself, so it's available to be adopted by an individual or group as a third-party project.

django.core.management.setup_environ This function temporarily modified sys.path in order to make the parent "project" directory importable under the old flat *startproject* layout. This function is now deprecated, as its path workarounds are no longer needed with the new manage.py and default project layout.

This function was never documented or part of the public API, but it was widely recommended for use in setting up a "Django environment" for a user script. These uses should be replaced by setting the DJANGO_SETTINGS_MODULE environment variable or using *django.conf.settings.configure()*.

django.core.management.execute_manager This function was previously used by manage.py to execute a management command. It is identical to django.core.management.execute_from_command_line, except that it first calls setup_environ,

which is now deprecated. As such, `execute_manager` is also deprecated; `execute_from_command_line` can be used instead. Neither of these functions is documented as part of the public API, but a deprecation path is needed due to use in existing `manage.py` files.

`is_safe` and `needs_autoescape` attributes of template filters Two flags, `is_safe` and `needs_autoescape`, define how each template filter interacts with Django's auto-escaping behavior. They used to be attributes of the filter function:

```
@register.filter
def noop(value):
    return value
noop.is_safe = True
```

However, this technique caused some problems in combination with decorators, especially *`@stringfilter`*. Now, the flags are keyword arguments of *`@register.filter`*:

```
@register.filter(is_safe=True)
def noop(value):
    return value
```

See *filters and auto-escaping* for more information.

Wildcard expansion of application names in *INSTALLED_APPS* Until Django 1.3, *`INSTALLED_APPS`* accepted wildcards in application names, like `django.contrib.*`. The expansion was performed by a filesystem-based implementation of `from <package> import *`. Unfortunately, this can't be done reliably.

This behavior was never documented. Since it is unpythonic and not obviously useful, it was removed in Django 1.4. If you relied on it, you must edit your settings file to list all your applications explicitly.

`HttpRequest.raw_post_data` renamed to `HttpRequest.body` This attribute was confusingly named `HttpRequest.raw_post_data`, but it actually provided the body of the HTTP request. It's been renamed to `HttpRequest.body`, and `HttpRequest.raw_post_data` has been deprecated.

`django.contrib.sitemaps` bug fix with potential performance implications In previous versions, `Paginator` objects used in sitemap classes were cached, which could result in stale site maps. We've removed the caching, so each request to a site map now creates a new Paginator object and calls the *`items()`* method of the *`Sitemap`* subclass. Depending on what your `items()` method is doing, this may have a negative performance impact. To mitigate the performance impact, consider using the caching framework within your `Sitemap` subclass.

Versions of Python-Markdown earlier than 2.1 Versions of Python-Markdown earlier than 2.1 do not support the option to disable attributes. As a security issue, earlier versions of this library will not be supported by the markup contrib app in 1.5 under an accelerated deprecation timeline.

9.1.8 1.3 release

Django 1.3.7 release notes

February 20, 2013

Django 1.3.7 corrects a packaging problem with yesterday's 1.3.6 release.

The release contained stray `.pyc` files that caused "bad magic number" errors when running with some versions of Python. This releases corrects this, and also fixes a bad documentation link in the project template `settings.py` file generated by `manage.py startproject`.

Django 1.3.6 release notes

February 19, 2013

Django 1.3.6 fixes four security issues present in previous Django releases in the 1.3 series.

This is the sixth bugfix/security release in the Django 1.3 series.

Host header poisoning

Some parts of Django – independent of end-user-written applications – make use of full URLs, including domain name, which are generated from the HTTP Host header. Django's documentation has for some time contained notes advising users on how to configure Web servers to ensure that only valid Host headers can reach the Django application. However, it has been reported to us that even with the recommended Web server configurations there are still techniques available for tricking many common Web servers into supplying the application with an incorrect and possibly malicious Host header.

For this reason, Django 1.3.6 adds a new setting, `ALLOWED_HOSTS`, which should contain an explicit list of valid host/domain names for this site. A request with a Host header not matching an entry in this list will raise `SuspiciousOperation` if `request.get_host()` is called. For full details see the documentation for the *`ALLOWED_HOSTS`* setting.

The default value for this setting in Django 1.3.6 is `['*']` (matching any host), for backwards-compatibility, but we strongly encourage all sites to set a more restrictive value.

This host validation is disabled when `DEBUG` is `True` or when running tests.

XML deserialization

The XML parser in the Python standard library is vulnerable to a number of attacks via external entities and entity expansion. Django uses this parser for deserializing XML-formatted database fixtures. The fixture deserializer is not intended for use with untrusted data, but in order to err on the side of safety in Django 1.3.6 the XML deserializer refuses to parse an XML document with a DTD (DOCTYPE definition), which closes off these attack avenues.

These issues in the Python standard library are CVE-2013-1664 and CVE-2013-1665. More information available from the Python security team.

Django's XML serializer does not create documents with a DTD, so this should not cause any issues with the typical round-trip from `dumpdata` to `loaddata`, but if you feed your own XML documents to the `loaddata` management command, you will need to ensure they do not contain a DTD.

Formset memory exhaustion

Previous versions of Django did not validate or limit the form-count data provided by the client in a formset's management form, making it possible to exhaust a server's available memory by forcing it to create very large numbers of forms.

In Django 1.3.6, all formsets have a strictly-enforced maximum number of forms (1000 by default, though it can be set higher via the `max_num` formset factory argument).

Admin history view information leakage

In previous versions of Django, an admin user without change permission on a model could still view the unicode representation of instances via their admin history log. Django 1.3.6 now limits the admin history log view for an object to users with change permission for that model.

Django 1.3.5 release notes

December 10, 2012

Django 1.3.5 addresses two security issues present in previous Django releases in the 1.3 series.

Please be aware that this security release is slightly different from previous ones. Both issues addressed here have been dealt with in prior security updates to Django. In one case, we have received ongoing reports of problems, and in the other we've chosen to take further steps to tighten up Django's code in response to independent discovery of potential problems from multiple sources.

Host header poisoning

Several earlier Django security releases focused on the issue of poisoning the HTTP Host header, causing Django to generate URLs pointing to arbitrary, potentially-malicious domains.

In response to further input received and reports of continuing issues following the previous release, we're taking additional steps to tighten Host header validation. Rather than attempt to accommodate all features HTTP supports here, Django's Host header validation attempts to support a smaller, but far more common, subset:

- Hostnames must consist of characters `[A-Za-z0-9]` plus hyphen ('-') or dot ('.').
- IP addresses – both IPv4 and IPv6 – are permitted.
- Port, if specified, is numeric.

Any deviation from this will now be rejected, raising the exception *`django.core.exceptions.SuspiciousOperation`*.

Redirect poisoning

Also following up on a previous issue: in July of this year, we made changes to Django's HTTP redirect classes, performing additional validation of the scheme of the URL to redirect to (since, both within Django's own supplied applications and many third-party applications, accepting a user-supplied redirect target is a common pattern).

Since then, two independent audits of the code turned up further potential problems. So, similar to the Host-header issue, we are taking steps to provide tighter validation in response to reported problems (primarily with third-party applications, but to a certain extent also within Django itself). This comes in two parts:

1. A new utility function, `django.utils.http.is_safe_url`, is added; this function takes a URL and a hostname, and checks that the URL is either relative, or if absolute matches the supplied hostname. This function is intended for use whenever user-supplied redirect targets are accepted, to ensure that such redirects cannot lead to arbitrary third-party sites.

2. All of Django's own built-in views – primarily in the authentication system – which allow user-supplied redirect targets now use `is_safe_url` to validate the supplied URL.

Django 1.3.4 release notes

October 17, 2012

This is the fourth release in the Django 1.3 series.

Host header poisoning

Some parts of Django – independent of end-user-written applications – make use of full URLs, including domain name, which are generated from the HTTP Host header. Some attacks against this are beyond Django's ability to control, and require the web server to be properly configured; Django's documentation has for some time contained notes advising users on such configuration.

Django's own built-in parsing of the Host header is, however, still vulnerable, as was reported to us recently. The Host header parsing in Django 1.3.3 and Django 1.4.1 – specifically, `django.http.HttpRequest.get_host()` – was incorrectly handling username/password information in the header. Thus, for example, the following Host header would be accepted by Django when running on "validsite.com":

```
Host: validsite.com:random@evilsite.com
```

Using this, an attacker can cause parts of Django – particularly the password-reset mechanism – to generate and display arbitrary URLs to users.

To remedy this, the parsing in `HttpRequest.get_host()` is being modified; Host headers which contain potentially dangerous content (such as username/password pairs) now raise the exception *`django.core.exceptions.SuspiciousOperation`*.

Details of this issue were initially posted online as a security advisory.

Django 1.3.3 release notes

August 1, 2012

Following Monday's security release of Django 1.3.2, we began receiving reports that one of the fixes applied was breaking Python 2.4 compatibility for Django 1.3. Since Python 2.4 is a supported Python version for that release series, this release fixes compatibility with Python 2.4.

Django 1.3.2 release notes

July 30, 2012

This is the second security release in the Django 1.3 series, fixing several security issues in Django 1.3. Django 1.3.2 is a recommended upgrade for all users of Django 1.3.

For a full list of issues addressed in this release, see the security advisory.

Django 1.3.1 release notes

September 9, 2011

Welcome to Django 1.3.1!

This is the first security release in the Django 1.3 series, fixing several security issues in Django 1.3. Django 1.3.1 is a recommended upgrade for all users of Django 1.3.

For a full list of issues addressed in this release, see the security advisory.

Django 1.3 release notes

March 23, 2011

Welcome to Django 1.3!

Nearly a year in the making, Django 1.3 includes quite a few *new features* and plenty of bug fixes and improvements to existing features. These release notes cover the new features in 1.3, as well as some *backwards-incompatible changes* you'll want to be aware of when upgrading from Django 1.2 or older versions.

Overview

Django 1.3's focus has mostly been on resolving smaller, long-standing feature requests, but that hasn't prevented a few fairly significant new features from landing, including:

- A framework for writing *class-based views*.
- Built-in support for *using Python's logging facilities*.
- Contrib support for *easy handling of static files*.
- Django's testing framework now supports (and ships with a copy of) *the unittest2 library*.

Wherever possible, of course, new features are introduced in a backwards-compatible manner per our API stability policy policy. As a result of this policy, Django 1.3 *begins the deprecation process for some features*.

Python compatibility

The release of Django 1.2 was notable for having the first shift in Django's Python compatibility policy; prior to Django 1.2, Django supported any 2.x version of Python from 2.3 up. As of Django 1.2, the minimum requirement was raised to Python 2.4.

Django 1.3 continues to support Python 2.4, but will be the final Django release series to do so; beginning with Django 1.4, the minimum supported Python version will be 2.5. A document outlining our full timeline for deprecating Python 2.x and moving to Python 3.x will be published shortly after the release of Django 1.3.

What's new in Django 1.3

Class-based views Django 1.3 adds a framework that allows you to use a class as a view. This means you can compose a view out of a collection of methods that can be subclassed and overridden to provide common views of data without having to write too much code.

Analogs of all the old function-based generic views have been provided, along with a completely generic view base class that can be used as the basis for reusable applications that can be easily extended.

See the documentation on class-based generic views for more details. There is also a document to help you convert your function-based generic views to class-based views.

Logging Django 1.3 adds framework-level support for Python's `logging` module. This means you can now easily configure and control logging as part of your Django project. A number of logging handlers and logging calls have been added to Django's own code as well – most notably, the error emails sent on a HTTP 500 server error are now handled as a logging activity. See the documentation on Django's logging interface for more details.

Extended static files handling Django 1.3 ships with a new contrib app – `django.contrib.staticfiles` – to help developers handle the static media files (images, CSS, JavaScript, etc.) that are needed to render a complete web page.

In previous versions of Django, it was common to place static assets in *MEDIA_ROOT* along with user-uploaded files, and serve them both at *MEDIA_URL*. Part of the purpose of introducing the `staticfiles` app is to make it easier to keep static files separate from user-uploaded files. Static assets should now go in `static/` subdirectories of your apps or in other static assets directories listed in *STATICFILES_DIRS*, and will be served at *STATIC_URL*.

See the reference documentation of the app for more details or learn how to manage static files.

unittest2 support Python 2.7 introduced some major changes to the `unittest` library, adding some extremely useful features. To ensure that every Django project can benefit from these new features, Django ships with a copy of unittest2, a copy of the Python 2.7 unittest library, backported for Python 2.4 compatibility.

To access this library, Django provides the `django.utils.unittest` module alias. If you are using Python 2.7, or you have installed `unittest2` locally, Django will map the alias to the installed version of the unittest library. Otherwise, Django will use its own bundled version of unittest2.

To take advantage of this alias, simply use:

```
from django.utils import unittest
```

wherever you would have historically used:

```
import unittest
```

If you want to continue to use the base unittest library, you can – you just won't get any of the nice new unittest2 features.

Transaction context managers Users of Python 2.5 and above may now use transaction management functions as context managers. For example:

```
with transaction.autocommit():
    # ...
```

Configurable delete-cascade *ForeignKey* and *OneToOneField* now accept an *on_delete* argument to customize behavior when the referenced object is deleted. Previously, deletes were always cascaded; available alternatives now include set null, set default, set to any value, protect, or do nothing.

For more information, see the *on_delete* documentation.

Contextual markers and comments for translatable strings For translation strings with ambiguous meaning, you can now use the `pgettext` function to specify the context of the string.

And if you just want to add some information for translators, you can also add special translator comments in the source.

For more information, see *Contextual markers* and *Comments for translators*.

Improvements to built-in template tags A number of improvements have been made to Django's built-in template tags:

- The *include* tag now accepts a `with` option, allowing you to specify context variables to the included template

- The *include* tag now accepts an `only` option, allowing you to exclude the current context from the included context
- The *with* tag now allows you to define multiple context variables in a single *with* block.
- The *load* tag now accepts a `from` argument, allowing you to load a single tag or filter from a library.

TemplateResponse It can sometimes be beneficial to allow decorators or middleware to modify a response *after* it has been constructed by the view. For example, you may want to change the template that is used, or put additional data into the context.

However, you can't (easily) modify the content of a basic *HttpResponse* after it has been constructed. To overcome this limitation, Django 1.3 adds a new *TemplateResponse* class. Unlike basic *HttpResponse* objects, *TemplateResponse* objects retain the details of the template and context that was provided by the view to compute the response. The final output of the response is not computed until it is needed, later in the response process.

For more details, see the documentation on the *TemplateResponse* class.

Caching changes Django 1.3 sees the introduction of several improvements to the Django's caching infrastructure.

Firstly, Django now supports multiple named caches. In the same way that Django 1.2 introduced support for multiple database connections, Django 1.3 allows you to use the new *CACHES* setting to define multiple named cache connections.

Secondly, *versioning*, *site-wide prefixing* and *transformation* have been added to the cache API.

Thirdly, *cache key creation* has been updated to take the request query string into account on `GET` requests.

Finally, support for pylibmc has been added to the memcached cache backend.

For more details, see the documentation on caching in Django.

Permissions for inactive users If you provide a custom auth backend with `supports_inactive_user` set to `True`, an inactive `User` instance will check the backend for permissions. This is useful for further centralizing the permission handling. See the authentication docs for more details.

GeoDjango The GeoDjango test suite is now included when *running the Django test suite* with `runtests.py` when using *spatial database backends*.

`MEDIA_URL` and `STATIC_URL` must end in a slash Previously, the *MEDIA_URL* setting only required a trailing slash if it contained a suffix beyond the domain name.

A trailing slash is now *required* for *MEDIA_URL* and the new *STATIC_URL* setting as long as it is not blank. This ensures there is a consistent way to combine paths in templates.

Project settings which provide either of both settings without a trailing slash will now raise a `PendingDeprecationWarning`.

In Django 1.4 this same condition will raise `DeprecationWarning`, and in Django 1.5 will raise an `ImproperlyConfigured` exception.

Everything else Django 1.1 and 1.2 added lots of big ticket items to Django, like multiple-database support, model validation, and a session-based messages framework. However, this focus on big features came at the cost of lots of smaller features.

To compensate for this, the focus of the Django 1.3 development process has been on adding lots of smaller, long standing feature requests. These include:

- Improved tools for accessing and manipulating the current *Site* object in the sites framework.
- A *RequestFactory* for mocking requests in tests.
- A new test assertion – *assertNumQueries()* – making it easier to test the database activity associated with a view.
- Support for lookups spanning relations in admin's *list_filter*.
- Support for HTTPOnly cookies.
- *mail_admins()* and *mail_managers()* now support easily attaching HTML content to messages.
- *EmailMessage* now supports CC's.
- Error emails now include more of the detail and formatting of the debug server error page.
- *simple_tag()* now accepts a `takes_context` argument, making it easier to write simple template tags that require access to template context.
- A new *render()* shortcut – an alternative to `django.shortcuts.render_to_response()` providing a *RequestContext* by default.
- Support for combining *F expressions* with timedelta values when retrieving or updating database values.

Backwards-incompatible changes in 1.3

CSRF validation now applies to AJAX requests Prior to Django 1.2.5, Django's CSRF-prevention system exempted AJAX requests from CSRF verification; due to security issues reported to us, however, *all* requests are now subjected to CSRF verification. Consult the Django CSRF documentation for details on how to handle CSRF verification in AJAX requests.

Restricted filters in admin interface Prior to Django 1.2.5, the Django administrative interface allowed filtering on any model field or relation – not just those specified in `list_filter` – via query string manipulation. Due to security issues reported to us, however, query string lookup arguments in the admin must be for fields or relations specified in `list_filter` or `date_hierarchy`.

Deleting a model doesn't delete associated files In earlier Django versions, when a model instance containing a *FileField* was deleted, *FileField* took it upon itself to also delete the file from the backend storage. This opened the door to several data-loss scenarios, including rolled-back transactions and fields on different models referencing the same file. In Django 1.3, when a model is deleted the *FileField*'s `delete()` method won't be called. If you need cleanup of orphaned files, you'll need to handle it yourself (for instance, with a custom management command that can be run manually or scheduled to run periodically via e.g. cron).

PasswordInput default rendering behavior The *PasswordInput* form widget, intended for use with form fields which represent passwords, accepts a boolean keyword argument `render_value` indicating whether to send its data back to the browser when displaying a submitted form with errors. Prior to Django 1.3, this argument defaulted to `True`, meaning that the submitted password would be sent back to the browser as part of the form. Developers who wished to add a bit of additional security by excluding that value from the redisplayed form could instantiate a *PasswordInput* passing `render_value=False` .

Due to the sensitive nature of passwords, however, Django 1.3 takes this step automatically; the default value of `render_value` is now `False`, and developers who want the password value returned to the browser on a submission with errors (the previous behavior) must now explicitly indicate this. For example:

```
class LoginForm(forms.Form):
    username = forms.CharField(max_length=100)
    password = forms.CharField(widget=forms.PasswordInput(render_value=True))
```

Clearable default widget for FileField Django 1.3 now includes a *ClearableFileInput* form widget in addition to *FileInput*. ClearableFileInput renders with a checkbox to clear the field's value (if the field has a value and is not required); FileInput provided no means for clearing an existing file from a FileField.

ClearableFileInput is now the default widget for a FileField, so existing forms including FileField without assigning a custom widget will need to account for the possible extra checkbox in the rendered form output.

To return to the previous rendering (without the ability to clear the FileField), use the FileInput widget in place of ClearableFileInput. For instance, in a ModelForm for a hypothetical Document model with a FileField named document:

```
from django import forms
from myapp.models import Document

class DocumentForm(forms.ModelForm):
    class Meta:
        model = Document
        widgets = {'document': forms.FileInput}
```

New index on database session table Prior to Django 1.3, the database table used by the database backend for the sessions app had no index on the expire_date column. As a result, date-based queries on the session table – such as the query that is needed to purge old sessions – would be very slow if there were lots of sessions.

If you have an existing project that is using the database session backend, you don't have to do anything to accommodate this change. However, you may get a significant performance boost if you manually add the new index to the session table. The SQL that will add the index can be found by running the sqlindexes admin command:

```
python manage.py sqlindexes sessions
```

No more naughty words Django has historically provided (and enforced) a list of profanities. The comments app has enforced this list of profanities, preventing people from submitting comments that contained one of those profanities.

Unfortunately, the technique used to implement this profanities list was woefully naive, and prone to the Scunthorpe problem. Improving the built-in filter to fix this problem would require significant effort, and since natural language processing isn't the normal domain of a web framework, we have "fixed" the problem by making the list of prohibited words an empty list.

If you want to restore the old behavior, simply put a PROFANITIES_LIST setting in your settings file that includes the words that you want to prohibit (see the commit that implemented this change if you want to see the list of words that was historically prohibited). However, if avoiding profanities is important to you, you would be well advised to seek out a better, less naive approach to the problem.

Localflavor changes Django 1.3 introduces the following backwards-incompatible changes to local flavors:

- Canada (ca) – The province "Newfoundland and Labrador" has had its province code updated to "NL", rather than the older "NF". In addition, the Yukon Territory has had its province code corrected to "YT", instead of "YK".
- Indonesia (id) – The province "Nanggroe Aceh Darussalam (NAD)" has been removed from the province list in favor of the new official designation "Aceh (ACE)".

- United States of America (us) – The list of "states" used by `USStateField` has expanded to include Armed Forces postal codes. This is backwards-incompatible if you were relying on `USStateField` not including them.

FormSet updates In Django 1.3 `FormSet` creation behavior is modified slightly. Historically the class didn't make a distinction between not being passed data and being passed empty dictionary. This was inconsistent with behavior in other parts of the framework. Starting with 1.3 if you pass in empty dictionary the `FormSet` will raise a `ValidationError`.

For example with a `FormSet`:

```
>>> class ArticleForm(Form):
...     title = CharField()
...     pub_date = DateField()
>>> ArticleFormSet = formset_factory(ArticleForm)
```

the following code will raise a `ValidationError`:

```
>>> ArticleFormSet({})
Traceback (most recent call last):
...
ValidationError: [u'ManagementForm data is missing or has been tampered with']
```

if you need to instantiate an empty `FormSet`, don't pass in the data or use `None`:

```
>>> formset = ArticleFormSet()
>>> formset = ArticleFormSet(data=None)
```

Callables in templates Previously, a callable in a template would only be called automatically as part of the variable resolution process if it was retrieved via attribute lookup. This was an inconsistency that could result in confusing and unhelpful behavior:

```
>>> Template("{{ user.get_full_name }}").render(Context({'user': user}))
u'Joe Bloggs'
>>> Template("{{ full_name }}").render(Context({'full_name': user.get_full_name}))
u'&lt;bound method User.get_full_name of &lt;...
```

This has been resolved in Django 1.3 - the result in both cases will be `u'Joe Bloggs'`. Although the previous behavior was not useful for a template language designed for web designers, and was never deliberately supported, it is possible that some templates may be broken by this change.

Use of custom SQL to load initial data in tests Django provides a custom SQL hooks as a way to inject hand-crafted SQL into the database synchronization process. One of the possible uses for this custom SQL is to insert data into your database. If your custom SQL contains `INSERT` statements, those insertions will be performed every time your database is synchronized. This includes the synchronization of any test databases that are created when you run a test suite.

However, in the process of testing the Django 1.3, it was discovered that this feature has never completely worked as advertised. When using database backends that don't support transactions, or when using a TransactionTestCase, data that has been inserted using custom SQL will not be visible during the testing process.

Unfortunately, there was no way to rectify this problem without introducing a backwards incompatibility. Rather than leave SQL-inserted initial data in an uncertain state, Django now enforces the policy that data inserted by custom SQL will *not* be visible during testing.

This change only affects the testing process. You can still use custom SQL to load data into your production database as part of the `syncdb` process. If you require data to exist during test conditions, you should either insert it using *test fixtures*, or using the `setUp()` method of your test case.

Changed priority of translation loading Work has been done to simplify, rationalize and properly document the algorithm used by Django at runtime to build translations from the different translations found on disk, namely:

For translatable literals found in Python code and templates (`'django'` gettext domain):

- Priorities of translations included with applications listed in the *INSTALLED_APPS* setting were changed. To provide a behavior consistent with other parts of Django that also use such setting (templates, etc.) now, when building the translation that will be made available, the apps listed first have higher precedence than the ones listed later.
- Now it is possible to override the translations shipped with applications by using the *LOCALE_PATHS* setting whose translations have now higher precedence than the translations of *INSTALLED_APPS* applications. The relative priority among the values listed in this setting has also been modified so the paths listed first have higher precedence than the ones listed later.
- The `locale` subdirectory of the directory containing the settings, that usually coincides with and is known as the *project directory* is being deprecated in this release as a source of translations. (the precedence of these translations is intermediate between applications and *LOCALE_PATHS* translations). See the *corresponding deprecated features section* of this document.

For translatable literals found in JavaScript code (`'djangojs'` gettext domain):

- Similarly to the `'django'` domain translations: Overriding of translations shipped with applications by using the *LOCALE_PATHS* setting is now possible for this domain too. These translations have higher precedence than the translations of Python packages passed to the *javascript_catalog view*. Paths listed first have higher precedence than the ones listed later.
- Translations under the `locale` subdirectory of the *project directory* have never been taken in account for JavaScript translations and remain in the same situation considering the deprecation of such location.

Transaction management When using managed transactions – that is, anything but the default autocommit mode – it is important when a transaction is marked as "dirty". Dirty transactions are committed by the `commit_on_success` decorator or the `django.middleware.transaction.TransactionMiddleware`, and `commit_manually` forces them to be closed explicitly; clean transactions "get a pass", which means they are usually rolled back at the end of a request when the connection is closed.

Until Django 1.3, transactions were only marked dirty when Django was aware of a modifying operation performed in them; that is, either some model was saved, some bulk update or delete was performed, or the user explicitly called `transaction.set_dirty()`. In Django 1.3, a transaction is marked dirty when *any* database operation is performed.

As a result of this change, you no longer need to set a transaction dirty explicitly when you execute raw SQL or use a data-modifying `SELECT`. However, you *do* need to explicitly close any read-only transactions that are being managed using `commit_manually()`. For example:

```
@transaction.commit_manually
def my_view(request, name):
    obj = get_object_or_404(MyObject, name__iexact=name)
    return render_to_response('template', {'object':obj})
```

Prior to Django 1.3, this would work without error. However, under Django 1.3, this will raise a *TransactionManagementError* because the read operation that retrieves the `MyObject` instance leaves the transaction in a dirty state.

No password reset for inactive users Prior to Django 1.3, inactive users were able to request a password reset email and reset their password. In Django 1.3 inactive users will receive the same message as a nonexistent account.

Password reset view now accepts `from_email` The *`django.contrib.auth.views.password_reset()`* view now accepts a `from_email` parameter, which is passed to the `password_reset_form`'s `save()` method as a keyword argument. If you are using this view with a custom password reset form, then you will need to ensure your form's `save()` method accepts this keyword argument.

Features deprecated in 1.3

Django 1.3 deprecates some features from earlier releases. These features are still supported, but will be gradually phased out over the next few release cycles.

Code taking advantage of any of the features below will raise a `PendingDeprecationWarning` in Django 1.3. This warning will be silent by default, but may be turned on using Python's `warnings` module, or by running Python with a `-Wd` or `-Wall` flag.

In Django 1.4, these warnings will become a `DeprecationWarning`, which is *not* silent. In Django 1.5 support for these features will be removed entirely.

:

For more details, see the documentation Django's release process and our deprecation timeline.

`mod_python` support The `mod_python` library has not had a release since 2007 or a commit since 2008. The Apache Foundation board voted to remove `mod_python` from the set of active projects in its version control repositories, and its lead developer has shifted all of his efforts toward the lighter, slimmer, more stable, and more flexible `mod_wsgi` backend.

If you are currently using the `mod_python` request handler, you should redeploy your Django projects using another request handler. mod_wsgi is the request handler recommended by the Django project, but FastCGI is also supported. Support for `mod_python` deployment will be removed in Django 1.5.

Function-based generic views As a result of the introduction of class-based generic views, the function-based generic views provided by Django have been deprecated. The following modules and the views they contain have been deprecated:

- `django.views.generic.create_update`
- `django.views.generic.date_based`
- `django.views.generic.list_detail`
- `django.views.generic.simple`

Test client response `template` attribute Django's *test client* returns *`Response`* objects annotated with extra testing information. In Django versions prior to 1.3, this included a `template` attribute containing information about templates rendered in generating the response: either None, a single *`Template`* object, or a list of *`Template`* objects. This inconsistency in return values (sometimes a list, sometimes not) made the attribute difficult to work with.

In Django 1.3 the `template` attribute is deprecated in favor of a new *`templates`* attribute, which is always a list, even if it has only a single element or no elements.

`DjangoTestRunner` As a result of the introduction of support for unittest2, the features of `django.test.simple.DjangoTestRunner` (including fail-fast and Ctrl-C test termination) have been made redundant. In view of this redundancy, `DjangoTestRunner` has been turned into an empty placeholder class, and will be removed entirely in Django 1.5.

Changes to `url` and `ssi` Most template tags will allow you to pass in either constants or variables as arguments – for example:

```
{% extends "base.html" %}
```

allows you to specify a base template as a constant, but if you have a context variable `templ` that contains the value `base.html`:

```
{% extends templ %}
```

is also legal.

However, due to an accident of history, the `url` and `ssi` are different. These tags use the second, quoteless syntax, but interpret the argument as a constant. This means it isn't possible to use a context variable as the target of a `url` and `ssi` tag.

Django 1.3 marks the start of the process to correct this historical accident. Django 1.3 adds a new template library – `future` – that provides alternate implementations of the `url` and `ssi` template tags. This `future` library implement behavior that makes the handling of the first argument consistent with the handling of all other variables. So, an existing template that contains:

```
{% url sample %}
```

should be replaced with:

```
{% load url from future %}
{% url 'sample' %}
```

The tags implementing the old behavior have been deprecated, and in Django 1.5, the old behavior will be replaced with the new behavior. To ensure compatibility with future versions of Django, existing templates should be modified to use the new `future` libraries and syntax.

Changes to the login methods of the admin In previous version the admin app defined login methods in multiple locations and ignored the almost identical implementation in the already used auth app. A side effect of this duplication was the missing adoption of the changes made in r12634 to support a broader set of characters for usernames.

This release refactors the admin's login mechanism to use a subclass of the *`AuthenticationForm`* instead of a manual form validation. The previously undocumented method `'django.contrib.admin.sites.AdminSite.display_login_form'` has been removed in favor of a new *`login_form`* attribute.

`reset` and `sqlreset` management commands Those commands have been deprecated. The `flush` and `sqlflush` commands can be used to delete everything. You can also use ALTER TABLE or DROP TABLE statements manually.

GeoDjango

- The function-based *`TEST_RUNNER`* previously used to execute the GeoDjango test suite, `django.contrib.gis.tests.run_gis_tests`, was deprecated for the class-based runner, `django.contrib.gis.tests.GeoDjangoTestSuiteRunner`.

- Previously, calling *transform()* would silently do nothing when GDAL wasn't available. Now, a *GEOSException* is properly raised to indicate possible faulty application code. A warning is now raised if *transform()* is called when the SRID of the geometry is less than 0 or None.

CZBirthNumberField.clean Previously this field's clean() method accepted a second, gender, argument which allowed stronger validation checks to be made, however since this argument could never actually be passed from the Django form machinery it is now pending deprecation.

CompatCookie Previously, django.http exposed an undocumented CompatCookie class, which was a bugfix wrapper around the standard library SimpleCookie. As the fixes are moving upstream, this is now deprecated - you should use from django.http import SimpleCookie instead.

Loading of *project-level* translations This release of Django starts the deprecation process for inclusion of translations located under the so-called *project path* in the translation building process performed at runtime. The *LOCALE_PATHS* setting can be used for the same task by adding the filesystem path to a locale directory containing project-level translations to the value of that setting.

Rationale for this decision:

- The *project path* has always been a loosely defined concept (actually, the directory used for locating project-level translations is the directory containing the settings module) and there has been a shift in other parts of the framework to stop using it as a reference for location of assets at runtime.
- Detection of the locale subdirectory tends to fail when the deployment scenario is more complex than the basic one. e.g. it fails when the settings module is a directory (ticket #10765).
- There are potential strange development- and deployment-time problems like the fact that the project_dir/locale/ subdir can generate spurious error messages when the project directory is added to the Python path (manage.py runserver does this) and then it clashes with the equally named standard library module, this is a typical warning message:

```
/usr/lib/python2.6/gettext.py:49: ImportWarning: Not importing directory '/path/to/project/local
import locale, copy, os, re, struct, sys
```

- This location wasn't included in the translation building process for JavaScript literals. This deprecation removes such inconsistency.

PermWrapper moved to django.contrib.auth.context_processors In Django 1.2, we began the process of changing the location of the auth context processor from django.core.context_processors to django.contrib.auth.context_processors. However, the PermWrapper support class was mistakenly omitted from that migration. In Django 1.3, the PermWrapper class has also been moved to django.contrib.auth.context_processors, along with the PermLookupDict support class. The new classes are functionally identical to their old versions; only the module location has changed.

Removal of XMLField When Django was first released, Django included an XMLField that performed automatic XML validation for any field input. However, this validation function hasn't been performed since the introduction of newforms, prior to the 1.0 release. As a result, XMLField as currently implemented is functionally indistinguishable from a simple *TextField*.

For this reason, Django 1.3 has fast-tracked the deprecation of XMLField – instead of a two-release deprecation, XMLField will be removed entirely in Django 1.4.

It's easy to update your code to accommodate this change – just replace all uses of XMLField with TextField, and remove the schema_path keyword argument (if it is specified).

9.1.9 1.2 release

Django 1.2.7 release notes

September 10, 2011

Welcome to Django 1.2.7!

This is the seventh bugfix/security release in the Django 1.2 series. It replaces Django 1.2.6 due to problems with the 1.2.6 release tarball. Django 1.2.7 is a recommended upgrade for all users of any Django release in the 1.2.X series.

For more information, see the release advisory.

Django 1.2.6 release notes

September 9, 2011

Welcome to Django 1.2.6!

This is the sixth bugfix/security release in the Django 1.2 series, fixing several security issues present in Django 1.2.5. Django 1.2.6 is a recommended upgrade for all users of any Django release in the 1.2.X series.

For a full list of issues addressed in this release, see the security advisory.

Django 1.2.5 release notes

Welcome to Django 1.2.5!

This is the fifth "bugfix" release in the Django 1.2 series, improving the stability and performance of the Django 1.2 codebase.

With four exceptions, Django 1.2.5 maintains backwards compatibility with Django 1.2.4. It also contains a number of fixes and other improvements. Django 1.2.5 is a recommended upgrade for any development or deployment currently using or targeting Django 1.2.

For full details on the new features, backwards incompatibilities, and deprecated features in the 1.2 branch, see the Django 1.2 release notes.

Backwards incompatible changes

CSRF exception for AJAX requests Django includes a CSRF-protection mechanism, which makes use of a token inserted into outgoing forms. Middleware then checks for the token's presence on form submission, and validates it.

Prior to Django 1.2.5, our CSRF protection made an exception for AJAX requests, on the following basis:

- Many AJAX toolkits add an X-Requested-With header when using XMLHttpRequest.
- Browsers have strict same-origin policies regarding XMLHttpRequest.
- In the context of a browser, the only way that a custom header of this nature can be added is with XMLHttpRequest.

Therefore, for ease of use, we did not apply CSRF checks to requests that appeared to be AJAX on the basis of the X-Requested-With header. The Ruby on Rails web framework had a similar exemption.

Recently, engineers at Google made members of the Ruby on Rails development team aware of a combination of browser plugins and redirects which can allow an attacker to provide custom HTTP headers on a request to any website. This can allow a forged request to appear to be an AJAX request, thereby defeating CSRF protection which trusts the same-origin nature of AJAX requests.

Michael Koziarski of the Rails team brought this to our attention, and we were able to produce a proof-of-concept demonstrating the same vulnerability in Django's CSRF handling.

To remedy this, Django will now apply full CSRF validation to all requests, regardless of apparent AJAX origin. This is technically backwards-incompatible, but the security risks have been judged to outweigh the compatibility concerns in this case.

Additionally, Django will now accept the CSRF token in the custom HTTP header X-CSRFTOKEN, as well as in the form submission itself, for ease of use with popular JavaScript toolkits which allow insertion of custom headers into all AJAX requests.

Please see the *CSRF docs for example jQuery code* that demonstrates this technique, ensuring that you are looking at the documentation for your version of Django, as the exact code necessary is different for some older versions of Django.

FileField no longer deletes files In earlier Django versions, when a model instance containing a `FileField` was deleted, `FileField` took it upon itself to also delete the file from the backend storage. This opened the door to several potentially serious data-loss scenarios, including rolled-back transactions and fields on different models referencing the same file. In Django 1.2.5, `FileField` will never delete files from the backend storage. If you need cleanup of orphaned files, you'll need to handle it yourself (for instance, with a custom management command that can be run manually or scheduled to run periodically via e.g. cron).

Use of custom SQL to load initial data in tests Django provides a custom SQL hooks as a way to inject hand-crafted SQL into the database synchronization process. One of the possible uses for this custom SQL is to insert data into your database. If your custom SQL contains `INSERT` statements, those insertions will be performed every time your database is synchronized. This includes the synchronization of any test databases that are created when you run a test suite.

However, in the process of testing the Django 1.3, it was discovered that this feature has never completely worked as advertised. When using database backends that don't support transactions, or when using a TransactionTestCase, data that has been inserted using custom SQL will not be visible during the testing process.

Unfortunately, there was no way to rectify this problem without introducing a backwards incompatibility. Rather than leave SQL-inserted initial data in an uncertain state, Django now enforces the policy that data inserted by custom SQL will *not* be visible during testing.

This change only affects the testing process. You can still use custom SQL to load data into your production database as part of the `syncdb` process. If you require data to exist during test conditions, you should either insert it using *test fixtures*, or using the `setUp()` method of your test case.

ModelAdmin.lookup_allowed signature changed Django 1.2.4 introduced a method `lookup_allowed` on `ModelAdmin`, to cope with a security issue (changeset [15033]). Although this method was never documented, it seems some people have overridden `lookup_allowed`, especially to cope with regressions introduced by that changeset. While the method is still undocumented and not marked as stable, it may be helpful to know that the signature of this function has changed.

Django 1.2.4 release notes

Welcome to Django 1.2.4!

This is the fourth "bugfix" release in the Django 1.2 series, improving the stability and performance of the Django 1.2 codebase.

With one exception, Django 1.2.4 maintains backwards compatibility with Django 1.2.3. It also contains a number of fixes and other improvements. Django 1.2.4 is a recommended upgrade for any development or deployment currently using or targeting Django 1.2.

For full details on the new features, backwards incompatibilities, and deprecated features in the 1.2 branch, see the Django 1.2 release notes.

Backwards incompatible changes

Restricted filters in admin interface The Django administrative interface, django.contrib.admin, supports filtering of displayed lists of objects by fields on the corresponding models, including across database-level relationships. This is implemented by passing lookup arguments in the querystring portion of the URL, and options on the ModelAdmin class allow developers to specify particular fields or relationships which will generate automatic links for filtering.

One historically-undocumented and -unofficially-supported feature has been the ability for a user with sufficient knowledge of a model's structure and the format of these lookup arguments to invent useful new filters on the fly by manipulating the querystring.

However, it has been demonstrated that this can be abused to gain access to information outside of an admin user's permissions; for example, an attacker with access to the admin and sufficient knowledge of model structure and relations could construct query strings which – with repeated use of regular-expression lookups supported by the Django database API – expose sensitive information such as users' password hashes.

To remedy this, django.contrib.admin will now validate that querystring lookup arguments either specify only fields on the model being viewed, or cross relations which have been explicitly whitelisted by the application developer using the pre-existing mechanism mentioned above. This is backwards-incompatible for any users relying on the prior ability to insert arbitrary lookups.

One new feature

Ordinarily, a point release would not include new features, but in the case of Django 1.2.4, we have made an exception to this rule.

One of the bugs fixed in Django 1.2.4 involves a set of circumstances whereby a running a test suite on a multiple database configuration could cause the original source database (i.e., the actual production database) to be dropped, causing catastrophic loss of data. In order to provide a fix for this problem, it was necessary to introduce a new setting – *TEST_DEPENDENCIES* – that allows you to define any creation order dependencies in your database configuration.

Most users – even users with multiple-database configurations – need not be concerned about the data loss bug, or the manual configuration of *TEST_DEPENDENCIES*. See the original problem report documentation on *controlling the creation order of test databases* for details.

GeoDjango

The function-based *TEST_RUNNER* previously used to execute the GeoDjango test suite, `django.contrib.gis.tests.run_gis_tests`, was finally deprecated in favor of a class-based test runner, `django.contrib.gis.tests.GeoDjangoTestSuiteRunner`, added in this release.

In addition, the GeoDjango test suite is now included when *running the Django test suite* with `runtests.py` and using *spatial database backends*.

Django 1.2.3 release notes

Django 1.2.3 fixed a couple of release problems in the 1.2.2 release and was released two days after 1.2.2.

This release corrects the following problems:

- The patch applied for the security issue covered in Django 1.2.2 caused issues with non-ASCII responses using CSRF tokens.
- The patch also caused issues with some forms, most notably the user-editing forms in the Django administrative interface.
- The packaging manifest did not contain the full list of required files.

Django 1.2.2 release notes

Welcome to Django 1.2.2!

This is the second "bugfix" release in the Django 1.2 series, improving the stability and performance of the Django 1.2 codebase.

Django 1.2.2 maintains backwards compatibility with Django 1.2.1, but contain a number of fixes and other improvements. Django 1.2.2 is a recommended upgrade for any development or deployment currently using or targeting Django 1.2.

For full details on the new features, backwards incompatibilities, and deprecated features in the 1.2 branch, see the Django 1.2 release notes.

One new feature

Ordinarily, a point release would not include new features, but in the case of Django 1.2.2, we have made an exception to this rule.

In order to test a bug fix that forms part of the 1.2.2 release, it was necessary to add a feature – the `enforce_csrf_checks` flag – to the *test client*. This flag forces the test client to perform full CSRF checks on forms. The default behavior of the test client hasn't changed, but if you want to do CSRF checks with the test client, it is now possible to do so.

Django 1.2.1 release notes

Django 1.2.1 was released almost immediately after 1.2.0 to correct two small bugs: one was in the documentation packaging script, the other was a bug that affected datetime form field widgets when localization was enabled.

Django 1.2 release notes

May 17, 2010.

Welcome to Django 1.2!

Nearly a year in the making, Django 1.2 packs an impressive list of *new features* and lots of bug fixes. These release notes cover the new features, as well as important changes you'll want to be aware of when upgrading from Django 1.1 or older versions.

Overview

Django 1.2 introduces several large, important new features, including:

- Support for *multiple database connections* in a single Django instance.
- *Model validation* inspired by Django's form validation.

- Vastly *improved protection against Cross-Site Request Forgery* (CSRF).
- A new *user "messages" framework* with support for cookie- and session-based message for both anonymous and authenticated users.
- Hooks for *object-level permissions*, *permissions for anonymous users*, and *more flexible username requirements*.
- Customization of email sending via *email backends*.
- New *"smart" if template tag* which supports comparison operators.

These are just the highlights; full details and a complete list of features *may be found below*.

:

Django Advent covered the release of Django 1.2 with a series of articles and tutorials that cover some of the new features in depth.

Wherever possible these features have been introduced in a backwards-compatible manner per our API stability policy policy.

However, a handful of features *have* changed in ways that, for some users, will be backwards-incompatible. The big changes are:

- Support for Python 2.3 has been dropped. See the full notes below.
- The new CSRF protection framework is not backwards-compatible with the old system. Users of the old system will not be affected until the old system is removed in Django 1.4.

  However, upgrading to the new CSRF protection framework requires a few important backwards-incompatible changes, detailed in *CSRF Protection*, below.
- Authors of custom `Field` subclasses should be aware that a number of methods have had a change in prototype, detailed under *get_db_prep_*() methods on Field*, below.
- The internals of template tags have changed somewhat; authors of custom template tags that need to store state (e.g. custom control flow tags) should ensure that their code follows the new rules for *stateful template tags*
- The `user_passes_test()`, `login_required()`, and `permission_required()`, decorators from `django.contrib.auth` only apply to functions and no longer work on methods. There's a simple one-line fix *detailed below*.

Again, these are just the big features that will affect the most users. Users upgrading from previous versions of Django are heavily encouraged to consult the complete list of *backwards-incompatible changes* and the list of *deprecated features*.

Python compatibility

While not a new feature, it's important to note that Django 1.2 introduces the first shift in our Python compatibility policy since Django's initial public debut. Previous Django releases were tested and supported on 2.x Python versions from 2.3 up; Django 1.2, however, drops official support for Python 2.3. As such, the minimum Python version required for Django is now 2.4, and Django is tested and supported on Python 2.4, 2.5 and 2.6, and will be supported on the as-yet-unreleased Python 2.7.

This change should affect only a small number of Django users, as most operating-system vendors today are shipping Python 2.4 or newer as their default version. If you're still using Python 2.3, however, you'll need to stick to Django 1.1 until you can upgrade; per our support policy, Django 1.1 will continue to receive security support until the release of Django 1.3.

A roadmap for Django's overall 2.x Python support, and eventual transition to Python 3.x, is currently being developed, and will be announced prior to the release of Django 1.3.

What's new in Django 1.2

Support for multiple databases Django 1.2 adds the ability to use more than one database in your Django project. Queries can be issued at a specific database with the `using()` method on `QuerySet` objects. Individual objects can be saved to a specific database by providing a `using` argument when you call `save()`.

Model validation Model instances now have support for *validating their own data*, and both model and form fields now accept configurable lists of validators specifying reusable, encapsulated validation behavior. Note, however, that validation must still be performed explicitly. Simply invoking a model instance's `save()` method will not perform any validation of the instance's data.

Improved CSRF protection Django now has much improved protection against Cross-Site Request Forgery (CSRF) attacks. This type of attack occurs when a malicious website contains a link, a form button or some JavaScript that is intended to perform some action on your website, using the credentials of a logged-in user who visits the malicious site in their browser. A related type of attack, "login CSRF," where an attacking site tricks a user's browser into logging into a site with someone else's credentials, is also covered.

Messages framework Django now includes a robust and configurable messages framework with built-in support for cookie- and session-based messaging, for both anonymous and authenticated clients. The messages framework replaces the deprecated user message API and allows you to temporarily store messages in one request and retrieve them for display in a subsequent request (usually the next one).

Object-level permissions A foundation for specifying permissions at the per-object level has been added. Although there is no implementation of this in core, a custom authentication backend can provide this implementation and it will be used by *`django.contrib.auth.models.User`*. See the authentication docs for more information.

Permissions for anonymous users If you provide a custom auth backend with `supports_anonymous_user` set to `True`, AnonymousUser will check the backend for permissions, just like User already did. This is useful for centralizing permission handling - apps can always delegate the question of whether something is allowed or not to the authorization/authentication backend. See the authentication docs for more details.

Relaxed requirements for usernames The built-in *`User`* model's *`username`* field now allows a wider range of characters, including `@`, `+`, `.` and − characters.

Email backends You can now *configure the way that Django sends email*. Instead of using SMTP to send all email, you can now choose a configurable email backend to send messages. If your hosting provider uses a sandbox or some other non-SMTP technique for sending mail, you can now construct an email backend that will allow Django's standard mail sending methods to use those facilities.

This also makes it easier to debug mail sending. Django ships with backend implementations that allow you to send email to a *file*, to the *console*, or to *memory*. You can even configure all email to be *thrown away*.

"Smart" `if` tag The *`if`* tag has been upgraded to be much more powerful. First, we've added support for comparison operators. No longer will you have to type:

```
{% ifnotequal a b %}
 ...
{% endifnotequal %}
```

You can now do this:

```
{% if a != b %}
 ...
{% endif %}
```

There's really no reason to use `{% ifequal %}` or `{% ifnotequal %}` anymore, unless you're the nostalgic type.

The operators supported are `==`, `!=`, `<`, `>`, `<=`, `>=`, `in` and `not in`, all of which work like the Python operators, in addition to `and`, `or` and `not`, which were already supported.

Also, filters may now be used in the `if` expression. For example:

```
<div
  {% if user.email|lower == message.recipient|lower %}
    class="highlight"
  {% endif %}
>{{ message }}</div>
```

Template caching In previous versions of Django, every time you rendered a template, it would be reloaded from disk. In Django 1.2, you can use a *cached template loader* to load templates once, then cache the result for every subsequent render. This can lead to a significant performance improvement if your templates are broken into lots of smaller subtemplates (using the `{% extends %}` or `{% include %}` tags).

As a side effect, it is now much easier to support non-Django template languages.

Class-based template loaders As part of the changes made to introduce *Template caching* and following a general trend in Django, the template loaders API has been modified to use template loading mechanisms that are encapsulated in Python classes as opposed to functions, the only method available until Django 1.1.

All the template loaders *shipped with Django* have been ported to the new API but they still implement the function-based API and the template core machinery still accepts function-based loaders (builtin or third party) so there is no immediate need to modify your `TEMPLATE_LOADERS` setting in existing projects, things will keep working if you leave it untouched up to and including the Django 1.3 release.

If you have developed your own custom template loaders we suggest to consider porting them to a class-based implementation because the code for backwards compatibility with function-based loaders starts its deprecation process in Django 1.2 and will be removed in Django 1.4. There is a description of the API these loader classes must implement in the template API reference and you can also examine the source code of the loaders shipped with Django.

Natural keys in fixtures Fixtures can now refer to remote objects using *Natural keys*. This lookup scheme is an alternative to the normal primary-key based object references in a fixture, improving readability and resolving problems referring to objects whose primary key value may not be predictable or known.

Fast failure for tests Both the *`test`* subcommand of `django-admin.py` and the `runtests.py` script used to run Django's own test suite now support a `--failfast` option. When specified, this option causes the test runner to exit after encountering a failure instead of continuing with the test run. In addition, the handling of `Ctrl-C` during a test run has been improved to trigger a graceful exit from the test run that reports details of the tests that were run before the interruption.

`BigIntegerField` Models can now use a 64-bit *`BigIntegerField`* type.

Improved localization Django's internationalization framework has been expanded with locale-aware formatting and form processing. That means, if enabled, dates and numbers on templates will be displayed using the format specified for the current locale. Django will also use localized formats when parsing data in forms. See Format localization for more details.

`readonly_fields` in `ModelAdmin` *`django.contrib.admin.ModelAdmin.readonly_fields`* has been added to enable non-editable fields in add/change pages for models and inlines. Field and calculated values can be displayed alongside editable fields.

Customizable syntax highlighting You can now use a `DJANGO_COLORS` environment variable to modify or disable the colors used by `django-admin.py` to provide *syntax highlighting*.

Syndication feeds as views Syndication feeds can now be used directly as views in your URLconf. This means that you can maintain complete control over the URL structure of your feeds. Like any other view, feeds views are passed a `request` object, so you can do anything you would normally do with a view, like user based access control, or making a feed a named URL.

GeoDjango The most significant new feature for GeoDjango in 1.2 is support for multiple spatial databases. As a result, the following *spatial database backends* are now included:

- `django.contrib.gis.db.backends.postgis`
- `django.contrib.gis.db.backends.mysql`
- `django.contrib.gis.db.backends.oracle`
- `django.contrib.gis.db.backends.spatialite`

GeoDjango now supports the rich capabilities added in the PostGIS 1.5 release. New features include support for the *geography type* and enabling of *distance queries* with non-point geometries on geographic coordinate systems.

Support for 3D geometry fields was added, and may be enabled by setting the *`dim`* keyword to 3 in your *`GeometryField`*. The *`Extent3D`* aggregate and `extent3d() GeoQuerySet` method were added as a part of this feature.

The following *`GeoQuerySet`* methods are new in 1.2:

- *`force_rhr()`*
- *`reverse_geom()`*
- *`geohash()`*

The GEOS interface was updated to use thread-safe C library functions when available on the platform.

The GDAL interface now allows the user to set a *`spatial_filter`* on the features returned when iterating over a *`Layer`*.

Finally, GeoDjango's documentation is now included with Django's and is no longer hosted separately at geodjango.org.

JavaScript-assisted handling of inline related objects in the admin If a user has JavaScript enabled in their browser, the interface for inline objects in the admin now allows inline objects to be dynamically added and removed. Users without JavaScript-enabled browsers will see no change in the behavior of inline objects.

New `now` template tag format specifier characters: `c` and `u` The argument to the *`now`* has gained two new format characters: `c` to specify that a datetime value should be formatted in ISO 8601 format, and `u` that allows output of the microseconds part of a datetime or time value.

These are also available in others parts like the *`date`* and *`time`* template filters, the `humanize` template tag library and the new *format localization* framework.

Backwards-incompatible changes in 1.2

Wherever possible the new features above have been introduced in a backwards-compatible manner per our API stability policy policy. This means that practically all existing code which worked with Django 1.1 will continue to work with Django 1.2; such code will, however, begin issuing warnings (see below for details).

However, a handful of features *have* changed in ways that, for some users, will be immediately backwards-incompatible. Those changes are detailed below.

CSRF Protection We've made large changes to the way CSRF protection works, detailed in the CSRF documentation. Here are the major changes you should be aware of:

- `CsrfResponseMiddleware` and `CsrfMiddleware` have been deprecated and will be removed completely in Django 1.4, in favor of a template tag that should be inserted into forms.
- All contrib apps use a `csrf_protect` decorator to protect the view. This requires the use of the `csrf_token` template tag in the template. If you have used custom templates for contrib views, you MUST READ THE UPGRADE INSTRUCTIONS to fix those templates.

  Documentation removed

  The upgrade notes have been removed in current Django docs. Please refer to the docs for Django 1.3 or older to find these instructions.

- `CsrfViewMiddleware` is included in *`MIDDLEWARE_CLASSES`* by default. This turns on CSRF protection by default, so views that accept POST requests need to be written to work with the middleware. Instructions on how to do this are found in the CSRF docs.
- All of the CSRF has moved from contrib to core (with backwards compatible imports in the old locations, which are deprecated and will cease to be supported in Django 1.4).

`get_db_prep_*()` methods on `Field` Prior to Django 1.2, a custom `Field` had the option of defining several functions to support conversion of Python values into database-compatible values. A custom field might look something like:

```
class CustomModelField(models.Field):
    # ...
    def db_type(self):
        # ...

    def get_db_prep_save(self, value):
        # ...

    def get_db_prep_value(self, value):
        # ...

    def get_db_prep_lookup(self, lookup_type, value):
        # ...
```

In 1.2, these three methods have undergone a change in prototype, and two extra methods have been introduced:

```
class CustomModelField(models.Field):
    # ...

    def db_type(self, connection):
        # ...

    def get_prep_value(self, value):
        # ...

    def get_prep_lookup(self, lookup_type, value):
        # ...

    def get_db_prep_save(self, value, connection):
        # ...

    def get_db_prep_value(self, value, connection, prepared=False):
        # ...

    def get_db_prep_lookup(self, lookup_type, value, connection, prepared=False):
        # ...
```

These changes are required to support multiple databases – `db_type` and `get_db_prep_*` can no longer make any assumptions regarding the database for which it is preparing. The `connection` argument now provides the preparation methods with the specific connection for which the value is being prepared.

The two new methods exist to differentiate general data-preparation requirements from requirements that are database-specific. The `prepared` argument is used to indicate to the database-preparation methods whether generic value preparation has been performed. If an unprepared (i.e., `prepared=False`) value is provided to the `get_db_prep_*()` calls, they should invoke the corresponding `get_prep_*()` calls to perform generic data preparation.

We've provided conversion functions that will transparently convert functions adhering to the old prototype into functions compatible with the new prototype. However, these conversion functions will be removed in Django 1.4, so you should upgrade your `Field` definitions to use the new prototype as soon as possible.

If your `get_db_prep_*()` methods made no use of the database connection, you should be able to upgrade by renaming `get_db_prep_value()` to `get_prep_value()` and `get_db_prep_lookup()` to `get_prep_lookup()`. If you require database specific conversions, then you will need to provide an implementation `get_db_prep_*` that uses the `connection` argument to resolve database-specific values.

Stateful template tags Template tags that store rendering state on their `Node` subclass have always been vulnerable to thread-safety and other issues; as of Django 1.2, however, they may also cause problems when used with the new *cached template loader*.

All of the built-in Django template tags are safe to use with the cached loader, but if you're using custom template tags that come from third party packages, or from your own code, you should ensure that the `Node` implementation for each tag is thread-safe. For more information, see *template tag thread safety considerations*.

You may also need to update your templates if you were relying on the implementation of Django's template tags *not* being thread safe. The *`cycle`* tag is the most likely to be affected in this way, especially when used in conjunction with the *`include`* tag. Consider the following template fragment:

```
{% for object in object_list %}
    {% include "subtemplate.html" %}
{% endfor %}
```

with a `subtemplate.html` that reads:

```
{% cycle 'even' 'odd' %}
```

Using the non-thread-safe, pre-Django 1.2 renderer, this would output:

```
even odd even odd ...
```

Using the thread-safe Django 1.2 renderer, you will instead get:

```
even even even even ...
```

This is because each rendering of the *include* tag is an independent rendering. When the *cycle* tag was not thread safe, the state of the *cycle* tag would leak between multiple renderings of the same *include*. Now that the *cycle* tag is thread safe, this leakage no longer occurs.

user_passes_test, login_required and permission_required `django.contrib.auth.decorators` provides the decorators `login_required`, `permission_required` and `user_passes_test`. Previously it was possible to use these decorators both on functions (where the first argument is 'request') and on methods (where the first argument is 'self', and the second argument is 'request'). Unfortunately, flaws were discovered in the code supporting this: it only works in limited circumstances, and produces errors that are very difficult to debug when it does not work.

For this reason, the 'auto adapt' behavior has been removed, and if you are using these decorators on methods, you will need to manually apply *django.utils.decorators.method_decorator()* to convert the decorator to one that works with methods. For example, you would change code from this:

```
class MyClass(object):

    @login_required
    def my_view(self, request):
        pass
```

to this:

```
from django.utils.decorators import method_decorator

class MyClass(object):

    @method_decorator(login_required)
    def my_view(self, request):
        pass
```

or:

```
from django.utils.decorators import method_decorator

login_required_m = method_decorator(login_required)

class MyClass(object):

    @login_required_m
    def my_view(self, request):
        pass
```

For those of you who've been following the development trunk, this change also applies to other decorators introduced since 1.1, including `csrf_protect`, `cache_control` and anything created using `decorator_from_middleware`.

`if` tag changes Due to new features in the *`if`* template tag, it no longer accepts 'and', 'or' and 'not' as valid **variable** names. Previously, these strings could be used as variable names. Now, the keyword status is always enforced, and template code such as `{% if not %}` or `{% if and %}` will throw a `TemplateSyntaxError`. Also, `in` is a new keyword and so is not a valid variable name in this tag.

`LazyObject` `LazyObject` is an undocumented-but-often-used utility class used for lazily wrapping other objects of unknown type.

In Django 1.1 and earlier, it handled introspection in a non-standard way, depending on wrapped objects implementing a public method named `get_all_members()`. Since this could easily lead to name clashes, it has been changed to use the standard Python introspection method, involving `__members__` and `__dir__()`.

If you used `LazyObject` in your own code and implemented the `get_all_members()` method for wrapped objects, you'll need to make a couple of changes:

First, if your class does not have special requirements for introspection (i.e., you have not implemented `__getattr__()` or other methods that allow for attributes not discoverable by normal mechanisms), you can simply remove the `get_all_members()` method. The default implementation on `LazyObject` will do the right thing.

If you have more complex requirements for introspection, first rename the `get_all_members()` method to `__dir__()`. This is the standard introspection method for Python 2.6 and above. If you require support for Python versions earlier than 2.6, add the following code to the class:

```
__members__ = property(lambda self: self.__dir__())
```

`__dict__` on model instances Historically, the `__dict__` attribute of a model instance has only contained attributes corresponding to the fields on a model.

In order to support multiple database configurations, Django 1.2 has added a `_state` attribute to object instances. This attribute will appear in `__dict__` for a model instance. If your code relies on iterating over `__dict__` to obtain a list of fields, you must now be prepared to handle or filter out the `_state` attribute.

Test runner exit status code The exit status code of the test runners (`tests/runtests.py` and `python manage.py test`) no longer represents the number of failed tests, because a failure of 256 or more tests resulted in a wrong exit status code. The exit status code for the test runner is now 0 for success (no failing tests) and 1 for any number of test failures. If needed, the number of test failures can be found at the end of the test runner's output.

Cookie encoding To fix bugs with cookies in Internet Explorer, Safari, and possibly other browsers, our encoding of cookie values was changed so that the comma and semicolon are treated as non-safe characters, and are therefore encoded as `\054` and `\073` respectively. This could produce backwards incompatibilities, especially if you are storing comma or semi-colon in cookies and have JavaScript code that parses and manipulates cookie values client-side.

`ModelForm.is_valid()` and `ModelForm.errors` Much of the validation work for ModelForms has been moved down to the model level. As a result, the first time you call `ModelForm.is_valid()`, access `ModelForm.errors` or otherwise trigger form validation, your model will be cleaned in-place. This conversion used to happen when the model was saved. If you need an unmodified instance of your model, you should pass a copy to the `ModelForm` constructor.

`BooleanField` on MySQL In previous versions of Django, a model's `BooleanField` under MySQL would return its value as either `1` or `0`, instead of `True` or `False`; for most people this wasn't a problem because `bool` is a subclass of `int` in Python. In Django 1.2, however, `BooleanField` on MySQL correctly returns a real `bool`. The only time this should ever be an issue is if you were expecting the `repr` of a `BooleanField` to print `1` or `0`.

Changes to the interpretation of `max_num` in FormSets As part of enhancements made to the handling of FormSets, the default value and interpretation of the `max_num` parameter to the *django.forms.formsets.formset_factory()* and *django.forms.models.modelformset_factory()* functions has changed slightly. This change also affects the way the `max_num` argument is used for inline admin objects.

Previously, the default value for `max_num` was `0` (zero). FormSets then used the boolean value of `max_num` to determine if a limit was to be imposed on the number of generated forms. The default value of `0` meant that there was no default limit on the number of forms in a FormSet.

Starting with 1.2, the default value for `max_num` has been changed to `None`, and FormSets will differentiate between a value of `None` and a value of `0`. A value of `None` indicates that no limit on the number of forms is to be imposed; a value of `0` indicates that a maximum of 0 forms should be imposed. This doesn't necessarily mean that no forms will be displayed – see the *ModelFormSet documentation* for more details.

If you were manually specifying a value of `0` for `max_num`, you will need to update your FormSet and/or admin definitions.

:

JavaScript-assisted handling of inline related objects in the admin

`email_re` An undocumented regular expression for validating email addresses has been moved from `django.form.fields` to `django.core.validators`. You will need to update your imports if you are using it.

Features deprecated in 1.2

Finally, Django 1.2 deprecates some features from earlier releases. These features are still supported, but will be gradually phased out over the next few release cycles.

Code taking advantage of any of the features below will raise a `PendingDeprecationWarning` in Django 1.2. This warning will be silent by default, but may be turned on using Python's `warnings` module, or by running Python with a `-Wd` or `-Wall` flag.

In Django 1.3, these warnings will become a `DeprecationWarning`, which is *not* silent. In Django 1.4 support for these features will be removed entirely.

:

For more details, see the documentation Django's release process and our deprecation timeline.'

Specifying databases Prior to Django 1.2, Django used a number of settings to control access to a single database. Django 1.2 introduces support for multiple databases, and as a result the way you define database settings has changed.

Any existing Django settings file will continue to work as expected until Django 1.4. Until then, old-style database settings will be automatically translated to the new-style format.

In the old-style (pre 1.2) format, you had a number of `DATABASE_` settings in your settings file. For example:

```
DATABASE_NAME = 'test_db'
DATABASE_ENGINE = 'postgresql_psycopg2'
DATABASE_USER = 'myusername'
DATABASE_PASSWORD = 's3krit'
```

These settings are now in a dictionary named *DATABASES*. Each item in the dictionary corresponds to a single database connection, with the name `'default'` describing the default database connection. The setting names have also been shortened. The previous sample settings would now look like this:

```
DATABASES = {
    'default': {
        'NAME': 'test_db',
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'USER': 'myusername',
        'PASSWORD': 's3krit',
    }
}
```

This affects the following settings:

Old setting	New Setting
DATABASE_ENGINE	*ENGINE*
DATABASE_HOST	*HOST*
DATABASE_NAME	*NAME*
DATABASE_OPTIONS	*OPTIONS*
DATABASE_PASSWORD	*PASSWORD*
DATABASE_PORT	*PORT*
DATABASE_USER	*USER*
TEST_DATABASE_CHARSET	*TEST_CHARSET*
TEST_DATABASE_COLLATION	*TEST_COLLATION*
TEST_DATABASE_NAME	*TEST_NAME*

These changes are also required if you have manually created a database connection using `DatabaseWrapper()` from your database backend of choice.

In addition to the change in structure, Django 1.2 removes the special handling for the built-in database backends. All database backends must now be specified by a fully qualified module name (i.e., `django.db.backends.postgresql_psycopg2`, rather than just `postgresql_psycopg2`).

`postgresql` database backend The `psycopg1` library has not been updated since October 2005. As a result, the `postgresql` database backend, which uses this library, has been deprecated.

If you are currently using the `postgresql` backend, you should migrate to using the `postgresql_psycopg2` backend. To update your code, install the `psycopg2` library and change the *ENGINE* setting to use `django.db.backends.postgresql_psycopg2`.

CSRF response-rewriting middleware `CsrfResponseMiddleware`, the middleware that automatically inserted CSRF tokens into `POST` forms in outgoing pages, has been deprecated in favor of a template tag method (see above), and will be removed completely in Django 1.4. `CsrfMiddleware`, which includes the functionality of `CsrfResponseMiddleware` and `CsrfViewMiddleware`, has likewise been deprecated.

Also, the CSRF module has moved from contrib to core, and the old imports are deprecated, as described in the upgrading notes.

Documentation removed

The upgrade notes have been removed in current Django docs. Please refer to the docs for Django 1.3 or older to find these instructions.

`SMTPConnection` The `SMTPConnection` class has been deprecated in favor of a generic email backend API. Old code that explicitly instantiated an instance of an SMTPConnection:

```
from django.core.mail import SMTPConnection
connection = SMTPConnection()
messages = get_notification_email()
connection.send_messages(messages)
```

...should now call *get_connection()* to instantiate a generic email connection:

```
from django.core.mail import get_connection
connection = get_connection()
messages = get_notification_email()
connection.send_messages(messages)
```

Depending on the value of the *EMAIL_BACKEND* setting, this may not return an SMTP connection. If you explicitly require an SMTP connection with which to send email, you can explicitly request an SMTP connection:

```
from django.core.mail import get_connection
connection = get_connection('django.core.mail.backends.smtp.EmailBackend')
messages = get_notification_email()
connection.send_messages(messages)
```

If your call to construct an instance of `SMTPConnection` required additional arguments, those arguments can be passed to the *get_connection()* call:

```
connection = get_connection('django.core.mail.backends.smtp.EmailBackend', hostname='localhost', port
```

User Messages API The API for storing messages in the user `Message` model (via `user.message_set.create`) is now deprecated and will be removed in Django 1.4 according to the standard release process.

To upgrade your code, you need to replace any instances of this:

```
user.message_set.create('a message')
```

...with the following:

```
from django.contrib import messages
messages.add_message(request, messages.INFO, 'a message')
```

Additionally, if you make use of the method, you need to replace the following:

```
for message in user.get_and_delete_messages():
    ...
```

...with:

```
from django.contrib import messages
for message in messages.get_messages(request):
    ...
```

For more information, see the full messages documentation. You should begin to update your code to use the new API immediately.

Date format helper functions `django.utils.translation.get_date_formats()` and `django.utils.translation.get_partial_date_formats()` have been deprecated in favor of the appropriate calls to `django.utils.formats.get_format()`, which is locale-aware when *USE_L10N* is set to `True`, and falls back to default settings if set to `False`.

To get the different date formats, instead of writing this:

```
from django.utils.translation import get_date_formats
date_format, datetime_format, time_format = get_date_formats()
```

...use:

```
from django.utils import formats
date_format = formats.get_format('DATE_FORMAT')
datetime_format = formats.get_format('DATETIME_FORMAT')
time_format = formats.get_format('TIME_FORMAT')
```

Or, when directly formatting a date value:

```
from django.utils import formats
value_formatted = formats.date_format(value, 'DATETIME_FORMAT')
```

The same applies to the globals found in `django.forms.fields`:

- `DEFAULT_DATE_INPUT_FORMATS`
- `DEFAULT_TIME_INPUT_FORMATS`
- `DEFAULT_DATETIME_INPUT_FORMATS`

Use `django.utils.formats.get_format()` to get the appropriate formats.

Function-based test runners Django 1.2 changes the test runner tools to use a class-based approach. Old style function-based test runners will still work, but should be updated to use the new *class-based runners*.

`Feed` in `django.contrib.syndication.feeds` The `django.contrib.syndication.feeds.Feed` class has been replaced by the *`django.contrib.syndication.views.Feed`* class. The old `feeds.Feed` class is deprecated, and will be removed in Django 1.4.

The new class has an almost identical API, but allows instances to be used as views. For example, consider the use of the old framework in the following URLconf:

```
from django.conf.urls.defaults import *
from myproject.feeds import LatestEntries, LatestEntriesByCategory

feeds = {
    'latest': LatestEntries,
    'categories': LatestEntriesByCategory,
}

urlpatterns = patterns('',
    # ...
    (r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
        {'feed_dict': feeds}),
    # ...
)
```

Using the new Feed class, these feeds can be deployed directly as views:

```
from django.conf.urls.defaults import *
from myproject.feeds import LatestEntries, LatestEntriesByCategory

urlpatterns = patterns('',
    # ...
    (r'^feeds/latest/$', LatestEntries()),
    (r'^feeds/categories/(?P<category_id>\d+)/$', LatestEntriesByCategory()),
```

```
    # ...
)
```

If you currently use the `feed()` view, the `LatestEntries` class would often not need to be modified apart from subclassing the new *`Feed`* class. The exception is if Django was automatically working out the name of the template to use to render the feed's description and title elements (if you were not specifying the `title_template` and `description_template` attributes). You should ensure that you always specify `title_template` and `description_template` attributes, or provide `item_title()` and `item_description()` methods.

However, `LatestEntriesByCategory` uses the `get_object()` method with the `bits` argument to specify a specific category to show. In the new *`Feed`* class, `get_object()` method takes a `request` and arguments from the URL, so it would look like this:

```
from django.contrib.syndication.views import Feed
from django.shortcuts import get_object_or_404
from myproject.models import Category

class LatestEntriesByCategory(Feed):
    def get_object(self, request, category_id):
        return get_object_or_404(Category, id=category_id)

    # ...
```

Additionally, the `get_feed()` method on `Feed` classes now take different arguments, which may impact you if you use the `Feed` classes directly. Instead of just taking an optional `url` argument, it now takes two arguments: the object returned by its own `get_object()` method, and the current `request` object.

To take into account `Feed` classes not being initialized for each request, the `__init__()` method now takes no arguments by default. Previously it would have taken the `slug` from the URL and the `request` object.

In accordance with RSS best practices, RSS feeds will now include an `atom:link` element. You may need to update your tests to take this into account.

For more information, see the full syndication framework documentation.

Technical message IDs Up to version 1.1 Django used technical message IDs to provide localizers the possibility to translate date and time formats. They were translatable *translation strings* that could be recognized because they were all upper case (for example *`DATETIME_FORMAT`*, *`DATE_FORMAT`*, *`TIME_FORMAT`*). They have been deprecated in favor of the new Format localization infrastructure that allows localizers to specify that information in a `formats.py` file in the corresponding `django/conf/locale/<locale name>/` directory.

GeoDjango To allow support for multiple databases, the GeoDjango database internals were changed substantially. The largest backwards-incompatible change is that the module `django.contrib.gis.db.backend` was renamed to *`django.contrib.gis.db.backends`*, where the full-fledged *spatial database backends* now exist. The following sections provide information on the most-popular APIs that were affected by these changes.

`SpatialBackend` Prior to the creation of the separate spatial backends, the `django.contrib.gis.db.backend.SpatialBackend` object was provided as an abstraction to introspect on the capabilities of the spatial database. All of the attributes and routines provided by `SpatialBackend` are now a part of the `ops` attribute of the database backend.

The old module `django.contrib.gis.db.backend` is still provided for backwards-compatibility access to a `SpatialBackend` object, which is just an alias to the `ops` module of the *default* spatial database connection.

Users that were relying on undocumented modules and objects within `django.contrib.gis.db.backend`, rather the abstractions provided by `SpatialBackend`, are required to modify their code. For example, the following import which would work in 1.1 and below:

```
from django.contrib.gis.db.backend.postgis import PostGISAdaptor
```

Would need to be changed:

```
from django.db import connection
PostGISAdaptor = connection.ops.Adapter
```

`SpatialRefSys` and `GeometryColumns` models In previous versions of GeoDjango, *`django.contrib.gis.db.models`* had `SpatialRefSys` and `GeometryColumns` models for querying the OGC spatial metadata tables `spatial_ref_sys` and `geometry_columns`, respectively.

While these aliases are still provided, they are only for the *default* database connection and exist only if the default connection is using a supported spatial database backend.

: Because the table structure of the OGC spatial metadata tables differs across spatial databases, the `SpatialRefSys` and `GeometryColumns` models can no longer be associated with the `gis` application name. Thus, no models will be returned when using the `get_models` method in the following example:

```
>>> from django.db.models import get_app, get_models
>>> get_models(get_app('gis'))
[]
```

To get the correct `SpatialRefSys` and `GeometryColumns` for your spatial database use the methods provided by the spatial backend:

```
>>> from django.db import connections
>>> SpatialRefSys = connections['my_spatialite'].ops.spatial_ref_sys()
>>> GeometryColumns = connections['my_postgis'].ops.geometry_columns()
```

: When using the models returned from the `spatial_ref_sys()` and `geometry_columns()` method, you'll still need to use the correct database alias when querying on the non-default connection. In other words, to ensure that the models in the example above use the correct database:

```
sr_qs = SpatialRefSys.objects.using('my_spatialite').filter(...)
gc_qs = GeometryColumns.objects.using('my_postgis').filter(...)
```

Language code `no` The currently used language code for Norwegian Bokmål `no` is being replaced by the more common language code `nb`.

Function-based template loaders Django 1.2 changes the template loading mechanism to use a class-based approach. Old style function-based template loaders will still work, but should be updated to use the new class-based template loaders.

9.1.10 1.1 release

Django 1.1.4 release notes

Welcome to Django 1.1.4!

This is the fourth "bugfix" release in the Django 1.1 series, improving the stability and performance of the Django 1.1 codebase.

With one exception, Django 1.1.4 maintains backwards compatibility with Django 1.1.3. It also contains a number of fixes and other improvements. Django 1.1.4 is a recommended upgrade for any development or deployment currently using or targeting Django 1.1.

For full details on the new features, backwards incompatibilities, and deprecated features in the 1.1 branch, see the Django 1.1 release notes.

Backwards incompatible changes

CSRF exception for AJAX requests Django includes a CSRF-protection mechanism, which makes use of a token inserted into outgoing forms. Middleware then checks for the token's presence on form submission, and validates it.

Prior to Django 1.2.5, our CSRF protection made an exception for AJAX requests, on the following basis:

- Many AJAX toolkits add an X-Requested-With header when using XMLHttpRequest.
- Browsers have strict same-origin policies regarding XMLHttpRequest.
- In the context of a browser, the only way that a custom header of this nature can be added is with XMLHttpRequest.

Therefore, for ease of use, we did not apply CSRF checks to requests that appeared to be AJAX on the basis of the X-Requested-With header. The Ruby on Rails web framework had a similar exemption.

Recently, engineers at Google made members of the Ruby on Rails development team aware of a combination of browser plugins and redirects which can allow an attacker to provide custom HTTP headers on a request to any website. This can allow a forged request to appear to be an AJAX request, thereby defeating CSRF protection which trusts the same-origin nature of AJAX requests.

Michael Koziarski of the Rails team brought this to our attention, and we were able to produce a proof-of-concept demonstrating the same vulnerability in Django's CSRF handling.

To remedy this, Django will now apply full CSRF validation to all requests, regardless of apparent AJAX origin. This is technically backwards-incompatible, but the security risks have been judged to outweigh the compatibility concerns in this case.

Additionally, Django will now accept the CSRF token in the custom HTTP header X-CSRFTOKEN, as well as in the form submission itself, for ease of use with popular JavaScript toolkits which allow insertion of custom headers into all AJAX requests.

Please see the *CSRF docs for example jQuery code* that demonstrates this technique, ensuring that you are looking at the documentation for your version of Django, as the exact code necessary is different for some older versions of Django.

Django 1.1.3 release notes

Welcome to Django 1.1.3!

This is the third "bugfix" release in the Django 1.1 series, improving the stability and performance of the Django 1.1 codebase.

With one exception, Django 1.1.3 maintains backwards compatibility with Django 1.1.2. It also contains a number of fixes and other improvements. Django 1.1.2 is a recommended upgrade for any development or deployment currently using or targeting Django 1.1.

For full details on the new features, backwards incompatibilities, and deprecated features in the 1.1 branch, see the Django 1.1 release notes.

Backwards incompatible changes

Restricted filters in admin interface The Django administrative interface, django.contrib.admin, supports filtering of displayed lists of objects by fields on the corresponding models, including across database-level relationships. This is implemented by passing lookup arguments in the querystring portion of the URL, and options on the ModelAdmin class allow developers to specify particular fields or relationships which will generate automatic links for filtering.

One historically-undocumented and -unofficially-supported feature has been the ability for a user with sufficient knowledge of a model's structure and the format of these lookup arguments to invent useful new filters on the fly by manipulating the querystring.

However, it has been demonstrated that this can be abused to gain access to information outside of an admin user's permissions; for example, an attacker with access to the admin and sufficient knowledge of model structure and relations could construct query strings which – with repeated use of regular-expression lookups supported by the Django database API – expose sensitive information such as users' password hashes.

To remedy this, django.contrib.admin will now validate that querystring lookup arguments either specify only fields on the model being viewed, or cross relations which have been explicitly whitelisted by the application developer using the pre-existing mechanism mentioned above. This is backwards-incompatible for any users relying on the prior ability to insert arbitrary lookups.

Django 1.1.2 release notes

Welcome to Django 1.1.2!

This is the second "bugfix" release in the Django 1.1 series, improving the stability and performance of the Django 1.1 codebase.

Django 1.1.2 maintains backwards compatibility with Django 1.1.0, but contain a number of fixes and other improvements. Django 1.1.2 is a recommended upgrade for any development or deployment currently using or targeting Django 1.1.

For full details on the new features, backwards incompatibilities, and deprecated features in the 1.1 branch, see the Django 1.1 release notes.

Backwards-incompatible changes in 1.1.2

Test runner exit status code The exit status code of the test runners (`tests/runtests.py` and `python manage.py test`) no longer represents the number of failed tests, since a failure of 256 or more tests resulted in a wrong exit status code. The exit status code for the test runner is now 0 for success (no failing tests) and 1 for any number of test failures. If needed, the number of test failures can be found at the end of the test runner's output.

Cookie encoding To fix bugs with cookies in Internet Explorer, Safari, and possibly other browsers, our encoding of cookie values was changed so that the characters comma and semi-colon are treated as non-safe characters, and are therefore encoded as `\054` and `\073` respectively. This could produce backwards incompatibilities, especially if you are storing comma or semi-colon in cookies and have JavaScript code that parses and manipulates cookie values client-side.

One new feature

Ordinarily, a point release would not include new features, but in the case of Django 1.1.2, we have made an exception to this rule. Django 1.2 (the next major release of Django) will contain a feature that will improve protection against Cross-Site Request Forgery (CSRF) attacks. This feature requires the use of a new `csrf_token` template tag in all forms that Django renders.

To make it easier to support both 1.1.X and 1.2.X versions of Django with the same templates, we have decided to introduce the `csrf_token` template tag to the 1.1.X branch. In the 1.1.X branch, `csrf_token` does nothing - it has no effect on templates or form processing. However, it means that the same template will work with Django 1.2.

Django 1.1 release notes

July 29, 2009

Welcome to Django 1.1!

Django 1.1 includes a number of nifty *new features*, lots of bug fixes, and an easy upgrade path from Django 1.0.

Backwards-incompatible changes in 1.1

Django has a policy of API stability. This means that, in general, code you develop against Django 1.0 should continue to work against 1.1 unchanged. However, we do sometimes make backwards-incompatible changes if they're necessary to resolve bugs, and there are a handful of such (minor) changes between Django 1.0 and Django 1.1.

Before upgrading to Django 1.1 you should double-check that the following changes don't impact you, and upgrade your code if they do.

Changes to constraint names Django 1.1 modifies the method used to generate database constraint names so that names are consistent regardless of machine word size. This change is backwards incompatible for some users.

If you are using a 32-bit platform, you're off the hook; you'll observe no differences as a result of this change.

However, **users on 64-bit platforms may experience some problems** using the `reset` management command. Prior to this change, 64-bit platforms would generate a 64-bit, 16 character digest in the constraint name; for example:

```
ALTER TABLE myapp_sometable ADD CONSTRAINT object_id_refs_id_5e8f10c132091d1e FOREIGN KEY ...
```

Following this change, all platforms, regardless of word size, will generate a 32-bit, 8 character digest in the constraint name; for example:

```
ALTER TABLE myapp_sometable ADD CONSTRAINT object_id_refs_id_32091d1e FOREIGN KEY ...
```

As a result of this change, you will not be able to use the `reset` management command on any table made by a 64-bit machine. This is because the new generated name will not match the historically generated name; as a result, the SQL constructed by the reset command will be invalid.

If you need to reset an application that was created with 64-bit constraints, you will need to manually drop the old constraint prior to invoking `reset`.

Test cases are now run in a transaction Django 1.1 runs tests inside a transaction, allowing better test performance (see *test performance improvements* for details).

This change is slightly backwards incompatible if existing tests need to test transactional behavior, if they rely on invalid assumptions about the test environment, or if they require a specific test case ordering.

For these cases, *TransactionTestCase* can be used instead. This is a just a quick fix to get around test case errors revealed by the new rollback approach; in the long-term tests should be rewritten to correct the test case.

Removed `SetRemoteAddrFromForwardedFor` middleware For convenience, Django 1.0 included an optional middleware class – `django.middleware.http.SetRemoteAddrFromForwardedFor` – which updated the value of `REMOTE_ADDR` based on the HTTP `X-Forwarded-For` header commonly set by some proxy configurations.

It has been demonstrated that this mechanism cannot be made reliable enough for general-purpose use, and that (despite documentation to the contrary) its inclusion in Django may lead application developers to assume that the value of `REMOTE_ADDR` is "safe" or in some way reliable as a source of authentication.

While not directly a security issue, we've decided to remove this middleware with the Django 1.1 release. It has been replaced with a class that does nothing other than raise a `DeprecationWarning`.

If you've been relying on this middleware, the easiest upgrade path is:

- Examine the code as it existed before it was removed.
- Verify that it works correctly with your upstream proxy, modifying it to support your particular proxy (if necessary).
- Introduce your modified version of `SetRemoteAddrFromForwardedFor` as a piece of middleware in your own project.

Names of uploaded files are available later In Django 1.0, files uploaded and stored in a model's *FileField* were saved to disk before the model was saved to the database. This meant that the actual file name assigned to the file was available before saving. For example, it was available in a model's pre-save signal handler.

In Django 1.1 the file is saved as part of saving the model in the database, so the actual file name used on disk cannot be relied on until *after* the model has been saved.

Changes to how model formsets are saved In Django 1.1, *BaseModelFormSet* now calls `ModelForm.save()`.

This is backwards-incompatible if you were modifying `self.initial` in a model formset's `__init__`, or if you relied on the internal `_total_form_count` or `_initial_form_count` attributes of BaseFormSet. Those attributes are now public methods.

Fixed the `join` filter's escaping behavior The *join* filter no longer escapes the literal value that is passed in for the connector.

This is backwards incompatible for the special situation of the literal string containing one of the five special HTML characters. Thus, if you were writing `{{ foo|join:"&" }}`, you now have to write `{{ foo|join:"&" }}`.

The previous behavior was a bug and contrary to what was documented and expected.

Permanent redirects and the `redirect_to()` generic view Django 1.1 adds a `permanent` argument to the `django.views.generic.simple.redirect_to()` view. This is technically backwards-incompatible if you were using the `redirect_to` view with a format-string key called 'permanent', which is highly unlikely.

Features deprecated in 1.1

One feature has been marked as deprecated in Django 1.1:

- You should no longer use `AdminSite.root()` to register that admin views. That is, if your URLconf contains the line:

```
(r'^admin/(.*)', admin.site.root),
```

  You should change it to read:

```
(r'^admin/', include(admin.site.urls)),
```

You should begin to remove use of this feature from your code immediately.

`AdminSite.root` will raise a `PendingDeprecationWarning` if used in Django 1.1. This warning is hidden by default. In Django 1.2, this warning will be upgraded to a `DeprecationWarning`, which will be displayed loudly. Django 1.3 will remove `AdminSite.root()` entirely.

For more details on our deprecation policies and strategy, see Django's release process.

What's new in Django 1.1

Quite a bit: since Django 1.0, we've made 1,290 code commits, fixed 1,206 bugs, and added roughly 10,000 lines of documentation.

The major new features in Django 1.1 are:

ORM improvements Two major enhancements have been added to Django's object-relational mapper (ORM): aggregate support, and query expressions.

Aggregate support It's now possible to run SQL aggregate queries (i.e. `COUNT()`, `MAX()`, `MIN()`, etc.) from within Django's ORM. You can choose to either return the results of the aggregate directly, or else annotate the objects in a *`QuerySet`* with the results of the aggregate query.

This feature is available as new *`aggregate()`* and *`annotate()`* methods, and is covered in detail in the ORM aggregation documentation.

Query expressions Queries can now refer to a another field on the query and can traverse relationships to refer to fields on related models. This is implemented in the new *`F`* object; for full details, including examples, consult the *`F expressions documentation`*.

Model improvements A number of features have been added to Django's model layer:

"Unmanaged" models You can now control whether or not Django manages the life-cycle of the database tables for a model using the *`managed`* model option. This defaults to `True`, meaning that Django will create the appropriate database tables in `syncdb` and remove them as part of the `reset` command. That is, Django *manages* the database table's lifecycle.

If you set this to `False`, however, no database table creating or deletion will be automatically performed for this model. This is useful if the model represents an existing table or a database view that has been created by some other means.

For more details, see the documentation for the *`managed`* option.

Proxy models You can now create *proxy models*: subclasses of existing models that only add Python-level (rather than database-level) behavior and aren't represented by a new table. That is, the new model is a *proxy* for some underlying model, which stores all the real data.

All the details can be found in the *proxy models documentation*. This feature is similar on the surface to unmanaged models, so the documentation has an explanation of *how proxy models differ from unmanaged models*.

Deferred fields In some complex situations, your models might contain fields which could contain a lot of data (for example, large text fields), or require expensive processing to convert them to Python objects. If you know you don't need those particular fields, you can now tell Django not to retrieve them from the database.

You'll do this with the new queryset methods `defer()` and `only()`.

Testing improvements A few notable improvements have been made to the testing framework.

Test performance improvements Tests written using Django's testing framework now run dramatically faster (as much as 10 times faster in many cases).

This was accomplished through the introduction of transaction-based tests: when using `django.test.TestCase`, your tests will now be run in a transaction which is rolled back when finished, instead of by flushing and re-populating the database. This results in an immense speedup for most types of unit tests. See the documentation for `TestCase` and `TransactionTestCase` for a full description, and some important notes on database support.

Test client improvements A couple of small – but highly useful – improvements have been made to the test client:

- The test `Client` now can automatically follow redirects with the `follow` argument to `Client.get()` and `Client.post()`. This makes testing views that issue redirects simpler.
- It's now easier to get at the template context in the response returned the test client: you'll simply access the context as `request.context[key]`. The old way, which treats `request.context` as a list of contexts, one for each rendered template in the inheritance chain, is still available if you need it.

New admin features Django 1.1 adds a couple of nifty new features to Django's admin interface:

Editable fields on the change list You can now make fields editable on the admin list views via the new *list_editable* admin option. These fields will show up as form widgets on the list pages, and can be edited and saved in bulk.

Admin "actions" You can now define admin actions that can perform some action to a group of models in bulk. Users will be able to select objects on the change list page and then apply these bulk actions to all selected objects.

Django ships with one pre-defined admin action to delete a group of objects in one fell swoop.

Conditional view processing Django now has much better support for conditional view processing using the standard `ETag` and `Last-Modified` HTTP headers. This means you can now easily short-circuit view processing by testing less-expensive conditions. For many views this can lead to a serious improvement in speed and reduction in bandwidth.

URL namespaces Django 1.1 improves *named URL patterns* with the introduction of URL "namespaces."

In short, this feature allows the same group of URLs, from the same application, to be included in a Django URLConf multiple times, with varying (and potentially nested) named prefixes which will be used when performing reverse resolution. In other words, reusable applications like Django's admin interface may be registered multiple times without URL conflicts.

For full details, see *the documentation on defining URL namespaces*.

GeoDjango In Django 1.1, GeoDjango (i.e. `django.contrib.gis`) has several new features:

- Support for SpatiaLite – a spatial database for SQLite – as a spatial backend.
- Geographic aggregates (`Collect`, `Extent`, `MakeLine`, `Union`) and `F` expressions.
- New `GeoQuerySet` methods: `collect`, `geojson`, and `snap_to_grid`.
- A new list interface methods for `GEOSGeometry` objects.

For more details, see the GeoDjango documentation.

Other improvements Other new features and changes introduced since Django 1.0 include:

- The CSRF protection middleware has been split into two classes – `CsrfViewMiddleware` checks incoming requests, and `CsrfResponseMiddleware` processes outgoing responses. The combined `CsrfMiddleware` class (which does both) remains for backwards-compatibility, but using the split classes is now recommended in order to allow fine-grained control of when and where the CSRF processing takes place.
- `reverse()` and code which uses it (e.g., the `{% url %}` template tag) now works with URLs in Django's administrative site, provided that the admin URLs are set up via `include(admin.site.urls)` (sending admin requests to the `admin.site.root` view still works, but URLs in the admin will not be "reversible" when configured this way).
- The `include()` function in Django URLconf modules can now accept sequences of URL patterns (generated by `patterns()`) in addition to module names.
- Instances of Django forms (see the forms overview) now have two additional methods, `hidden_fields()` and `visible_fields()`, which return the list of hidden – i.e., `<input type="hidden">` – and visible fields on the form, respectively.
- The `redirect_to` generic view now accepts an additional keyword argument `permanent`. If `permanent` is `True`, the view will emit an HTTP permanent redirect (status code 301). If `False`, the view will emit an HTTP temporary redirect (status code 302).
- A new database lookup type – `week_day` – has been added for `DateField` and `DateTimeField`. This type of lookup accepts a number between 1 (Sunday) and 7 (Saturday), and returns objects where the field value matches that day of the week. See *the full list of lookup types* for details.
- The `{% for %}` tag in Django's template language now accepts an optional `{% empty %}` clause, to be displayed when `{% for %}` is asked to loop over an empty sequence. See the list of built-in template tags for examples of this.
- The *`dumpdata`* management command now accepts individual model names as arguments, allowing you to export the data just from particular models.
- There's a new *`safeseq`* template filter which works just like *`safe`* for lists, marking each item in the list as safe.
- Cache backends now support `incr()` and `decr()` commands to increment and decrement the value of a cache key. On cache backends that support atomic increment/decrement – most notably, the memcached backend – these operations will be atomic, and quite fast.

- Django now can easily delegate authentication to the Web server via a new authentication backend that supports the standard `REMOTE_USER` environment variable used for this purpose.
- There's a new *`django.shortcuts.redirect()`* function that makes it easier to issue redirects given an object, a view name, or a URL.
- The `postgresql_psycopg2` backend now supports *native PostgreSQL autocommit*. This is an advanced, PostgreSQL-specific feature, that can make certain read-heavy applications a good deal faster.

What's next?

We'll take a short break, and then work on Django 1.2 will begin – no rest for the weary! If you'd like to help, discussion of Django development, including progress toward the 1.2 release, takes place daily on the django-developers mailing list:

- https://groups.google.com/group/django-developers

... and in the `#django-dev` IRC channel on `irc.freenode.net`. Feel free to join the discussions!

Django's online documentation also includes pointers on how to contribute to Django:

- How to contribute to Django

Contributions on any level – developing code, writing documentation or simply triaging tickets and helping to test proposed bugfixes – are always welcome and appreciated.

And that's the way it is.

9.1.11 1.0 release

Django 1.0.2 release notes

Welcome to Django 1.0.2!

This is the second "bugfix" release in the Django 1.0 series, improving the stability and performance of the Django 1.0 codebase. As such, Django 1.0.2 contains no new features (and, pursuant to our compatibility policy, maintains backwards compatibility with Django 1.0.0), but does contain a number of fixes and other improvements. Django 1.0.2 is a recommended upgrade for any development or deployment currently using or targeting Django 1.0.

Fixes and improvements in Django 1.0.2

The primary reason behind this release is to remedy an issue in the recently-released Django 1.0.1; the packaging scripts used for Django 1.0.1 omitted some directories from the final release package, including one directory required by `django.contrib.gis` and part of Django's unit-test suite.

Django 1.0.2 contains updated packaging scripts, and the release package contains the directories omitted from Django 1.0.1. As such, this release contains all of the fixes and improvements from Django 1.0.1; see the Django 1.0.1 release notes for details.

Additionally, in the period since Django 1.0.1 was released:

- Updated Hebrew and Danish translations have been added.
- The default `__repr__` method of Django models has been made more robust in the face of bad Unicode data coming from the `__unicode__` method; rather than raise an exception in such cases, `repr()` will now contain the string "[Bad Unicode data]" in place of the invalid Unicode.

- A bug involving the interaction of Django's `SafeUnicode` class and the MySQL adapter has been resolved; `SafeUnicode` instances (generated, for example, by template rendering) can now be assigned to model attributes and saved to MySQL without requiring an explicit intermediate cast to `unicode`.
- A bug affecting filtering on a nullable `DateField` in SQLite has been resolved.
- Several updates and improvements have been made to Django's documentation.

Django 1.0.1 release notes

Welcome to Django 1.0.1!

This is the first "bugfix" release in the Django 1.0 series, improving the stability and performance of the Django 1.0 codebase. As such, Django 1.0.1 contains no new features (and, pursuant to our compatibility policy, maintains backwards compatibility with Django 1.0), but does contain a number of fixes and other improvements. Django 1.0.1 is a recommended upgrade for any development or deployment currently using or targeting Django 1.0.

Fixes and improvements in Django 1.0.1

Django 1.0.1 contains over two hundred fixes to the original Django 1.0 codebase; full details of every fix are available in the history of the 1.0.X branch, but here are some of the highlights:

- Several fixes in `django.contrib.comments`, pertaining to RSS feeds of comments, default ordering of comments and the XHTML and internationalization of the default templates for comments.
- Multiple fixes for Django's support of Oracle databases, including pagination support for GIS QuerySets, more efficient slicing of results and improved introspection of existing databases.
- Several fixes for query support in the Django object-relational mapper, including repeated setting and resetting of ordering and fixes for working with `INSERT`-only queries.
- Multiple fixes for inline forms in formsets.
- Multiple fixes for `unique` and `unique_together` model constraints in automatically-generated forms.
- Fixed support for custom callable `upload_to` declarations when handling file uploads through automatically-generated forms.
- Fixed support for sorting an admin change list based on a callable attributes in `list_display`.
- A fix to the application of autoescaping for literal strings passed to the `join` template filter. Previously, literal strings passed to `join` were automatically escaped, contrary to *the documented behavior for autoescaping and literal strings*. Literal strings passed to `join` are no longer automatically escaped, meaning you must now manually escape them; this is an incompatibility if you were relying on this bug, but not if you were relying on escaping behaving as documented.
- Improved and expanded translation files for many of the languages Django supports by default.
- And as always, a large number of improvements to Django's documentation, including both corrections to existing documents and expanded and new documentation.

Django 1.0 release notes

Welcome to Django 1.0!

We've been looking forward to this moment for over three years, and it's finally here. Django 1.0 represents the largest milestone in Django's development to date: a Web framework that a group of perfectionists can truly be proud of.

Django 1.0 represents over three years of community development as an Open Source project. Django's received contributions from hundreds of developers, been translated into fifty languages, and today is used by developers on every continent and in every kind of job.

An interesting historical note: when Django was first released in July 2005, the initial released version of Django came from an internal repository at revision number 8825. Django 1.0 represents revision 8961 of our public repository. It seems fitting that our 1.0 release comes at the moment where community contributions overtake those made privately.

Stability and forwards-compatibility

The release of Django 1.0 comes with a promise of API stability and forwards-compatibility. In a nutshell, this means that code you develop against Django 1.0 will continue to work against 1.1 unchanged, and you should need to make only minor changes for any 1.X release.

See the API stability guide for full details.

Backwards-incompatible changes

Django 1.0 has a number of backwards-incompatible changes from Django 0.96. If you have apps written against Django 0.96 that you need to port, see our detailed porting guide:

Porting your apps from Django 0.96 to 1.0 Django 1.0 breaks compatibility with 0.96 in some areas.

This guide will help you port 0.96 projects and apps to 1.0. The first part of this document includes the common changes needed to run with 1.0. If after going through the first part your code still breaks, check the section *Less-common Changes* for a list of a bunch of less-common compatibility issues.

:

The 1.0 release notes. That document explains the new features in 1.0 more deeply; the porting guide is more concerned with helping you quickly update your code.

Common changes This section describes the changes between 0.96 and 1.0 that most users will need to make.

Use Unicode Change string literals (`'foo'`) into Unicode literals (`u'foo'`). Django now uses Unicode strings throughout. In most places, raw strings will continue to work, but updating to use Unicode literals will prevent some obscure problems.

See Unicode data for full details.

Models Common changes to your models file:

Rename `maxlength` to `max_length` Rename your `maxlength` argument to `max_length` (this was changed to be consistent with form fields):

Replace `__str__` with `__unicode__` Replace your model's `__str__` function with a `__unicode__` method, and make sure you *use Unicode* (`u'foo'`) in that method.

Remove `prepopulated_from` Remove the `prepopulated_from` argument on model fields. It's no longer valid and has been moved to the `ModelAdmin` class in `admin.py`. See *the admin*, below, for more details about changes to the admin.

Remove `core` Remove the `core` argument from your model fields. It is no longer necessary, since the equivalent functionality (part of *inline editing*) is handled differently by the admin interface now. You don't have to worry about inline editing until you get to *the admin* section, below. For now, remove all references to `core`.

Replace `class Admin:` with `admin.py` Remove all your inner `class Admin` declarations from your models. They won't break anything if you leave them, but they also won't do anything. To register apps with the admin you'll move those declarations to an `admin.py` file; see *the admin* below for more details.

:

A contributor to djangosnippets has written a script that'll scan your models.py and generate a corresponding admin.py.

Example Below is an example `models.py` file with all the changes you'll need to make:

Old (0.96) `models.py`:

```
class Author(models.Model):
    first_name = models.CharField(maxlength=30)
    last_name = models.CharField(maxlength=30)
    slug = models.CharField(maxlength=60, prepopulate_from=('first_name', 'last_name'))

    class Admin:
        list_display = ['first_name', 'last_name']

    def __str__(self):
        return '%s %s' % (self.first_name, self.last_name)
```

New (1.0) `models.py`:

```
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    slug = models.CharField(max_length=60)

    def __unicode__(self):
        return u'%s %s' % (self.first_name, self.last_name)
```

New (1.0) `admin.py`:

```
from django.contrib import admin
from models import Author

class AuthorAdmin(admin.ModelAdmin):
    list_display = ['first_name', 'last_name']
    prepopulated_fields = {
        'slug': ('first_name', 'last_name')
    }

admin.site.register(Author, AuthorAdmin)
```

The Admin One of the biggest changes in 1.0 is the new admin. The Django administrative interface (`django.contrib.admin`) has been completely refactored; admin definitions are now completely decoupled from model definitions, the framework has been rewritten to use Django's new form-handling library and redesigned with extensibility and customization in mind.

Practically, this means you'll need to rewrite all of your `class Admin` declarations. You've already seen in *models* above how to replace your `class Admin` with a `admin.site.register()` call in an `admin.py` file. Below are some more details on how to rewrite that `Admin` declaration into the new syntax.

Use new inline syntax The new `edit_inline` options have all been moved to `admin.py`. Here's an example:

Old (0.96):

```
class Parent(models.Model):
    ...

class Child(models.Model):
    parent = models.ForeignKey(Parent, edit_inline=models.STACKED, num_in_admin=3)
```

New (1.0):

```
class ChildInline(admin.StackedInline):
    model = Child
    extra = 3

class ParentAdmin(admin.ModelAdmin):
    model = Parent
    inlines = [ChildInline]

admin.site.register(Parent, ParentAdmin)
```

See *InlineModelAdmin objects* for more details.

Simplify `fields`, or use `fieldsets` The old `fields` syntax was quite confusing, and has been simplified. The old syntax still works, but you'll need to use `fieldsets` instead.

Old (0.96):

```
class ModelOne(models.Model):
    ...

    class Admin:
        fields = (
            (None, {'fields': ('foo','bar')}),
        )

class ModelTwo(models.Model):
    ...

    class Admin:
        fields = (
            ('group1', {'fields': ('foo','bar'),   'classes': 'collapse'}),
            ('group2', {'fields': ('spam','eggs'), 'classes': 'collapse wide'}),
        )
```

New (1.0):

```
class ModelOneAdmin(admin.ModelAdmin):
    fields = ('foo', 'bar')

class ModelTwoAdmin(admin.ModelAdmin):
    fieldsets = (
        ('group1', {'fields': ('foo','bar'),   'classes': 'collapse'}),
        ('group2', {'fields': ('spam','eggs'), 'classes': 'collapse wide'}),
    )
```

:

- More detailed information about the changes and the reasons behind them can be found on the NewformsAdminBranch wiki page
- The new admin comes with a ton of new features; you can read about them in the admin documentation.

URLs

Update your root `urls.py` If you're using the admin site, you need to update your root `urls.py`.

Old (0.96) `urls.py`:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^admin/', include('django.contrib.admin.urls')),

    # ... the rest of your URLs here ...
)
```

New (1.0) `urls.py`:

```
from django.conf.urls.defaults import *

# The next two lines enable the admin and load each admin.py file:
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    (r'^admin/(.*)', admin.site.root),

    # ... the rest of your URLs here ...
)
```

Views

Use `django.forms` instead of `newforms` Replace `django.newforms` with `django.forms` – Django 1.0 renamed the `newforms` module (introduced in 0.96) to plain old `forms`. The `oldforms` module was also removed.

If you're already using the `newforms` library, and you used our recommended `import` statement syntax, all you have to do is change your import statements.

Old:

```
from django import newforms as forms
```

New:

```
from django import forms
```

If you're using the old forms system (formerly known as `django.forms` and `django.oldforms`), you'll have to rewrite your forms. A good place to start is the forms documentation

Handle uploaded files using the new API Replace use of uploaded files – that is, entries in `request.FILES` – as simple dictionaries with the new *`UploadedFile`*. The old dictionary syntax no longer works.

Thus, in a view like:

```
def my_view(request):
    f = request.FILES['file_field_name']
    ...
```

...you'd need to make the following changes:

Old (0.96)	New (1.0)
`f['content']`	`f.read()`
`f['filename']`	`f.name`
`f['content-type']`	`f.content_type`

Work with file fields using the new API The internal implementation of *`django.db.models.FileField`* have changed. A visible result of this is that the way you access special attributes (URL, filename, image size, etc.) of these model fields has changed. You will need to make the following changes, assuming your model's *`FileField`* is called `myfile`:

Old (0.96)	New (1.0)
`myfile.get_content_filename()`	`myfile.content.path`
`myfile.get_content_url()`	`myfile.content.url`
`myfile.get_content_size()`	`myfile.content.size`
`myfile.save_content_file()`	`myfile.content.save()`
`myfile.get_content_width()`	`myfile.content.width`
`myfile.get_content_height()`	`myfile.content.height`

Note that the `width` and `height` attributes only make sense for *`ImageField`* fields. More details can be found in the model API documentation.

Use `Paginator` instead of `ObjectPaginator` The `ObjectPaginator` in 0.96 has been removed and replaced with an improved version, *`django.core.paginator.Paginator`*.

Templates

Learn to love autoescaping By default, the template system now automatically HTML-escapes the output of every variable. To learn more, see *Automatic HTML escaping*.

To disable auto-escaping for an individual variable, use the *`safe`* filter:

```
This will be escaped: {{ data }}
This will not be escaped: {{ data|safe }}
```

To disable auto-escaping for an entire template, wrap the template (or just a particular section of the template) in the *`autoescape`* tag:

```
{% autoescape off %}
   ... unescaped template content here ...
{% endautoescape %}
```

Less-common changes The following changes are smaller, more localized changes. They should only affect more advanced users, but it's probably worth reading through the list and checking your code for these things.

Signals

- Add `**kwargs` to any registered signal handlers.
- Connect, disconnect, and send signals via methods on the *`Signal`* object instead of through module methods in `django.dispatch.dispatcher`.
- Remove any use of the `Anonymous` and `Any` sender options; they no longer exist. You can still receive signals sent by any sender by using `sender=None`
- Make any custom signals you've declared into instances of *`django.dispatch.Signal`* instead of anonymous objects.

Here's quick summary of the code changes you'll need to make:

Old (0.96)	New (1.0)
`def callback(sender)`	`def callback(sender, **kwargs)`
`sig = object()`	`sig = django.dispatch.Signal()`
`dispatcher.connect(callback, sig)`	`sig.connect(callback)`
`dispatcher.send(sig, sender)`	`sig.send(sender)`
`dispatcher.connect(callback, sig, sender=Any)`	`sig.connect(callback, sender=None)`

Comments If you were using Django 0.96's `django.contrib.comments` app, you'll need to upgrade to the new comments app introduced in 1.0. See the upgrade guide for details.

Template tags

`spaceless` tag The spaceless template tag now removes *all* spaces between HTML tags, instead of preserving a single space.

Local flavors

U.S. local flavor `django.contrib.localflavor.usa` has been renamed to `django.contrib.localflavor.us`. This change was made to match the naming scheme of other local flavors. To migrate your code, all you need to do is change the imports.

Sessions

Getting a new session key `SessionBase.get_new_session_key()` has been renamed to `_get_new_session_key()`. `get_new_session_object()` no longer exists.

Fixtures

Loading a row no longer calls `save()` Previously, loading a row automatically ran the model's `save()` method. This is no longer the case, so any fields (for example: timestamps) that were auto-populated by a `save()` now need explicit values in any fixture.

Settings

Better exceptions The old `EnvironmentError` has split into an `ImportError` when Django fails to find the settings module and a `RuntimeError` when you try to reconfigure settings after having already used them.

`LOGIN_URL` has moved The *`LOGIN_URL`* constant moved from `django.contrib.auth` into the `settings` module. Instead of using `from django.contrib.auth import LOGIN_URL` refer to *`settings.LOGIN_URL`*.

`APPEND_SLASH` behavior has been updated In 0.96, if a URL didn't end in a slash or have a period in the final component of its path, and *`APPEND_SLASH`* was True, Django would redirect to the same URL, but with a slash appended to the end. Now, Django checks to see whether the pattern without the trailing slash would be matched by something in your URL patterns. If so, no redirection takes place, because it is assumed you deliberately wanted to catch that pattern.

For most people, this won't require any changes. Some people, though, have URL patterns that look like this:

```
r'/some_prefix/(.*)$'
```

Previously, those patterns would have been redirected to have a trailing slash. If you always want a slash on such URLs, rewrite the pattern as:

```
r'/some_prefix/(.*/)$'
```

Smaller model changes

Different exception from `get()` Managers now return a *`MultipleObjectsReturned`* exception instead of `AssertionError`:

Old (0.96):

```
try:
    Model.objects.get(...)
except AssertionError:
    handle_the_error()
```

New (1.0):

```
try:
    Model.objects.get(...)
except Model.MultipleObjectsReturned:
    handle_the_error()
```

`LazyDate` has been fired The `LazyDate` helper class no longer exists.

Default field values and query arguments can both be callable objects, so instances of `LazyDate` can be replaced with a reference to `datetime.datetime.now`:

Old (0.96):

```
class Article(models.Model):
    title = models.CharField(maxlength=100)
    published = models.DateField(default=LazyDate())
```

New (1.0):

```
import datetime

class Article(models.Model):
    title = models.CharField(max_length=100)
    published = models.DateField(default=datetime.datetime.now)
```

`DecimalField` is new, and `FloatField` is now a proper float Old (0.96):

```
class MyModel(models.Model):
    field_name = models.FloatField(max_digits=10, decimal_places=3)
    ...
```

New (1.0):

```
class MyModel(models.Model):
    field_name = models.DecimalField(max_digits=10, decimal_places=3)
    ...
```

If you forget to make this change, you will see errors about `FloatField` not taking a `max_digits` attribute in `__init__`, because the new `FloatField` takes no precision-related arguments.

If you're using MySQL or PostgreSQL, no further changes are needed. The database column types for `DecimalField` are the same as for the old `FloatField`.

If you're using SQLite, you need to force the database to view the appropriate columns as decimal types, rather than floats. To do this, you'll need to reload your data. Do this after you have made the change to using `DecimalField` in your code and updated the Django code.

: Back up your database first!
For SQLite, this means making a copy of the single file that stores the database (the name of that file is the `DATABASE_NAME` in your settings.py file).

To upgrade each application to use a `DecimalField`, you can do the following, replacing `<app>` in the code below with each app's name:

```
$ ./manage.py dumpdata --format=xml <app> > data-dump.xml
$ ./manage.py reset <app>
$ ./manage.py loaddata data-dump.xml
```

Notes:

1. It's important that you remember to use XML format in the first step of this process. We are exploiting a feature of the XML data dumps that makes porting floats to decimals with SQLite possible.
2. In the second step you will be asked to confirm that you are prepared to lose the data for the application(s) in question. Say yes; we'll restore this data in the third step, of course.
3. `DecimalField` is not used in any of the apps shipped with Django prior to this change being made, so you do not need to worry about performing this procedure for any of the standard Django models.

If something goes wrong in the above process, just copy your backed up database file over the original file and start again.

Internationalization

`django.views.i18n.set_language()` now requires a POST request Previously, a GET request was used. The old behavior meant that state (the locale used to display the site) could be changed by a GET request, which is against the HTTP specification's recommendations. Code calling this view must ensure that a POST request is now made, instead of a GET. This means you can no longer use a link to access the view, but must use a form submission of some kind (e.g. a button).

`_()` is no longer in builtins `_()` (the callable object whose name is a single underscore) is no longer monkey-patched into builtins – that is, it's no longer available magically in every module.

If you were previously relying on `_()` always being present, you should now explicitly import `ugettext` or `ugettext_lazy`, if appropriate, and alias it to _ yourself:

```
from django.utils.translation import ugettext as _
```

HTTP request/response objects

Dictionary access to `HttpRequest` `HttpRequest` objects no longer directly support dictionary-style access; previously, both `GET` and `POST` data were directly available on the `HttpRequest` object (e.g., you could check for a piece of form data by using `if 'some_form_key' in request` or by reading `request['some_form_key']`. This is no longer supported; if you need access to the combined `GET` and `POST` data, use `request.REQUEST` instead.

It is strongly suggested, however, that you always explicitly look in the appropriate dictionary for the type of request you expect to receive (`request.GET` or `request.POST`); relying on the combined `request.REQUEST` dictionary can mask the origin of incoming data.

Accessing `HTTPResponse` headers `django.http.HttpResponse.headers` has been renamed to `_headers` and *`HttpResponse`* now supports containment checking directly. So use `if header in response:` instead of `if header in response.headers:`.

Generic relations

Generic relations have been moved out of core The generic relation classes – `GenericForeignKey` and `GenericRelation` – have moved into the *`django.contrib.contenttypes`* module.

Testing

`django.test.Client.login()` has changed Old (0.96):

```
from django.test import Client
c = Client()
c.login('/path/to/login','myuser','mypassword')
```

New (1.0):

```
# ... same as above, but then:
c.login(username='myuser', password='mypassword')
```

Management commands

Running management commands from your code *django.core.management* has been greatly refactored.

Calls to management services in your code now need to use `call_command`. For example, if you have some test code that calls flush and load_data:

```
from django.core import management
management.flush(verbosity=0, interactive=False)
management.load_data(['test_data'], verbosity=0)
```

...you'll need to change this code to read:

```
from django.core import management
management.call_command('flush', verbosity=0, interactive=False)
management.call_command('loaddata', 'test_data', verbosity=0)
```

Subcommands must now precede options `django-admin.py` and `manage.py` now require subcommands to precede options. So:

```
$ django-admin.py --settings=foo.bar runserver
```

...no longer works and should be changed to:

```
$ django-admin.py runserver --settings=foo.bar
```

Syndication

`Feed.__init__` has changed The `__init__()` method of the syndication framework's `Feed` class now takes an `HttpRequest` object as its second parameter, instead of the feed's URL. This allows the syndication framework to work without requiring the sites framework. This only affects code that subclasses `Feed` and overrides the `__init__()` method, and code that calls `Feed.__init__()` directly.

Data structures

`SortedDictFromList` is gone `django.newforms.forms.SortedDictFromList` was removed. `django.utils.datastructures.SortedDict` can now be instantiated with a sequence of tuples.

To update your code:

1. Use `django.utils.datastructures.SortedDict` wherever you were using `django.newforms.forms.SortedDictFromList`.
2. Because `django.utils.datastructures.SortedDict.copy` doesn't return a deepcopy as `SortedDictFromList.copy()` did, you will need to update your code if you were relying on a deepcopy. Do this by using `copy.deepcopy` directly.

Database backend functions

Database backend functions have been renamed Almost *all* of the database backend-level functions have been renamed and/or relocated. None of these were documented, but you'll need to change your code if you're using any of these functions, all of which are in *django.db*:

Old (0.96)	New (1.0)
backend.get_autoinc_sql	connection.ops.autoinc_sql
backend.get_date_extract_sql	connection.ops.date_extract_sql
backend.get_date_trunc_sql	connection.ops.date_trunc_sql
backend.get_datetime_cast_sql	connection.ops.datetime_cast_sql
backend.get_deferrable_sql	connection.ops.deferrable_sql
backend.get_drop_foreignkey_sql	connection.ops.drop_foreignkey_sql
backend.get_fulltext_search_sql	connection.ops.fulltext_search_sql
backend.get_last_insert_id	connection.ops.last_insert_id
backend.get_limit_offset_sql	connection.ops.limit_offset_sql
backend.get_max_name_length	connection.ops.max_name_length
backend.get_pk_default_value	connection.ops.pk_default_value
backend.get_random_function_sql	connection.ops.random_function_sql
backend.get_sql_flush	connection.ops.sql_flush
backend.get_sql_sequence_reset	connection.ops.sequence_reset_sql
backend.get_start_transaction_sql	connection.ops.start_transaction_sql
backend.get_tablespace_sql	connection.ops.tablespace_sql
backend.quote_name	connection.ops.quote_name
backend.get_query_set_class	connection.ops.query_set_class
backend.get_field_cast_sql	connection.ops.field_cast_sql
backend.get_drop_sequence	connection.ops.drop_sequence_sql
backend.OPERATOR_MAPPING	connection.operators
backend.allows_group_by_ordinal	connection.features.allows_group_by_ordinal
backend.allows_unique_and_pk	connection.features.allows_unique_and_pk
backend.autoindexes_primary_keys	connection.features.autoindexes_primary_keys
backend.needs_datetime_string_cast	connection.features.needs_datetime_string_cast
backend.needs_upper_for_iops	connection.features.needs_upper_for_iops
backend.supports_constraints	connection.features.supports_constraints
backend.supports_tablespaces	connection.features.supports_tablespaces
backend.uses_case_insensitive_names	connection.features.uses_case_insensitive_names
backend.uses_custom_queryset	connection.features.uses_custom_queryset

A complete list of backwards-incompatible changes can be found at https://code.djangoproject.com/wiki/BackwardsIncompatibleChange

What's new in Django 1.0

A *lot*!

Since Django 0.96, we've made over 4,000 code commits, fixed more than 2,000 bugs, and edited, added, or removed around 350,000 lines of code. We've also added 40,000 lines of new documentation, and greatly improved what was already there.

In fact, new documentation is one of our favorite features of Django 1.0, so we might as well start there. First, there's a new documentation site:

- https://docs.djangoproject.com/

The documentation has been greatly improved, cleaned up, and generally made awesome. There's now dedicated search, indexes, and more.

We can't possibly document everything that's new in 1.0, but the documentation will be your definitive guide. Anywhere you see something like:

This feature is new in Django 1.0

You'll know that you're looking at something new or changed.

The other major highlights of Django 1.0 are:

Re-factored admin application The Django administrative interface (`django.contrib.admin`) has been completely refactored; admin definitions are now completely decoupled from model definitions (no more `class Admin` declaration in models!), rewritten to use Django's new form-handling library (introduced in the 0.96 release as `django.newforms`, and now available as simply `django.forms`) and redesigned with extensibility and customization in mind. Full documentation for the admin application is available online in the official Django documentation:

See the admin reference for details

Improved Unicode handling Django's internals have been refactored to use Unicode throughout; this drastically simplifies the task of dealing with non-Western-European content and data in Django. Additionally, utility functions have been provided to ease interoperability with third-party libraries and systems which may or may not handle Unicode gracefully. Details are available in Django's Unicode-handling documentation.

See Unicode data.

An improved ORM Django's object-relational mapper – the component which provides the mapping between Django model classes and your database, and which mediates your database queries – has been dramatically improved by a massive refactoring. For most users of Django this is backwards-compatible; the public-facing API for database querying underwent a few minor changes, but most of the updates took place in the ORM's internals. A guide to the changes, including backwards-incompatible modifications and mentions of new features opened up by this refactoring, is available on the Django wiki.

Automatic escaping of template variables To provide improved security against cross-site scripting (XSS) vulnerabilities, Django's template system now automatically escapes the output of variables. This behavior is configurable, and allows both variables and larger template constructs to be marked as safe (requiring no escaping) or unsafe (requiring escaping). A full guide to this feature is in the documentation for the *`autoescape`* tag.

`django.contrib.gis` (GeoDjango) A project over a year in the making, this adds world-class GIS (Geographic Information Systems) support to Django, in the form of a `contrib` application. Its documentation is currently being maintained externally, and will be merged into the main Django documentation shortly. Huge thanks go to Justin Bronn, Jeremy Dunck, Brett Hoerner and Travis Pinney for their efforts in creating and completing this feature.

See http://geodjango.org/ for details.

Pluggable file storage Django's built-in `FileField` and `ImageField` now can take advantage of pluggable file-storage backends, allowing extensive customization of where and how uploaded files get stored by Django. For details, see the files documentation; big thanks go to Marty Alchin for putting in the hard work to get this completed.

Jython compatibility Thanks to a lot of work from Leo Soto during a Google Summer of Code project, Django's codebase has been refactored to remove incompatibilities with Jython, an implementation of Python written in Java, which runs Python code on the Java Virtual Machine. Django is now compatible with the forthcoming Jython 2.5 release.

See Running Django on Jython.

Generic relations in forms and admin Classes are now included in `django.contrib.contenttypes` which can be used to support generic relations in both the admin interface and in end-user forms. See *the documentation for generic relations* for details.

`INSERT/UPDATE` distinction Although Django's default behavior of having a model's `save()` method automatically determine whether to perform an `INSERT` or an `UPDATE` at the SQL level is suitable for the majority of cases, there are occasional situations where forcing one or the other is useful. As a result, models can now support an additional parameter to `save()` which can force a specific operation.

See *Forcing an INSERT or UPDATE* for details.

Split `CacheMiddleware` Django's `CacheMiddleware` has been split into three classes: `CacheMiddleware` itself still exists and retains all of its previous functionality, but it is now built from two separate middleware classes which handle the two parts of caching (inserting into and reading from the cache) separately, offering additional flexibility for situations where combining these functions into a single middleware posed problems.

Full details, including updated notes on appropriate use, are in the caching documentation.

Refactored `django.contrib.comments` As part of a Google Summer of Code project, Thejaswi Puthraya carried out a major rewrite and refactoring of Django's bundled comment system, greatly increasing its flexibility and customizability.

Removal of deprecated features A number of features and methods which had previously been marked as deprecated, and which were scheduled for removal prior to the 1.0 release, are no longer present in Django. These include imports of the form library from `django.newforms` (now located simply at `django.forms`), the `form_for_model` and `form_for_instance` helper functions (which have been replaced by `ModelForm`) and a number of deprecated features which were replaced by the dispatcher, file-uploading and file-storage refactorings introduced in the Django 1.0 alpha releases.

Known issues

We've done our best to make Django 1.0 as solid as possible, but unfortunately there are a couple of issues that we know about in the release.

Multi-table model inheritance with `to_field` If you're using *multiple table model inheritance*, be aware of this caveat: child models using a custom `parent_link` and `to_field` will cause database integrity errors. A set of models like the following are **not valid**:

```
class Parent(models.Model):
    name = models.CharField(max_length=10)
    other_value = models.IntegerField(unique=True)

class Child(Parent):
    father = models.OneToOneField(Parent, primary_key=True, to_field="other_value", parent_link=True)
    value = models.IntegerField()
```

This bug will be fixed in the next release of Django.

Caveats with support of certain databases Django attempts to support as many features as possible on all database backends. However, not all database backends are alike, and in particular many of the supported database differ greatly from version to version. It's a good idea to checkout our notes on supported database:

- *MySQL notes*
- *SQLite notes*
- *Oracle notes*

9.1.12 Pre-1.0 releases

Django version 0.96 release notes

Welcome to Django 0.96!

The primary goal for 0.96 is a cleanup and stabilization of the features introduced in 0.95. There have been a few small *backwards-incompatible changes* since 0.95, but the upgrade process should be fairly simple and should not require major changes to existing applications.

However, we're also releasing 0.96 now because we have a set of backwards-incompatible changes scheduled for the near future. Once completed, they will involve some code changes for application developers, so we recommend that you stick with Django 0.96 until the next official release; then you'll be able to upgrade in one step instead of needing to make incremental changes to keep up with the development version of Django.

Backwards-incompatible changes

The following changes may require you to update your code when you switch from 0.95 to 0.96:

`MySQLdb` version requirement Due to a bug in older versions of the `MySQLdb` Python module (which Django uses to connect to MySQL databases), Django's MySQL backend now requires version 1.2.1p2 or higher of `MySQLdb`, and will raise exceptions if you attempt to use an older version.

If you're currently unable to upgrade your copy of `MySQLdb` to meet this requirement, a separate, backwards-compatible backend, called "mysql_old", has been added to Django. To use this backend, change the `DATABASE_ENGINE` setting in your Django settings file from this:

```
DATABASE_ENGINE = "mysql"
```

to this:

```
DATABASE_ENGINE = "mysql_old"
```

However, we strongly encourage MySQL users to upgrade to a more recent version of `MySQLdb` as soon as possible, The "mysql_old" backend is provided only to ease this transition, and is considered deprecated; aside from any necessary security fixes, it will not be actively maintained, and it will be removed in a future release of Django.

Also, note that some features, like the new `DATABASE_OPTIONS` setting (see the databases documentation for details), are only available on the "mysql" backend, and will not be made available for "mysql_old".

Database constraint names changed The format of the constraint names Django generates for foreign key references have changed slightly. These names are generally only used when it is not possible to put the reference directly on the affected column, so they are not always visible.

The effect of this change is that running `manage.py reset` and similar commands against an existing database may generate SQL with the new form of constraint name, while the database itself contains constraints named in the old form; this will cause the database server to raise an error message about modifying non-existent constraints.

If you need to work around this, there are two methods available:

1. Redirect the output of `manage.py` to a file, and edit the generated SQL to use the correct constraint names before executing it.
2. Examine the output of `manage.py sqlall` to see the new-style constraint names, and use that as a guide to rename existing constraints in your database.

Name changes in `manage.py` A few of the options to `manage.py` have changed with the addition of fixture support:

- There are new `dumpdata` and `loaddata` commands which, as you might expect, will dump and load data to/from the database. These commands can operate against any of Django's supported serialization formats.
- The `sqlinitialdata` command has been renamed to `sqlcustom` to emphasize that `loaddata` should be used for data (and `sqlcustom` for other custom SQL – views, stored procedures, etc.).
- The vestigial `install` command has been removed. Use `syncdb`.

Backslash escaping changed The Django database API now escapes backslashes given as query parameters. If you have any database API code that matches backslashes, and it was working before (despite the lack of escaping), you'll have to change your code to "unescape" the slashes one level.

For example, this used to work:

```
# Find text containing a single backslash
MyModel.objects.filter(text__contains='\\\\')
```

The above is now incorrect, and should be rewritten as:

```
# Find text containing a single backslash
MyModel.objects.filter(text__contains='\\')
```

Removed ENABLE_PSYCO setting The `ENABLE_PSYCO` setting no longer exists. If your settings file includes `ENABLE_PSYCO` it will have no effect; to use Psyco, we recommend writing a middleware class to activate it.

What's new in 0.96?

This revision represents over a thousand source commits and over four hundred bug fixes, so we can't possibly catalog all the changes. Here, we describe the most notable changes in this release.

New forms library `django.newforms` is Django's new form-handling library. It's a replacement for `django.forms`, the old form/manipulator/validation framework. Both APIs are available in 0.96, but over the next two releases we plan to switch completely to the new forms system, and deprecate and remove the old system.

There are three elements to this transition:

- We've copied the current `django.forms` to `django.oldforms`. This allows you to upgrade your code *now* rather than waiting for the backwards-incompatible change and rushing to fix your code after the fact. Just change your import statements like this:

```
    from django import forms             # 0.95-style
    from django import oldforms as forms # 0.96-style
```

- The next official release of Django will move the current `django.newforms` to `django.forms`. This will be a backwards-incompatible change, and anyone still using the old version of `django.forms` at that time will need to change their import statements as described above.
- The next release after that will completely remove `django.oldforms`.

Although the `newforms` library will continue to evolve, it's ready for use for most common cases. We recommend that anyone new to form handling skip the old forms system and start with the new.

For more information about `django.newforms`, read the newforms documentation.

URLconf improvements You can now use any callable as the callback in URLconfs (previously, only strings that referred to callables were allowed). This allows a much more natural use of URLconfs. For example, this URLconf:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    ('^myview/$', 'mysite.myapp.views.myview')
)
```

can now be rewritten as:

```
from django.conf.urls.defaults import *
from mysite.myapp.views import myview

urlpatterns = patterns('',
    ('^myview/$', myview)
)
```

One useful application of this can be seen when using decorators; this change allows you to apply decorators to views *in your URLconf*. Thus, you can make a generic view require login very easily:

```
from django.conf.urls.defaults import *
from django.contrib.auth.decorators import login_required
from django.views.generic.list_detail import object_list
from mysite.myapp.models import MyModel

info = {
    "queryset" : MyModel.objects.all(),
}

urlpatterns = patterns('',
    ('^myview/$', login_required(object_list), info)
)
```

Note that both syntaxes (strings and callables) are valid, and will continue to be valid for the foreseeable future.

The test framework Django now includes a test framework so you can start transmuting fear into boredom (with apologies to Kent Beck). You can write tests based on `doctest` or `unittest` and test your views with a simple test client.

There is also new support for "fixtures" – initial data, stored in any of the supported serialization formats, that will be loaded into your database at the start of your tests. This makes testing with real data much easier.

See the testing documentation for the full details.

Improvements to the admin interface A small change, but a very nice one: dedicated views for adding and updating users have been added to the admin interface, so you no longer need to worry about working with hashed passwords in the admin.

Thanks

Since 0.95, a number of people have stepped forward and taken a major new role in Django's development. We'd like to thank these people for all their hard work:

- Russell Keith-Magee and Malcolm Tredinnick for their major code contributions. This release wouldn't have been possible without them.
- Our new release manager, James Bennett, for his work in getting out 0.95.1, 0.96, and (hopefully) future release.
- Our ticket managers Chris Beaven (aka SmileyChris), Simon Greenhill, Michael Radziej, and Gary Wilson. They agreed to take on the monumental task of wrangling our tickets into nicely cataloged submission. Figuring out what to work on is now about a million times easier; thanks again, guys.
- Everyone who submitted a bug report, patch or ticket comment. We can't possibly thank everyone by name – over 200 developers submitted patches that went into 0.96 – but everyone who's contributed to Django is listed in AUTHORS.

Django version 0.95 release notes

Welcome to the Django 0.95 release.

This represents a significant advance in Django development since the 0.91 release in January 2006. The details of every change in this release would be too extensive to list in full, but a summary is presented below.

Suitability and API stability

This release is intended to provide a stable reference point for developers wanting to work on production-level applications that use Django.

However, it's not the 1.0 release, and we'll be introducing further changes before 1.0. For a clear look at which areas of the framework will change (and which ones will *not* change) before 1.0, see the `api-stability.txt` file, which lives in the docs/ directory of the distribution.

You may have a need to use some of the features that are marked as "subject to API change" in that document, but that's OK with us as long as it's OK with you, and as long as you understand APIs may change in the future.

Fortunately, most of Django's core APIs won't be changing before version 1.0. There likely won't be as big of a change between 0.95 and 1.0 versions as there was between 0.91 and 0.95.

Changes and new features

The major changes in this release (for developers currently using the 0.91 release) are a result of merging the 'magic-removal' branch of development. This branch removed a number of constraints in the way Django code had to be written that were a consequence of decisions made in the early days of Django, prior to its open-source release. It's now possible to write more natural, Pythonic code that works as expected, and there's less "black magic" happening behind the scenes.

Aside from that, another main theme of this release is a dramatic increase in usability. We've made countless improvements in error messages, documentation, etc., to improve developers' quality of life.

The new features and changes introduced in 0.95 include:

- Django now uses a more consistent and natural filtering interface for retrieving objects from the database.
- User-defined models, functions and constants now appear in the module namespace they were defined in. (Previously everything was magically transferred to the django.models.* namespace.)
- Some optional applications, such as the FlatPage, Sites and Redirects apps, have been decoupled and moved into django.contrib. If you don't want to use these applications, you no longer have to install their database tables.
- Django now has support for managing database transactions.
- We've added the ability to write custom authentication and authorization backends for authenticating users against alternate systems, such as LDAP.
- We've made it easier to add custom table-level functions to models, through a new "Manager" API.
- It's now possible to use Django without a database. This simply means that the framework no longer requires you to have a working database set up just to serve dynamic pages. In other words, you can just use URLconfs/views on their own. Previously, the framework required that a database be configured, regardless of whether you actually used it.
- It's now more explicit and natural to override save() and delete() methods on models, rather than needing to hook into the pre_save() and post_save() method hooks.
- Individual pieces of the framework now can be configured without requiring the setting of an environment variable. This permits use of, for example, the Django templating system inside other applications.
- More and more parts of the framework have been internationalized, as we've expanded internationalization (i18n) support. The Django codebase, including code and templates, has now been translated, at least in part, into 31 languages. From Arabic to Chinese to Hungarian to Welsh, it is now possible to use Django's admin site in your native language.

The number of changes required to port from 0.91-compatible code to the 0.95 code base are significant in some cases. However, they are, for the most part, reasonably routine and only need to be done once. A list of the necessary changes is described in the Removing The Magic wiki page. There is also an easy checklist for reference when undertaking the porting operation.

Problem reports and getting help

Need help resolving a problem with Django? The documentation in the distribution is also available online at the Django website. The FAQ document is especially recommended, as it contains a number of issues that come up time and again.

For more personalized help, the django-users mailing list is a very active list, with more than 2,000 subscribers who can help you solve any sort of Django problem. We recommend you search the archives first, though, because many common questions appear with some regularity, and any particular problem may already have been answered.

Finally, for those who prefer the more immediate feedback offered by IRC, there's a #django channel on irc.freenode.net that is regularly populated by Django users and developers from around the world. Friendly people are usually available at any hour of the day – to help, or just to chat.

Thanks for using Django!

The Django Team July 2006

9.2 Security releases

Whenever a security issue is disclosed via Django's security policies, appropriate release notes are now added to all affected release series.

Additionally, an archive of disclosed security issues is maintained.

9.2.1 Archive of security issues

Django's development team is strongly committed to responsible reporting and disclosure of security-related issues, as outlined in Django's security policies.

As part of that commitment, we maintain the following historical list of issues which have been fixed and disclosed. For each issue, the list below includes the date, a brief description, the CVE identifier if applicable, a list of affected versions, a link to the full disclosure and links to the appropriate patch(es).

Some important caveats apply to this information:

- Lists of affected versions include only those versions of Django which had stable, security-supported releases at the time of disclosure. This means older versions (whose security support had expired) and versions which were in pre-release (alpha/beta/RC) states at the time of disclosure may have been affected, but are not listed.
- The Django project has on occasion issued security advisories, pointing out potential security problems which can arise from improper configuration or from other issues outside of Django itself. Some of these advisories have received CVEs; when that is the case, they are listed here, but as they have no accompanying patches or releases, only the description, disclosure and CVE will be listed.

Issues prior to Django's security process

Some security issues were handled before Django had a formalized security process in use. For these, new releases may not have been issued at the time and CVEs may not have been assigned.

August 16, 2006 - CVE-2007-0404

Filename validation issue in translation framework. Full description

Versions affected

- Django 0.90 (patch)
- Django 0.91 (patch)
- Django 0.95 (patch) (released January 21 2007)

January 21, 2007 - CVE-2007-0405

Apparent "caching" of authenticated user. Full description

Versions affected

- Django 0.95 (patch)

Issues under Django's security process

All other security issues have been handled under versions of Django's security process. These are listed below.

October 26, 2007 - CVE-2007-5712

Denial-of-service via arbitrarily-large `Accept-Language` header. Full description

Versions affected

- Django 0.91 (patch)
- Django 0.95 (patch)
- Django 0.96 (patch)

May 14, 2008 - CVE-2008-2302

XSS via admin login redirect. Full description

Versions affected

- Django 0.91 (patch)
- Django 0.95 (patch)
- Django 0.96 (patch)

September 2, 2008 - CVE-2008-3909

CSRF via preservation of POST data during admin login. Full description

Versions affected

- Django 0.91 (patch)
- Django 0.95 (patch)
- Django 0.96 (patch)

July 28, 2009 - CVE-2009-2659

Directory-traversal in development server media handler. Full description

Versions affected

- Django 0.96 (patch)
- Django 1.0 (patch)

October 9, 2009 - CVE-2009-3965

Denial-of-service via pathological regular expression performance. Full description

Versions affected

- Django 1.0 (patch)
- Django 1.1 (patch)

September 8, 2010 - CVE-2010-3082

XSS via trusting unsafe cookie value. Full description

Versions affected

- Django 1.2 (patch)

December 22, 2010 - CVE-2010-4534

Information leakage in administrative interface. Full description

Versions affected

- Django 1.1 (patch)
- Django 1.2 (patch)

December 22, 2010 - CVE-2010-4535

Denial-of-service in password-reset mechanism. Full description

Versions affected

- Django 1.1 (patch)
- Django 1.2 (patch)

February 8, 2011 - CVE-2011-0696

CSRF via forged HTTP headers. Full description

Versions affected

- Django 1.1 (patch)
- Django 1.2 (patch)

February 8, 2011 - CVE-2011-0697

XSS via unsanitized names of uploaded files. Full description

Versions affected

- Django 1.1 (patch)
- Django 1.2 (patch)

February 8, 2011 - CVE-2011-0698

Directory-traversal on Windows via incorrect path-separator handling. Full description

Versions affected

- Django 1.1 (patch)
- Django 1.2 (patch)

September 9, 2011 - CVE-2011-4136

Session manipulation when using memory-cache-backed session. Full description

Versions affected

- Django 1.2 (patch)
- Django 1.3 (patch)

September 9, 2011 - CVE-2011-4137

Denial-of-service via `URLField.verify_exists`. Full description

Versions affected

- Django 1.2 (patch)
- Django 1.3 (patch)

September 9, 2011 - CVE-2011-4138

Information leakage/arbitrary request issuance via `URLField.verify_exists`. Full description

Versions affected

- Django 1.2: (patch)
- Django 1.3: (patch)

September 9, 2011 - CVE-2011-4139

`Host` header cache poisoning. Full description

Versions affected

- Django 1.2 (patch)
- Django 1.3 (patch)

September 9, 2011 - CVE-2011-4140

Potential CSRF via `Host` header. Full description

Versions affected This notification was an advisory only, so no patches were issued.

- Django 1.2
- Django 1.3

July 30, 2012 - CVE-2012-3442

XSS via failure to validate redirect scheme. Full description

Versions affected

- Django 1.3: (patch)
- Django 1.4: (patch)

July 30, 2012 - CVE-2012-3443

Denial-of-service via compressed image files. Full description

Versions affected

- Django 1.3: (patch)
- Django 1.4: (patch)

July 30, 2012 - CVE-2012-3444

Denial-of-service via large image files. Full description

Versions affected

- Django 1.3 (patch)
- Django 1.4 (patch)

October 17, 2012 - CVE-2012-4520

`Host` header poisoning. Full description

Versions affected

- Django 1.3 (patch)
- Django 1.4 (patch)

December 10, 2012 - No CVE 1

Additional hardening of `Host` header handling. Full description

Versions affected

- Django 1.3 (patch)
- Django 1.4 (patch)

December 10, 2012 - No CVE 2

Additional hardening of redirect validation. Full description

Versions affected

- Django 1.3: (patch)
- Django 1.4: (patch)

February 19, 2013 - No CVE

Additional hardening of `Host` header handling. Full description

Versions affected

- Django 1.3 (patch)
- Django 1.4 (patch)

February 19, 2013 - CVE-2013-1664 / CVE-2013-1665

Entity-based attacks against Python XML libraries. Full description

Versions affected

- Django 1.3 (patch)
- Django 1.4 (patch)

February 19, 2013 - CVE-2013-0305

Information leakage via admin history log. Full description

Versions affected

- Django 1.3 (patch)
- Django 1.4 (patch)

February 19, 2013 - CVE-2013-0306

Denial-of-service via formset `max_num` bypass. Full description

Versions affected

- Django 1.3 (patch)
- Django 1.4 (patch)

August 13, 2013 - CVE-2013-4249

XSS via admin trusting `URLField` values. Full description

Versions affected

- Django 1.5 (patch)

August 13, 2013 - CVE-2013-6044

Possible XSS via unvalidated URL redirect schemes. Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)

September 10, 2013 - CVE-2013-4315

Directory-traversal via `ssi` template tag. Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)

September 14, 2013 - CVE-2013-1443

Denial-of-service via large passwords. Full description

Versions affected

- Django 1.4 (patch and Python compatibility fix)
- Django 1.5 (patch)

April 21, 2014 - CVE-2014-0472

Unexpected code execution using `reverse()`. Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

April 21, 2014 - CVE-2014-0473

Caching of anonymous pages could reveal CSRF token. Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

April 21, 2014 - CVE-2014-0474

MySQL typecasting causes unexpected query results. Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

May 18, 2014 - CVE-2014-1418

Caches may be allowed to store and serve private data. Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

May 18, 2014 - CVE-2014-3730

Malformed URLs from user input incorrectly validated. Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

August 20, 2014 - CVE-2014-0480

`reverse()` can generate URLs pointing to other hosts. Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

August 20, 2014 - CVE-2014-0481

File upload denial of service. Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

August 20, 2014 - CVE-2014-0482

`RemoteUserMiddleware` session hijacking. Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

August 20, 2014 - CVE-2014-0483

Data leakage via querystring manipulation in admin. Full description

Versions affected

- Django 1.4 (patch)
- Django 1.5 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

January 13, 2015 - CVE-2015-0219

WSGI header spoofing via underscore/dash conflation. Full description

Versions affected

- Django 1.4 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

January 13, 2015 - CVE-2015-0220

Mitigated possible XSS attack via user-supplied redirect URLs. Full description

Versions affected

- Django 1.4 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

January 13, 2015 - CVE-2015-0221

Denial-of-service attack against `django.views.static.serve()`. Full description

Versions affected

- Django 1.4 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)

January 13, 2015 - CVE-2015-0222

Database denial-of-service with `ModelMultipleChoiceField`. Full description

Versions affected

- Django 1.6 (patch)
- Django 1.7 (patch)

March 9, 2015 - CVE-2015-2241

XSS attack via properties in `ModelAdmin.readonly_fields`. Full description

Versions affected

- Django 1.7 (patch)
- Django 1.8 (patch)

March 18, 2015 - CVE-2015-2316

Denial-of-service possibility with `strip_tags()`. Full description

Versions affected

- Django 1.6 (patch)
- Django 1.7 (patch)
- Django 1.8 (patch)

March 18, 2015 - CVE-2015-2317

Mitigated possible XSS attack via user-supplied redirect URLs. Full description

Versions affected

- Django 1.4 (patch)
- Django 1.6 (patch)
- Django 1.7 (patch)
- Django 1.8 (patch)

May 20, 2015 - CVE-2015-3982

Fixed session flushing in the cached_db backend. Full description

Versions affected

- Django 1.8 (patch)

July 8, 2015 - CVE-2015-5143

Denial-of-service possibility by filling session store. Full description

Versions affected

- Django 1.8 (patch)
- Django 1.7 (patch)
- Django 1.4 (patch)

July 8, 2015 - CVE-2015-5144

Header injection possibility since validators accept newlines in input. Full description

Versions affected

- Django 1.8 (patch)
- Django 1.7 (patch)
- Django 1.4 (patch)

July 8, 2015 - CVE-2015-5145

Denial-of-service possibility in URL validation. Full description

Versions affected

- Django 1.8 (patch)

August 18, 2015 - CVE-2015-5963 / CVE-2015-5964

Denial-of-service possibility in `logout()` view by filling session store. Full description

Versions affected

- Django 1.8 (patch)
- Django 1.7 (patch)
- Django 1.4 (patch)

November 24, 2015 - CVE-2015-8213

Settings leak possibility in `date` template filter. Full description

Versions affected

- Django 1.8 (patch)
- Django 1.7 (patch)

February 1, 2016 - CVE-2016-2048

User with "change" but not "add" permission can create objects for `ModelAdmin`'s with `save_as=True`. Full description

Versions affected

- Django 1.9 (patch)

March 1, 2016 - CVE-2016-2512

Malicious redirect and possible XSS attack via user-supplied redirect URLs containing basic auth. Full description

Versions affected

- Django 1.9 (patch)
- Django 1.8 (patch)

March 1, 2016 - CVE-2016-2513

User enumeration through timing difference on password hasher work factor upgrade. Full description

Versions affected

- Django 1.9 (patch)
- Django 1.8 (patch)

July 18, 2016 - CVE-2016-6186

XSS in admin's add/change related popup. Full description

Versions affected

- Django 1.9 (patch)
- Django 1.8 (patch)

September 26, 2016 - CVE-2016-7401

CSRF protection bypass on a site with Google Analytics. Full description

Versions affected

- Django 1.9 (patch)
- Django 1.8 (patch)

CHAPTER 10

Django internals

Documentation for people hacking on Django itself. This is the place to go if you'd like to help improve Django or learn about how Django is managed.

10.1 Contributing to Django

Django is a community that lives on its volunteers. As it keeps growing, we always need more people to help others. As soon as you learn Django, you can contribute in many ways:

- Join the *django-users* mailing list and answer questions. This mailing list has a huge audience, and we really want to maintain a friendly and helpful atmosphere. If you're new to the Django community, you should read the posting guidelines.
- Join the #django IRC channel on Freenode and answer questions. By explaining Django to other users, you're going to learn a lot about the framework yourself.
- Blog about Django. We syndicate all the Django blogs we know about on the community page; if you'd like to see your blog on that page you can register it here.
- Contribute to open-source Django projects, write some documentation, or release your own code as an open-source pluggable application. The ecosystem of pluggable applications is a big strength of Django, help us build it!

If you think working *with* Django is fun, wait until you start working *on* it. We're passionate about helping Django users make the jump to contributing members of the community, so there are several ways you can help Django's development:

- Report bugs in our ticket tracker.
- Join the *django-developers* mailing list and share your ideas for how to improve Django. We're always open to suggestions.
- Submit patches for new and/or fixed behavior. If you're looking for an easy way to start contributing to Django read the Writing your first patch for Django tutorial and have a look at the easy pickings tickets. The *Patch review checklist* will also be helpful.
- Improve the documentation or write unit tests.
- Triage tickets and review patches created by other users.

Really, **ANYONE** can do something to help make Django better and greater!

Browse the following sections to find out how:

10.1.1 Advice for new contributors

New contributor and not sure what to do? Want to help but just don't know how to get started? This is the section for you.

Basic tools and workflow

If you are new to contributing to Django, the Writing your first patch for Django tutorial will give you an introduction to the tools and the workflow.

First steps

Start with these easy tasks to discover Django's development process.

- **Sign the Contributor License Agreement**

  The code that you write belongs to you or your employer. If your contribution is more than one or two lines of code, you need to sign the CLA. See the Contributor License Agreement FAQ for a more thorough explanation.

- **Triage tickets**

  If an unreviewed ticket reports a bug, try and reproduce it. If you can reproduce it and it seems valid, make a note that you confirmed the bug and accept the ticket. Make sure the ticket is filed under the correct component area. Consider writing a patch that adds a test for the bug's behavior, even if you don't fix the bug itself. See more at *How can I help with triaging?*

- **Look for tickets that are accepted and review patches to build familiarity with the codebase and the process**

  Mark the appropriate flags if a patch needs docs or tests. Look through the changes a patch makes, and keep an eye out for syntax that is incompatible with older but still supported versions of Python. Run the tests and make sure they pass. Where possible and relevant, try them out on a database other than SQLite. Leave comments and feedback!

- **Keep old patches up to date**

  Oftentimes the codebase will change between a patch being submitted and the time it gets reviewed. Make sure it still applies cleanly and functions as expected. Simply updating a patch is both useful and important! See more on Submitting patches.

- **Write some documentation**

  Django's documentation is great but it can always be improved. Did you find a typo? Do you think that something should be clarified? Go ahead and suggest a documentation patch! See also the guide on Writing documentation.

: The reports page contains links to many useful Trac queries, including several that are useful for triaging tickets and reviewing patches as suggested above.

Guidelines

As a newcomer on a large project, it's easy to experience frustration. Here's some advice to make your work on Django more useful and rewarding.

- **Pick a subject area that you care about, that you are familiar with, or that you want to learn about**

  You don't already have to be an expert on the area you want to work on; you become an expert through your ongoing contributions to the code.

- **Analyze tickets' context and history**

  Trac isn't an absolute; the context is just as important as the words. When reading Trac, you need to take into account who says things, and when they were said. Support for an idea two years ago doesn't necessarily mean that the idea will still have support. You also need to pay attention to who *hasn't* spoken – for example, if a core team member hasn't been recently involved in a discussion, then a ticket may not have the support required to get into trunk.

- **Start small**

  It's easier to get feedback on a little issue than on a big one. See the easy pickings.

- **If you're going to engage in a big task, make sure that your idea has support first**

  This means getting someone else to confirm that a bug is real before you fix the issue, and ensuring that the core team supports a proposed feature before you go implementing it.

- **Be bold! Leave feedback!**

  Sometimes it can be scary to put your opinion out to the world and say "this ticket is correct" or "this patch needs work", but it's the only way the project moves forward. The contributions of the broad Django community ultimately have a much greater impact than that of the core team. We can't do it without **you**!

- **Err on the side of caution when marking things Ready For Check-in**

  If you're really not certain if a ticket is ready, don't mark it as such. Leave a comment instead, letting others know your thoughts. If you're mostly certain, but not completely certain, you might also try asking on IRC to see if someone else can confirm your suspicions.

- **Wait for feedback, and respond to feedback that you receive**

  Focus on one or two tickets, see them through from start to finish, and repeat. The shotgun approach of taking on lots of tickets and letting some fall by the wayside ends up doing more harm than good.

- **Be rigorous**

  When we say "**PEP 8**, and must have docs and tests", we mean it. If a patch doesn't have docs and tests, there had better be a good reason. Arguments like "I couldn't find any existing tests of this feature" don't carry much weight–while it may be true, that means you have the extra-important job of writing the very first tests for that feature, not that you get a pass from writing tests altogether.

FAQ

1. **This ticket I care about has been ignored for days/weeks/months! What can I do to get it committed?**

   First off, it's not personal. Django is entirely developed by volunteers (even the core team), and sometimes folks just don't have time. The best thing to do is to send a gentle reminder to the *django-developers* mailing list asking for review on the ticket, or to bring it up in the #django-dev IRC channel.

2. **I'm sure my ticket is absolutely 100% perfect, can I mark it as RFC myself?**

   Short answer: No. It's always better to get another set of eyes on a ticket. If you're having trouble getting that second set of eyes, see question 1, above.

10.1.2 Reporting bugs and requesting features

: Please report security issues **only** to security@djangoproject.com. This is a private list only open to long-time, highly trusted Django developers, and its archives are not public. For further details, please see our security policies.

Otherwise, before reporting a bug or requesting a new feature, please consider these general points:

- Check that someone hasn't already filed the bug or feature request by searching or running custom queries in the ticket tracker.
- Don't use the ticket system to ask support questions. Use the *django-users* list or the #django IRC channel for that.
- Don't reopen issues that have been marked "wontfix" by a core developer. This mark means that the decision has been made that we can't or won't fix this particular issue. If you're not sure why, please ask on *django-developers*.
- Don't use the ticket tracker for lengthy discussions, because they're likely to get lost. If a particular ticket is controversial, please move the discussion to *django-developers*.

Reporting bugs

Well-written bug reports are *incredibly* helpful. However, there's a certain amount of overhead involved in working with any bug tracking system so your help in keeping our ticket tracker as useful as possible is appreciated. In particular:

- **Do** read the FAQ to see if your issue might be a well-known question.
- **Do** ask on *django-users* or #django *first* if you're not sure if what you're seeing is a bug.
- **Do** write complete, reproducible, specific bug reports. You must include a clear, concise description of the problem, and a set of instructions for replicating it. Add as much debug information as you can: code snippets, test cases, exception backtraces, screenshots, etc. A nice small test case is the best way to report a bug, as it gives us an easy way to confirm the bug quickly.
- **Don't** post to *django-developers* just to announce that you have filed a bug report. All the tickets are mailed to another list, *django-updates*, which is tracked by developers and interested community members; we see them as they are filed.

To understand the lifecycle of your ticket once you have created it, refer to Triaging tickets.

Reporting user interface bugs and features

If your bug or feature request touches on anything visual in nature, there are a few additional guidelines to follow:

- Include screenshots in your ticket which are the visual equivalent of a minimal testcase. Show off the issue, not the crazy customizations you've made to your browser.
- If the issue is difficult to show off using a still image, consider capturing a *brief* screencast. If your software permits it, capture only the relevant area of the screen.
- If you're offering a patch which changes the look or behavior of Django's UI, you **must** attach before *and* after screenshots/screencasts. Tickets lacking these are difficult for triagers and core developers to assess quickly.
- Screenshots don't absolve you of other good reporting practices. Make sure to include URLs, code snippets, and step-by-step instructions on how to reproduce the behavior visible in the screenshots.
- Make sure to set the UI/UX flag on the ticket so interested parties can find your ticket.

Requesting features

We're always trying to make Django better, and your feature requests are a key part of that. Here are some tips on how to make a request most effectively:

- Make sure the feature actually requires changes in Django's core. If your idea can be developed as an independent application or module — for instance, you want to support another database engine — we'll probably suggest that you to develop it independently. Then, if your project gathers sufficient community support, we may consider it for inclusion in Django.
- First request the feature on the *django-developers* list, not in the ticket tracker. It'll get read more closely if it's on the mailing list. This is even more important for large-scale feature requests. We like to discuss any big changes to Django's core on the mailing list before actually working on them.
- Describe clearly and concisely what the missing feature is and how you'd like to see it implemented. Include example code (non-functional is OK) if possible.
- Explain *why* you'd like the feature. In some cases this is obvious, but since Django is designed to help real developers get real work done, you'll need to explain it, if it isn't obvious why the feature would be useful.

If core developers agree on the feature, then it's appropriate to create a ticket. Include a link the discussion on *django-developers* in the ticket description.

As with most open-source projects, code talks. If you are willing to write the code for the feature yourself or, even better, if you've already written it, it's much more likely to be accepted. Just fork Django on GitHub, create a feature branch, and show us your work!

See also: *Documenting new features*.

How we make decisions

Whenever possible, we strive for a rough consensus. To that end, we'll often have informal votes on *django-developers* about a feature. In these votes we follow the voting style invented by Apache and used on Python itself, where votes are given as +1, +0, -0, or -1. Roughly translated, these votes mean:

- +1: "I love the idea and I'm strongly committed to it."
- +0: "Sounds OK to me."
- -0: "I'm not thrilled, but I won't stand in the way."
- -1: "I strongly disagree and would be very unhappy to see the idea turn into reality."

Although these votes on *django-developers* are informal, they'll be taken very seriously. After a suitable voting period, if an obvious consensus arises we'll follow the votes.

However, consensus is not always possible. If consensus cannot be reached, or if the discussion towards a consensus fizzles out without a concrete decision, any *core team member* may defer the decision to the *technical board*.

Internally, the technical board will use the same voting mechanism. A proposition will be considered carried if:

- There are at least three "+1" votes from members of the technical board.
- There is no "-1" vote from any member of the technical board.

Votes should be submitted within a week.

Since this process allows any technical board member to veto a proposal, a "-1" vote should be accompanied by an explanation of what it would take to convert that "-1" into at least a "+0".

Votes on technical matters should be announced and held in public on the *django-developers* mailing list.

10.1.3 Triaging tickets

Django uses Trac for managing the work on the code base. Trac is a community-tended garden of the bugs people have found and the features people would like to see added. As in any garden, sometimes there are weeds to be pulled and sometimes there are flowers and vegetables that need picking. We need your help to sort out one from the other, and in the end we all benefit together.

Like all gardens, we can aspire to perfection but in reality there's no such thing. Even in the most pristine garden there are still snails and insects. In a community garden there are also helpful people who – with the best of intentions – fertilize the weeds and poison the roses. It's the job of the community as a whole to self-manage, keep the problems to a minimum, and educate those coming into the community so that they can become valuable contributing members.

Similarly, while we aim for Trac to be a perfect representation of the state of Django's progress, we acknowledge that this simply will not happen. By distributing the load of Trac maintenance to the community, we accept that there will be mistakes. Trac is "mostly accurate", and we give allowances for the fact that sometimes it will be wrong. That's okay. We're perfectionists with deadlines.

We rely on the community to keep participating, keep tickets as accurate as possible, and raise issues for discussion on our mailing lists when there is confusion or disagreement.

Django is a community project, and every contribution helps. We can't do this without **you**!

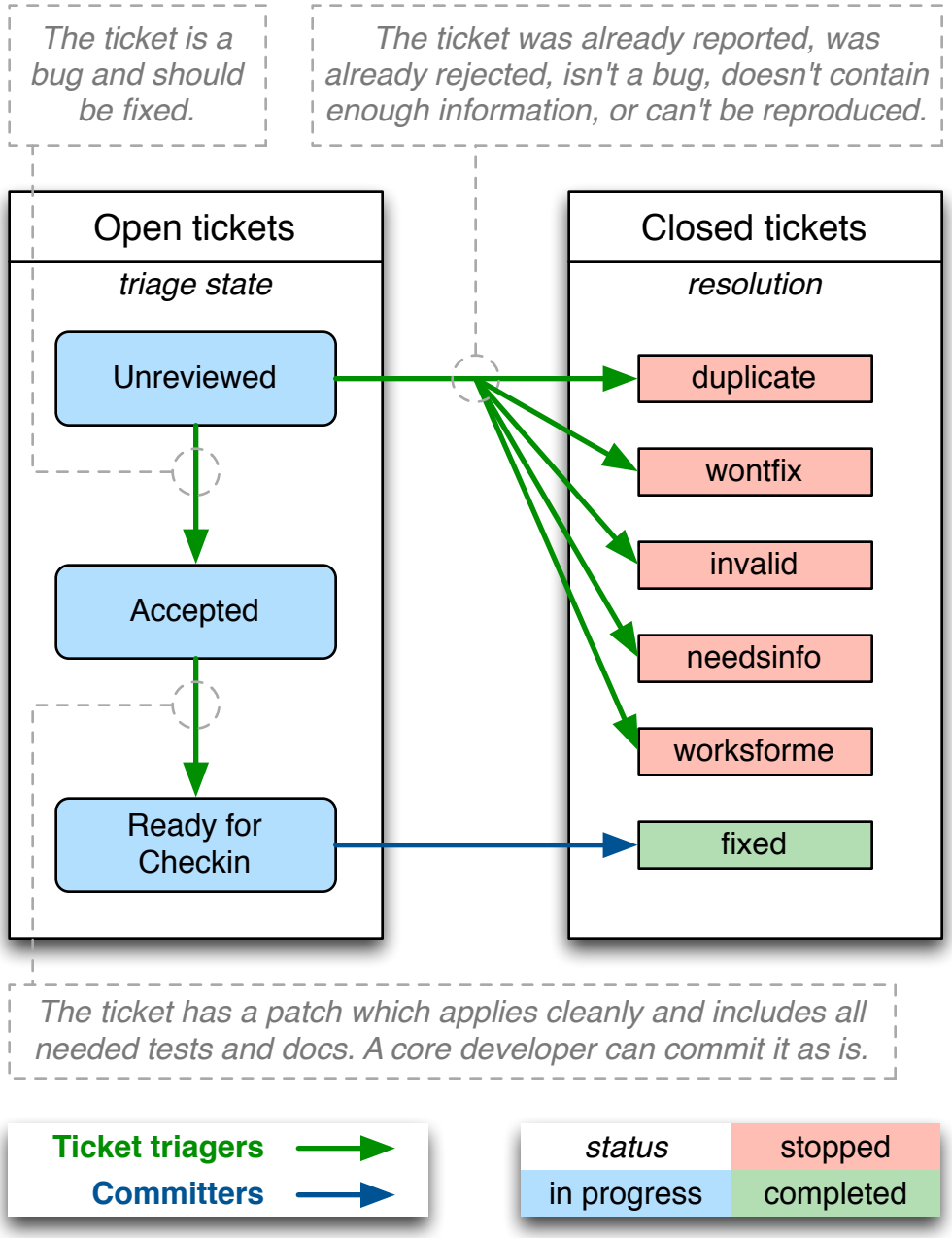
Triage workflow

Unfortunately, not all bug reports and feature requests in the ticket tracker provide all the required details. A number of tickets have patches, but those patches don't meet all the requirements of a *good patch*.

One way to help out is to *triage* tickets that have been created by other users. The core team and several community members work on this regularly, but more help is always appreciated.

Most of the workflow is based around the concept of a ticket's *triage stages*. Each stage describes where in its lifetime a given ticket is at any time. Along with a handful of flags, this attribute easily tells us what and who each ticket is waiting on.

Since a picture is worth a thousand words, let's start there:



We've got two roles in this diagram:

- Committers (also called core developers): people with commit access who are responsible for making decisions and integrating the contributions of the community.
- Ticket triagers: anyone in the Django community who chooses to become involved in Django's development process. Our Trac installation is intentionally left open to the public, and anyone can triage tickets. Django is a community project, and we encourage *triage by the community*.

By way of example, here we see the lifecycle of an average ticket:

- Alice creates a ticket, and uploads an incomplete patch (no tests, incorrect implementation).
- Bob reviews the patch, marks it "Accepted", "needs tests", and "patch needs improvement", and leaves a com-

ment telling Alice how the patch could be improved.

- Alice updates the patch, adding tests (but not changing the implementation). She removes the two flags.
- Charlie reviews the patch and resets the "patch needs improvement" flag with another comment about improving the implementation.
- Alice updates the patch, fixing the implementation. She removes the "patch needs improvement" flag.
- Daisy reviews the patch, and marks it RFC.
- Jacob, a core developer, reviews the RFC patch, applies it to his checkout, and commits it.

Some tickets require much less feedback than this, but then again some tickets require much much more.

Triage stages

Below we describe in more detail the various stages that a ticket may flow through during its lifetime.

Unreviewed

The ticket has not been reviewed by anyone who felt qualified to make a judgment about whether the ticket contained a valid issue, a viable feature, or ought to be closed for any of the various reasons.

Accepted

The big gray area! The absolute meaning of "accepted" is that the issue described in the ticket is valid and is in some stage of being worked on. Beyond that there are several considerations:

- **Accepted + No Flags**

  The ticket is valid, but no one has submitted a patch for it yet. Often this means you could safely start writing a patch for it. This is generally more true for the case of accepted bugs than accepted features. A ticket for a bug that has been accepted means that the issue has been verified by at least one triager as a legitimate bug - and should probably be fixed if possible. An accepted new feature may only mean that one triager thought the feature would be good to have, but this alone does not represent a consensus view or imply with any certainty that a patch will be accepted for that feature. Seek more feedback before writing an extensive patch if you are in doubt.

- **Accepted + Has Patch**

  The ticket is waiting for people to review the supplied patch. This means downloading the patch and trying it out, verifying that it contains tests and docs, running the test suite with the included patch, and leaving feedback on the ticket.

- **Accepted + Has Patch + Needs ...**

  This means the ticket has been reviewed, and has been found to need further work. "Needs tests" and "Needs documentation" are self-explanatory. "Patch needs improvement" will generally be accompanied by a comment on the ticket explaining what is needed to improve the code.

Ready For Checkin

The ticket was reviewed by any member of the community other than the person who supplied the patch and found to meet all the requirements for a commit-ready patch. A committer now needs to give the patch a final review prior to being committed. See the *New contributors' FAQ* for "My ticket has been in RFC forever! What should I do?"

Someday/Maybe

This stage isn't shown on the diagram. It's only used by core developers to keep track of high-level ideas or long term feature requests.

These tickets are uncommon and overall less useful since they don't describe concrete actionable issues. They are enhancement requests that we might consider adding someday to the framework if an excellent patch is submitted. They are not a high priority.

Other triage attributes

A number of flags, appearing as checkboxes in Trac, can be set on a ticket:

Has patch

This means the ticket has an associated patch. These will be reviewed to see if the patch is "good".

The following three fields (Needs documentation, Needs tests, Patch needs improvement) apply only if a patch has been supplied.

Needs documentation

This flag is used for tickets with patches that need associated documentation. Complete documentation of features is a prerequisite before we can check them into the codebase.

Needs tests

This flags the patch as needing associated unit tests. Again, this is a required part of a valid patch.

Patch needs improvement

This flag means that although the ticket *has* a patch, it's not quite ready for checkin. This could mean the patch no longer applies cleanly, there is a flaw in the implementation, or that the code doesn't meet our standards.

Easy pickings

Tickets that would require small, easy, patches.

Type

Tickets should be categorized by *type* between:

- **New Feature** For adding something new.
- **Bug** For when an existing thing is broken or not behaving as expected.
- **Cleanup/optimization** For when nothing is broken but something could be made cleaner, better, faster, stronger.

Component

Tickets should be classified into *components* indicating which area of the Django codebase they belong to. This makes tickets better organized and easier to find.

Severity

The *severity* attribute is used to identify blockers, that is, issues which should get fixed before releasing the next version of Django. Typically those issues are bugs causing regressions from earlier versions or potentially causing severe data losses. This attribute is quite rarely used and the vast majority of tickets have a severity of "Normal".

Version

It is possible to use the *version* attribute to indicate in which version the reported bug was identified.

UI/UX

This flag is used for tickets that relate to User Interface and User Experiences questions. For example, this flag would be appropriate for user-facing features in forms or the admin interface.

Cc

You may add your username or email address to this field to be notified when new contributions are made to the ticket.

Keywords

With this field you may label a ticket with multiple keywords. This can be useful, for example, to group several tickets of a same theme. Keywords can either be comma or space separated. Keyword search finds the keyword string anywhere in the keywords. For example, clicking on a ticket with the keyword "form" will yield similar tickets tagged with keywords containing strings such as "formset", "modelformset", and "ManagementForm".

Closing Tickets

When a ticket has completed its useful lifecycle, it's time for it to be closed. Closing a ticket is a big responsibility, though. You have to be sure that the issue is really resolved, and you need to keep in mind that the reporter of the ticket may not be happy to have their ticket closed (unless it's fixed, of course). If you're not certain about closing a ticket, just leave a comment with your thoughts instead.

If you do close a ticket, you should always make sure of the following:

- Be certain that the issue is resolved.
- Leave a comment explaining the decision to close the ticket.
- If there is a way they can improve the ticket to reopen it, let them know.
- If the ticket is a duplicate, reference the original ticket. Also cross-reference the closed ticket by leaving a comment in the original one – this allows to access more related information about the reported bug or requested feature.

- **Be polite.** No one likes having their ticket closed. It can be frustrating or even discouraging. The best way to avoid turning people off from contributing to Django is to be polite and friendly and to offer suggestions for how they could improve this ticket and other tickets in the future.

A ticket can be resolved in a number of ways:

- **fixed** Used by the core developers once a patch has been rolled into Django and the issue is fixed.
- **invalid** Used if the ticket is found to be incorrect. This means that the issue in the ticket is actually the result of a user error, or describes a problem with something other than Django, or isn't a bug report or feature request at all (for example, some new users submit support queries as tickets).
- **wontfix** Used when a core developer decides that this request is not appropriate for consideration in Django. This is usually chosen after discussion in the *django-developers* mailing list. Feel free to start or join in discussions of "wontfix" tickets on the *django-developers* mailing list, but please do not reopen tickets closed as "wontfix" by a core developer.
- **duplicate** Used when another ticket covers the same issue. By closing duplicate tickets, we keep all the discussion in one place, which helps everyone.
- **worksforme** Used when the ticket doesn't contain enough detail to replicate the original bug.
- **needsinfo** Used when the ticket does not contain enough information to replicate the reported issue but is potentially still valid. The ticket should be reopened when more information is supplied.

If you believe that the ticket was closed in error – because you're still having the issue, or it's popped up somewhere else, or the triagers have made a mistake – please reopen the ticket and provide further information. Again, please do not reopen tickets that have been marked as "wontfix" by core developers and bring the issue to *django-developers* instead.

How can I help with triaging?

The triage process is primarily driven by community members. Really, **ANYONE** can help.

Core developers may provide feedback on issues they're familiar with, or make decisions on controversial ones, but they aren't responsible for triaging tickets in general.

To get involved, start by creating an account on Trac. If you have an account but have forgotten your password, you can reset it using the password reset page.

Then, you can help out by:

- Closing "Unreviewed" tickets as "invalid", "worksforme" or "duplicate."
- Closing "Unreviewed" tickets as "needsinfo" when the description is too sparse to be actionable, or when they're feature requests requiring a discussion on *django-developers*.
- Correcting the "Needs tests", "Needs documentation", or "Has patch" flags for tickets where they are incorrectly set.
- Setting the "Easy pickings" flag for tickets that are small and relatively straightforward.
- Set the *type* of tickets that are still uncategorized.
- Checking that old tickets are still valid. If a ticket hasn't seen any activity in a long time, it's possible that the problem has been fixed but the ticket hasn't yet been closed.
- Identifying trends and themes in the tickets. If there are a lot of bug reports about a particular part of Django, it may indicate we should consider refactoring that part of the code. If a trend is emerging, you should raise it for discussion (referencing the relevant tickets) on *django-developers*.

- Verify if patches submitted by other users are correct. If they are correct and also contain appropriate documentation and tests then move them to the "Ready for Checkin" stage. If they are not correct then leave a comment to explain why and set the corresponding flags ("Patch needs improvement", "Needs tests" etc.).

: The Reports page contains links to many useful Trac queries, including several that are useful for triaging tickets and reviewing patches as suggested above.

You can also find more Advice for new contributors.

However, we do ask the following of all general community members working in the ticket database:

- Please **don't** close tickets as "wontfix." The core developers will make the final determination of the fate of a ticket, usually after consultation with the community.
- Please **don't** promote your own tickets to "Ready for checkin". You may mark other people's tickets which you've reviewed as "Ready for checkin", but you should get at minimum one other community member to review a patch that you submit.
- Please **don't** reverse a decision that has been made by a core developer. If you disagree with a decision that has been made, please post a message to *django-developers*.
- If you're unsure if you should be making a change, don't make the change but instead leave a comment with your concerns on the ticket, or post a message to *django-developers*. It's okay to be unsure, but your input is still valuable.

Bisecting a regression

A regression is a bug that's present in some newer version of Django but not in an older one. An extremely helpful piece of information is the commit that introduced the regression. Knowing the commit that caused the change in behavior helps identify if the change was intentional or if it was an inadvertent side-effect. Here's how you can determine this.

Begin by writing a regression test for Django's test suite for the issue. For example, we'll pretend we're debugging a regression in migrations. After you've written the test and confirmed that it fails on the latest master, put it in a separate file that you can run standalone. For our example, we'll pretend we created `tests/migrations/test_regression.py`, which can be run with:

```
$ ./runtests.py migrations.test_regression
```

Next, we mark the current point in history as being "bad" since the test fails:

```
$ git bisect bad
You need to start by "git bisect start"
Do you want me to do it for you [Y/n]? y
```

Now, we need to find a point in git history before the regression was introduced (i.e. a point where the test passes). Use something like `git checkout HEAD~100` to checkout an earlier revision (100 commits earlier, in this case). Check if the test fails. If so, mark that point as "bad" (`git bisect bad`), then checkout an earlier revision and recheck. Once you find a revision where your test passes, mark it as "good":

```
$ git bisect good
Bisecting: X revisions left to test after this (roughly Y steps)
...
```

Now we're ready for the fun part: using `git bisect run` to automate the rest of the process:

```
$ git bisect run python runtests.py migrations.test_regression
```

You should see `git bisect` use a binary search to automatically checkout revisions between the good and bad commits until it finds the first "bad" commit where the test fails.

Now, report your results on the Trac ticket, and please include the regression test as an attachment. When someone writes a fix for the bug, they'll already have your test as a starting point.

10.1.4 Writing code

So you'd like to write some code to improve Django. Awesome! Browse the following sections to find out how to give your code patches the best chances to be included in Django core:

Coding style

Please follow these coding standards when writing code for inclusion in Django.

Python style

- Please conform to the indentation style dictated in the `.editorconfig` file. We recommend using a text editor with EditorConfig support to avoid indentation and whitespace issues. The Python files use 4 spaces for indentation and the HTML files use 2 spaces.
- Unless otherwise specified, follow **PEP 8**.

  Use flake8 to check for problems in this area. Note that our `setup.cfg` file contains some excluded files (deprecated modules we don't care about cleaning up and some third-party code that Django vendors) as well as some excluded errors that we don't consider as gross violations. Remember that **PEP 8** is only a guide, so respect the style of the surrounding code as a primary goal.

  An exception to **PEP 8** is our rules on line lengths. Don't limit lines of code to 79 characters if it means the code looks significantly uglier or is harder to read. We allow up to 119 characters as this is the width of GitHub code review; anything longer requires horizontal scrolling which makes review more difficult. This check is included when you run `flake8`. Documentation, comments, and docstrings should be wrapped at 79 characters, even though **PEP 8** suggests 72.
- Use four spaces for indentation.
- Use underscores, not camelCase, for variable, function and method names (i.e. `poll.get_unique_voters()`, not `poll.getUniqueVoters`).
- Use `InitialCaps` for class names (or for factory functions that return classes).
- In docstrings, follow **PEP 257**. For example:

```
def foo():
    """
    Calculate something and return the result.
    """
    ...
```

- In tests, use *`assertRaisesMessage()`* instead of `assertRaises()` so you can check the exception message. Use `assertRaisesRegex()` (`six.assertRaisesRegex()` as long as we support Python 2) only if you need to use regular expression matching.

Imports

- Use isort to automate import sorting using the guidelines below.

  Quick start:

```
$ pip install isort
$ isort -rc .
```

  This runs `isort` recursively from your current directory, modifying any files that don't conform to the guidelines. If you need to have imports out of order (to avoid a circular import, for example) use a comment like this:

```
import module  # isort:skip
```

- Put imports in these groups: future, standard library, third-party libraries, other Django components, local Django component, try/excepts. Sort lines in each group alphabetically by the full module name. Place all `import module` statements before `from module import objects` in each section. Use absolute imports for other Django components and relative imports for local components.
- On each line, alphabetize the items with the upper case items grouped before the lower case items.
- Break long lines using parentheses and indent continuation lines by 4 spaces. Include a trailing comma after the last import and put the closing parenthesis on its own line.

  Use a single blank line between the last import and any module level code, and use two blank lines above the first function or class.

  For example (comments are for explanatory purposes only):

  `django/contrib/admin/example.py`

```
# future
from __future__ import unicode_literals

# standard library
import json
from itertools import chain

# third-party
import bcrypt

# Django
from django.http import Http404
from django.http.response import (
    Http404, HttpResponse, HttpResponseNotAllowed, StreamingHttpResponse,
    cookie,
)

# local Django
from .models import LogEntry

# try/except
try:
    import pytz
except ImportError:
    pytz = None

CONSTANT = 'foo'
```

```
    class Example(object):
        # ...
```

- Use convenience imports whenever available. For example, do this:

```
    from django.views import View
```

  instead of:

```
    from django.views.generic.base import View
```

Template style

- In Django template code, put one (and only one) space between the curly brackets and the tag contents.

  Do this:

```
    {{ foo }}
```

  Don't do this:

```
    {{foo}}
```

View style

- In Django views, the first parameter in a view function should be called `request`.

  Do this:

```
    def my_view(request, foo):
        # ...
```

  Don't do this:

```
    def my_view(req, foo):
        # ...
```

Model style

- Field names should be all lowercase, using underscores instead of camelCase.

  Do this:

```
    class Person(models.Model):
        first_name = models.CharField(max_length=20)
        last_name = models.CharField(max_length=40)
```

  Don't do this:

```
    class Person(models.Model):
        FirstName = models.CharField(max_length=20)
        Last_Name = models.CharField(max_length=40)
```

- The `class Meta` should appear *after* the fields are defined, with a single blank line separating the fields and the class definition.

  Do this:

```
class Person(models.Model):
    first_name = models.CharField(max_length=20)
    last_name = models.CharField(max_length=40)

    class Meta:
        verbose_name_plural = 'people'
```

Don't do this:

```
class Person(models.Model):
    first_name = models.CharField(max_length=20)
    last_name = models.CharField(max_length=40)
    class Meta:
        verbose_name_plural = 'people'
```

Don't do this, either:

```
class Person(models.Model):
    class Meta:
        verbose_name_plural = 'people'

    first_name = models.CharField(max_length=20)
    last_name = models.CharField(max_length=40)
```

- If you define a `__str__` method (previously `__unicode__` before Python 3 was supported), decorate the model class with *`python_2_unicode_compatible()`*.
- The order of model inner classes and standard methods should be as follows (noting that these are not all required):
  - All database fields
  - Custom manager attributes
  - `class Meta`
  - `def __str__()`
  - `def save()`
  - `def get_absolute_url()`
  - Any custom methods
- If `choices` is defined for a given model field, define each choice as a tuple of tuples, with an all-uppercase name as a class attribute on the model. Example:

```
class MyModel(models.Model):
    DIRECTION_UP = 'U'
    DIRECTION_DOWN = 'D'
    DIRECTION_CHOICES = (
        (DIRECTION_UP, 'Up'),
        (DIRECTION_DOWN, 'Down'),
    )
```

Use of `django.conf.settings`

Modules should not in general use settings stored in `django.conf.settings` at the top level (i.e. evaluated when the module is imported). The explanation for this is as follows:

Manual configuration of settings (i.e. not relying on the DJANGO_SETTINGS_MODULE environment variable) is allowed and possible as follows:

```
from django.conf import settings

settings.configure({}, SOME_SETTING='foo')
```

However, if any setting is accessed before the settings.configure line, this will not work. (Internally, settings is a LazyObject which configures itself automatically when the settings are accessed if it has not already been configured).

So, if there is a module containing some code as follows:

```
from django.conf import settings
from django.urls import get_callable

default_foo_view = get_callable(settings.FOO_VIEW)
```

...then importing this module will cause the settings object to be configured. That means that the ability for third parties to import the module at the top level is incompatible with the ability to configure the settings object manually, or makes it very difficult in some circumstances.

Instead of the above code, a level of laziness or indirection must be used, such as django.utils.functional.LazyObject, django.utils.functional.lazy() or lambda.

Miscellaneous

- Mark all strings for internationalization; see the i18n documentation for details.
- Remove import statements that are no longer used when you change code. flake8 will identify these imports for you. If an unused import needs to remain for backwards-compatibility, mark the end of with # NOQA to silence the flake8 warning.
- Systematically remove all trailing whitespaces from your code as those add unnecessary bytes, add visual clutter to the patches and can also occasionally cause unnecessary merge conflicts. Some IDE's can be configured to automatically remove them and most VCS tools can be set to highlight them in diff outputs.
- Please don't put your name in the code you contribute. Our policy is to keep contributors' names in the AUTHORS file distributed with Django – not scattered throughout the codebase itself. Feel free to include a change to the AUTHORS file in your patch if you make more than a single trivial change.

JavaScript style

For details about the JavaScript code style used by Django, see JavaScript.

Unit tests

Django comes with a test suite of its own, in the tests directory of the code base. It's our policy to make sure all tests pass at all times.

We appreciate any and all contributions to the test suite!

The Django tests all use the testing infrastructure that ships with Django for testing applications. See Writing and running tests for an explanation of how to write new tests.

Running the unit tests

Quickstart First, fork Django on GitHub.

Second, create and activate a virtual environment. If you're not familiar with how to do that, read our contributing tutorial.

Next, clone your fork, install some requirements, and run the tests:

```
$ git clone git@github.com:YourGitHubName/django.git django-repo
$ cd django-repo/tests
$ pip install -e ..
$ pip install -r requirements/py3.txt  # Python 2: py2.txt
$ ./runtests.py
```

Installing the requirements will likely require some operating system packages that your computer doesn't have installed. You can usually figure out which package to install by doing a Web search for the last line or so of the error message. Try adding your operating system to the search query if needed.

If you have trouble installing the requirements, you can skip that step, except on Python 2, where you must `pip install mock`. See *Running all the tests* for details on installing the optional test dependencies. If you don't have an optional dependency installed, the tests that require it will be skipped.

Running the tests requires a Django settings module that defines the databases to use. To make it easy to get started, Django provides and uses a sample settings module that uses the SQLite database. See *Using another settings module* to learn how to use a different settings module to run the tests with a different database.

Windows users

We recommend something like Git Bash to run the tests using the above approach.

Having problems? See *Troubleshooting* for some common issues.

Using another `settings` module The included settings module (`tests/test_sqlite.py`) allows you to run the test suite using SQLite. If you want to run the tests using a different database, you'll need to define your own settings file. Some tests, such as those for `contrib.postgres`, are specific to a particular database backend and will be skipped if run with a different backend.

To run the tests with different settings, ensure that the module is on your `PYTHONPATH` and pass the module with `--settings`.

The *DATABASES* setting in any test settings module needs to define two databases:

- A `default` database. This database should use the backend that you want to use for primary testing.
- A database with the alias `other`. The `other` database is used to test that queries can be directed to different databases. This database should use the same backend as the `default`, and it must have a different name.

If you're using a backend that isn't SQLite, you will need to provide other details for each database:

- The *USER* option needs to specify an existing user account for the database. That user needs permission to execute `CREATE DATABASE` so that the test database can be created.
- The *PASSWORD* option needs to provide the password for the *USER* that has been specified.

Test databases get their names by prepending `test_` to the value of the *NAME* settings for the databases defined in *DATABASES*. These test databases are deleted when the tests are finished.

You will also need to ensure that your database uses UTF-8 as the default character set. If your database server doesn't use UTF-8 as a default charset, you will need to include a value for *CHARSET* in the test settings dictionary for the applicable database.

Running only some of the tests Django's entire test suite takes a while to run, and running every single test could be redundant if, say, you just added a test to Django that you want to run quickly without running everything else. You can run a subset of the unit tests by appending the names of the test modules to `runtests.py` on the command line.

For example, if you'd like to run tests only for generic relations and internationalization, type:

```
$ ./runtests.py --settings=path.to.settings generic_relations i18n
```

How do you find out the names of individual tests? Look in `tests/` — each directory name there is the name of a test.

If you just want to run a particular class of tests, you can specify a list of paths to individual test classes. For example, to run the `TranslationTests` of the `i18n` module, type:

```
$ ./runtests.py --settings=path.to.settings i18n.tests.TranslationTests
```

Going beyond that, you can specify an individual test method like this:

```
$ ./runtests.py --settings=path.to.settings i18n.tests.TranslationTests.test_lazy_objects
```

Running the Selenium tests Some tests require Selenium and a Web browser. To run these tests, you must install the selenium package and run the tests with the `--selenium=<BROWSERS>` option. For example, if you have Firefox and Google Chrome installed:

```
$ ./runtests.py --selenium=firefox,chrome
```

See the selenium.webdriver package for the list of available browsers.

Specifying `--selenium` automatically sets `--tags=selenium` to run only the tests that require selenium.

Running all the tests If you want to run the full suite of tests, you'll need to install a number of dependencies:

- argon2-cffi 16.1.0+
- bcrypt
- docutils
- enum34 (Python 2 only)
- geoip2
- jinja2 2.7+
- numpy
- Pillow
- PyYAML
- pytz
- setuptools
- memcached, plus a *supported Python binding*
- mock (for Python 2)

- gettext (*gettext on Windows*)
- selenium
- sqlparse

You can find these dependencies in pip requirements files inside the `tests/requirements` directory of the Django source tree and install them like so:

```
$ pip install -r tests/requirements/py3.txt  # Python 2: py2.txt
```

If you encounter an error during the installation, your system might be missing a dependency for one or more of the Python packages. Consult the failing package's documentation or search the Web with the error message that you encounter.

You can also install the database adapter(s) of your choice using `oracle.txt`, `mysql.txt`, or `postgres.txt`.

If you want to test the memcached cache backend, you'll also need to define a *CACHES* setting that points at your memcached instance.

To run the GeoDjango tests, you will need to setup a spatial database and install the Geospatial libraries.

Each of these dependencies is optional. If you're missing any of them, the associated tests will be skipped.

Code coverage Contributors are encouraged to run coverage on the test suite to identify areas that need additional tests. The coverage tool installation and use is described in *testing code coverage*.

Coverage should be run in a single process to obtain accurate statistics. To run coverage on the Django test suite using the standard test settings:

```
$ coverage run ./runtests.py --settings=test_sqlite --parallel=1
```

After running coverage, generate the html report by running:

```
$ coverage html
```

When running coverage for the Django tests, the included `.coveragerc` settings file defines `coverage_html` as the output directory for the report and also excludes several directories not relevant to the results (test code or external code included in Django).

Contrib apps

Tests for contrib apps can be found in the `tests/` directory, typically under `<app_name>_tests`. For example, tests for `contrib.auth` are located in `tests/auth_tests`.

Troubleshooting

Many test failures with `UnicodeEncodeError` If the `locales` package is not installed, some tests will fail with a `UnicodeEncodeError`.

You can resolve this on Debian-based systems, for example, by running:

```
$ apt-get install locales
$ dpkg-reconfigure locales
```

Tests that only fail in combination In case a test passes when run in isolation but fails within the whole suite, we have some tools to help analyze the problem.

The `--bisect` option of `runtests.py` will run the failing test while halving the test set it is run together with on each iteration, often making it possible to identify a small number of tests that may be related to the failure.

For example, suppose that the failing test that works on its own is `ModelTest.test_eq`, then using:

```
$ ./runtests.py --bisect basic.tests.ModelTest.test_eq
```

will try to determine a test that interferes with the given one. First, the test is run with the first half of the test suite. If a failure occurs, the first half of the test suite is split in two groups and each group is then run with the specified test. If there is no failure with the first half of the test suite, the second half of the test suite is run with the specified test and split appropriately as described earlier. The process repeats until the set of failing tests is minimized.

The `--pair` option runs the given test alongside every other test from the suite, letting you check if another test has side-effects that cause the failure. So:

```
$ ./runtests.py --pair basic.tests.ModelTest.test_eq
```

will pair `test_eq` with every test label.

With both `--bisect` and `--pair`, if you already suspect which cases might be responsible for the failure, you may limit tests to be cross-analyzed by *specifying further test labels* after the first one:

```
$ ./runtests.py --pair basic.tests.ModelTest.test_eq queries transactions
```

You can also try running any set of tests in reverse using the `--reverse` option in order to verify that executing tests in a different order does not cause any trouble:

```
$ ./runtests.py basic --reverse
```

Seeing the SQL queries run during a test If you wish to examine the SQL being run in failing tests, you can turn on *SQL logging* using the `--debug-sql` option. If you combine this with `--verbosity=2`, all SQL queries will be output:

```
$ ./runtests.py basic --debug-sql
```

Seeing the full traceback of a test failure By default tests are run in parallel with one process per core. When the tests are run in parallel, however, you'll only see a truncated traceback for any test failures. You can adjust this behavior with the `--parallel` option:

```
$ ./runtests.py basic --parallel=1
```

You can also use the `DJANGO_TEST_PROCESSES` environment variable for this purpose.

Support for running tests in parallel and the `--parallel` option were added.

Tips for writing tests

Isolating model registration To avoid polluting the global *apps* registry and prevent unnecessary table creation, models defined in a test method should be bound to a temporary `Apps` instance:

```
from django.apps.registry import Apps
from django.db import models
from django.test import SimpleTestCase
```

```
class TestModelDefinition(SimpleTestCase):
    def test_model_definition(self):
        test_apps = Apps(['app_label'])

        class TestModel(models.Model):
            class Meta:
                apps = test_apps
        ...
```

django.test.utils.**isolate_apps**(**app_labels*, *attr_name=None*, *kwarg_name=None*)

Since this pattern involves a lot of boilerplate, Django provides the *isolate_apps()* decorator. It's used like this:

```
from django.db import models
from django.test import SimpleTestCase
from django.test.utils import isolate_apps

class TestModelDefinition(SimpleTestCase):
    @isolate_apps('app_label')
    def test_model_definition(self):
        class TestModel(models.Model):
            pass
        ...
```

Setting app_label

Models defined in a test method with no explicit *app_label* are automatically assigned the label of the app in which their test class is located.

In order to make sure the models defined within the context of *isolate_apps()* instances are correctly installed, you should pass the set of targeted app_label as arguments:

tests/app_label/tests.py

```
from django.db import models
from django.test import SimpleTestCase
from django.test.utils import isolate_apps

class TestModelDefinition(SimpleTestCase):
    @isolate_apps('app_label', 'other_app_label')
    def test_model_definition(self):
        # This model automatically receives app_label='app_label'
        class TestModel(models.Model):
            pass

        class OtherAppModel(models.Model):
            class Meta:
                app_label = 'other_app_label'
        ...
```

The decorator can also be applied to classes:

```
from django.db import models
from django.test import SimpleTestCase
from django.test.utils import isolate_apps

@isolate_apps('app_label')
class TestModelDefinition(SimpleTestCase):
```

```
    def test_model_definition(self):
        class TestModel(models.Model):
            pass
        ...
```

The temporary `Apps` instance used to isolate model registration can be retrieved as an attribute when used as a class decorator by using the `attr_name` parameter:

```
from django.db import models
from django.test import SimpleTestCase
from django.test.utils import isolate_apps

@isolate_apps('app_label', attr_name='apps')
class TestModelDefinition(SimpleTestCase):
    def test_model_definition(self):
        class TestModel(models.Model):
            pass
        self.assertIs(self.apps.get_model('app_label', 'TestModel'), TestModel)
```

Or as an argument on the test method when used as a method decorator by using the `kwarg_name` parameter:

```
from django.db import models
from django.test import SimpleTestCase
from django.test.utils import isolate_apps

class TestModelDefinition(SimpleTestCase):
    @isolate_apps('app_label', kwarg_name='apps')
    def test_model_definition(self, apps):
        class TestModel(models.Model):
            pass
        self.assertIs(apps.get_model('app_label', 'TestModel'), TestModel)
```

Submitting patches

We're always grateful for patches to Django's code. Indeed, bug reports with associated patches will get fixed *far* more quickly than those without patches.

Typo fixes and trivial documentation changes

If you are fixing a really trivial issue, for example changing a word in the documentation, the preferred way to provide the patch is using GitHub pull requests without a Trac ticket.

See the Working with Git and GitHub for more details on how to use pull requests.

"Claiming" tickets

In an open-source project with hundreds of contributors around the world, it's important to manage communication efficiently so that work doesn't get duplicated and contributors can be as effective as possible.

Hence, our policy is for contributors to "claim" tickets in order to let other developers know that a particular bug or feature is being worked on.

If you have identified a contribution you want to make and you're capable of fixing it (as measured by your coding ability, knowledge of Django internals and time availability), claim it by following these steps:

- Login using your GitHub account or create an account in our ticket system. If you have an account but have forgotten your password, you can reset it using the password reset page.
- If a ticket for this issue doesn't exist yet, create one in our ticket tracker.
- If a ticket for this issue already exists, make sure nobody else has claimed it. To do this, look at the "Owned by" section of the ticket. If it's assigned to "nobody," then it's available to be claimed. Otherwise, somebody else may be working on this ticket. Either find another bug/feature to work on, or contact the developer working on the ticket to offer your help. If a ticket has been assigned for weeks or months without any activity, it's probably safe to reassign it to yourself.
- Log into your account, if you haven't already, by clicking "GitHub Login" or "DjangoProject Login" in the upper left of the ticket page.
- Claim the ticket by clicking the "assign to myself" radio button under "Action" near the bottom of the page, then click "Submit changes."

: The Django software foundation requests that anyone contributing more than a trivial patch to Django sign and submit a Contributor License Agreement, this ensures that the Django Software Foundation has clear license to all contributions allowing for a clear license for all users.

Ticket claimers' responsibility Once you've claimed a ticket, you have a responsibility to work on that ticket in a reasonably timely fashion. If you don't have time to work on it, either unclaim it or don't claim it in the first place!

If there's no sign of progress on a particular claimed ticket for a week or two, another developer may ask you to relinquish the ticket claim so that it's no longer monopolized and somebody else can claim it.

If you've claimed a ticket and it's taking a long time (days or weeks) to code, keep everybody updated by posting comments on the ticket. If you don't provide regular updates, and you don't respond to a request for a progress report, your claim on the ticket may be revoked.

As always, more communication is better than less communication!

Which tickets should be claimed? Of course, going through the steps of claiming tickets is overkill in some cases.

In the case of small changes, such as typos in the documentation or small bugs that will only take a few minutes to fix, you don't need to jump through the hoops of claiming tickets. Just submit your patch and be done with it.

Of course, it is *always* acceptable, regardless whether someone has claimed it or not, to submit patches to a ticket if you happen to have a patch ready.

Patch style

Make sure that any contribution you do fulfills at least the following requirements:

- The code required to fix a problem or add a feature is an essential part of a patch, but it is not the only part. A good patch should also include a regression test to validate the behavior that has been fixed and to prevent the problem from arising again. Also, if some tickets are relevant to the code that you've written, mention the ticket numbers in some comments in the test so that one can easily trace back the relevant discussions after your patch gets committed, and the tickets get closed.
- If the code associated with a patch adds a new feature, or modifies behavior of an existing feature, the patch should also contain documentation.

When you think your work is ready to be reviewed, send a GitHub pull request. Please review the patch yourself using our *patch review checklist* first.

If you can't send a pull request for some reason, you can also use patches in Trac. When using this style, follow these guidelines.

- Submit patches in the format returned by the `git diff` command.
- Attach patches to a ticket in the ticket tracker, using the "attach file" button. Please *don't* put the patch in the ticket description or comment unless it's a single line patch.
- Name the patch file with a `.diff` extension; this will let the ticket tracker apply correct syntax highlighting, which is quite helpful.

Regardless of the way you submit your work, follow these steps.

- Make sure your code fulfills the requirements in our *patch review checklist*.
- Check the "Has patch" box on the ticket and make sure the "Needs documentation", "Needs tests", and "Patch needs improvement" boxes aren't checked. This makes the ticket appear in the "Patches needing review" queue on the Development dashboard.

Non-trivial patches

A "non-trivial" patch is one that is more than a simple bug fix. It's a patch that introduces Django functionality and makes some sort of design decision.

If you provide a non-trivial patch, include evidence that alternatives have been discussed on *django-developers*.

If you're not sure whether your patch should be considered non-trivial, just ask.

Deprecating a feature

There are a couple reasons that code in Django might be deprecated:

- If a feature has been improved or modified in a backwards-incompatible way, the old feature or behavior will be deprecated.
- Sometimes Django will include a backport of a Python library that's not included in a version of Python that Django currently supports. When Django no longer needs to support the older version of Python that doesn't include the library, the library will be deprecated in Django.

As the *deprecation policy* describes, the first release of Django that deprecates a feature (`A.B`) should raise a `RemovedInDjangoXXWarning` (where XX is the Django version where the feature will be removed) when the deprecated feature is invoked. Assuming we have good test coverage, these warnings are converted to errors when *running the test suite* with warnings enabled: `python -Wall runtests.py`. Thus, when adding a `RemovedInDjangoXXWarning` you need to eliminate or silence any warnings generated when running the tests.

The first step is to remove any use of the deprecated behavior by Django itself. Next you can silence warnings in tests that actually test the deprecated behavior by using the `ignore_warnings` decorator, either at the test or class level:

1. In a particular test:

```
from django.test import ignore_warnings
from django.utils.deprecation import RemovedInDjangoXXWarning

@ignore_warnings(category=RemovedInDjangoXXWarning)
def test_foo(self):
    ...
```

2. For an entire test case:

```
    from django.test import ignore_warnings
    from django.utils.deprecation import RemovedInDjangoXXWarning

    @ignore_warnings(category=RemovedInDjangoXXWarning)
    class MyDeprecatedTests(unittest.TestCase):
        ...
```

You can also add a test for the deprecation warning. You'll have to disable the "warning as error" behavior in your test by doing:

```
import warnings

def test_foo_deprecation_warning(self):
    with warnings.catch_warnings(record=True) as warns:
        warnings.simplefilter('always')  # prevent warnings from appearing as errors
        # invoke deprecated behavior

    self.assertEqual(len(warns), 1)
    msg = str(warns[0].message)
    self.assertEqual(msg, 'Expected deprecation message')
```

Finally, there are a couple of updates to Django's documentation to make:

1. If the existing feature is documented, mark it deprecated in documentation using the `.. deprecated:: A.B` annotation. Include a short description and a note about the upgrade path if applicable.
2. Add a description of the deprecated behavior, and the upgrade path if applicable, to the current release notes (`docs/releases/A.B.txt`) under the "Features deprecated in A.B" heading.
3. Add an entry in the deprecation timeline (`docs/internals/deprecation.txt`) under the appropriate version describing what code will be removed.

Once you have completed these steps, you are finished with the deprecation. In each *feature release*, all `RemovedInDjangoXXWarning`s matching the new version are removed.

JavaScript patches

For information on JavaScript patches, see the *JavaScript patches* documentation.

Patch review checklist

Use this checklist to review a pull request. If you are reviewing a pull request that is not your own and it passes all the criteria below, please set the "Triage Stage" on the corresponding Trac ticket to "Ready for checkin". If you've left comments for improvement on the pull request, please tick the appropriate flags on the Trac ticket based on the results of your review: "Patch needs improvement", "Needs documentation", and/or "Needs tests". As time and interest permits, core developers do final reviews of "Ready for checkin" tickets and will either commit the patch or bump it back to "Accepted" if further works need to be done. If you're looking to become a core developer, doing thorough reviews of patches is a great way to earn trust.

Looking for a patch to review? Check out the "Patches needing review" section of the Django Development Dashboard. Looking to get your patch reviewed? Ensure the Trac flags on the ticket are set so that the ticket appears in that queue.

Documentation

- Does the documentation build without any errors (`make html`, or `make.bat html` on Windows, from the `docs` directory)?

- Does the documentation follow the writing style guidelines in Writing documentation?
- Are there any *spelling errors*?

Bugs

- Is there a proper regression test (the test should fail before the fix is applied)?

New Features

- Are there tests to "exercise" all of the new code?
- Is there a release note in `docs/releases/A.B.txt`?
- Is there documentation for the feature and is it *annotated appropriately* with `.. versionadded:: A.B` or `.. versionchanged:: A.B`?

Deprecating a feature See the *Deprecating a feature* guide.

All code changes

- Does the coding style conform to our guidelines? Are there any `flake8` errors?
- If the change is backwards incompatible in any way, is there a note in the release notes (`docs/releases/A.B.txt`)?
- Is Django's test suite passing? Ask in `#django-dev` for a core dev to build the pull request against our continuous integration server.

All tickets

- Is the pull request a single squashed commit with a message that follows our *commit message format*?
- Are you the patch author and a new contributor? Please add yourself to the `AUTHORS` file and submit a Contributor License Agreement.

Working with Git and GitHub

This section explains how the community can contribute code to Django via pull requests. If you're interested in how core developers handle them, see Committing code.

Below, we are going to show how to create a GitHub pull request containing the changes for Trac ticket #xxxxx. By creating a fully-ready pull request, you will make the reviewer's job easier, meaning that your work is more likely to be merged into Django.

You could also upload a traditional patch to Trac, but it's less practical for reviews.

Installing Git

Django uses Git for its source control. You can download Git, but it's often easier to install with your operating system's package manager.

Django's Git repository is hosted on GitHub, and it is recommended that you also work using GitHub.

After installing Git, the first thing you should do is setup your name and email:

```
$ git config --global user.name "Your Real Name"
$ git config --global user.email "you@email.com"
```

Note that `user.name` should be your real name, not your GitHub nick. GitHub should know the email you use in the `user.email` field, as this will be used to associate your commits with your GitHub account.

Setting up local repository

When you have created your GitHub account, with the nick "GitHub_nick", and forked Django's repository, create a local copy of your fork:

```
git clone git@github.com:GitHub_nick/django.git
```

This will create a new directory "django", containing a clone of your GitHub repository. The rest of the git commands on this page need to be run within the cloned directory, so switch to it now:

```
cd django
```

Your GitHub repository will be called "origin" in Git.

You should also setup django/django as an "upstream" remote (that is, tell git that the reference Django repository was the source of your fork of it):

```
git remote add upstream git@github.com:django/django.git
git fetch upstream
```

You can add other remotes similarly, for example:

```
git remote add akaariai git@github.com:akaariai/django.git
```

Working on a ticket

When working on a ticket, create a new branch for the work, and base that work on upstream/master:

```
git checkout -b ticket_xxxxx upstream/master
```

The -b flag creates a new branch for you locally. Don't hesitate to create new branches even for the smallest things - that's what they are there for.

If instead you were working for a fix on the 1.4 branch, you would do:

```
git checkout -b ticket_xxxxx_1_4 upstream/stable/1.4.x
```

Assume the work is carried on the ticket_xxxxx branch. Make some changes and commit them:

```
git commit
```

When writing the commit message, follow the *commit message guidelines* to ease the work of the committer. If you're uncomfortable with English, try at least to describe precisely what the commit does.

If you need to do additional work on your branch, commit as often as necessary:

```
git commit -m 'Added two more tests for edge cases'
```

Publishing work You can publish your work on GitHub just by doing:

```
git push origin ticket_xxxxx
```

When you go to your GitHub page, you will notice a new branch has been created.

If you are working on a Trac ticket, you should mention in the ticket that your work is available from branch ticket_xxxxx of your GitHub repo. Include a link to your branch.

Note that the above branch is called a "topic branch" in Git parlance. You are free to rewrite the history of this branch, by using `git rebase` for example. Other people shouldn't base their work on such a branch, because their clone would become corrupt when you edit commits.

There are also "public branches". These are branches other people are supposed to fork, so the history of these branches should never change. Good examples of public branches are the `master` and `stable/A.B.x` branches in the django/django repository.

When you think your work is ready to be pulled into Django, you should create a pull request at GitHub. A good pull request means:

- commits with one logical change in each, following the coding style,
- well-formed messages for each commit: a summary line and then paragraphs wrapped at 72 characters thereafter – see the *committing guidelines* for more details,
- documentation and tests, if needed – actually tests are always needed, except for documentation changes.

The test suite must pass and the documentation must build without warnings.

Once you have created your pull request, you should add a comment in the related Trac ticket explaining what you've done. In particular, you should note the environment in which you ran the tests, for instance: "all tests pass under SQLite and MySQL".

Pull requests at GitHub have only two states: open and closed. The committer who will deal with your pull request has only two options: merge it or close it. For this reason, it isn't useful to make a pull request until the code is ready for merging – or sufficiently close that a committer will finish it himself.

Rebasing branches In the example above, you created two commits, the "Fixed ticket_xxxxx" commit and "Added two more tests" commit.

We do not want to have the entire history of your working process in your repository. Your commit "Added two more tests" would be unhelpful noise. Instead, we would rather only have one commit containing all your work.

To rework the history of your branch you can squash the commits into one by using interactive rebase:

```
git rebase -i HEAD~2
```

The HEAD~2 above is shorthand for two latest commits. The above command will open an editor showing the two commits, prefixed with the word "pick".

Change "pick" on the second line to "squash" instead. This will keep the first commit, and squash the second commit into the first one. Save and quit the editor. A second editor window should open, so you can reword the commit message for the commit now that it includes both your steps.

You can also use the "edit" option in rebase. This way you can change a single commit, for example to fix a typo in a docstring:

```
git rebase -i HEAD~3
# Choose edit, pick, pick for the commits
# Now you are able to rework the commit (use git add normally to add changes)
# When finished, commit work with "--amend" and continue
git commit --amend
# Reword the commit message if needed
```

```
git rebase --continue
# The second and third commits should be applied.
```

If your topic branch is already published at GitHub, for example if you're making minor changes to take into account a review, you will need to force-push the changes:

```
git push -f origin ticket_xxxxx
```

Note that this will rewrite history of ticket_xxxxx - if you check the commit hashes before and after the operation at GitHub you will notice that the commit hashes do not match anymore. This is acceptable, as the branch is merely a topic branch, and nobody should be basing their work on it.

After upstream has changed When upstream (django/django) has changed, you should rebase your work. To do this, use:

```
git fetch upstream
git rebase
```

The work is automatically rebased using the branch you forked on, in the example case using upstream/master.

The rebase command removes all your local commits temporarily, applies the upstream commits, and then applies your local commits again on the work.

If there are merge conflicts, you will need to resolve them and then use `git rebase --continue`. At any point you can use `git rebase --abort` to return to the original state.

Note that you want to *rebase* on upstream, not *merge* the upstream.

The reason for this is that by rebasing, your commits will always be *on top of* the upstream's work, not *mixed in with* the changes in the upstream. This way your branch will contain only commits related to its topic, which makes squashing easier.

After review It is unusual to get any non-trivial amount of code into core without changes requested by reviewers. In this case, it is often a good idea to add the changes as one incremental commit to your work. This allows the reviewer to easily check what changes you have done.

In this case, do the changes required by the reviewer. Commit as often as necessary. Before publishing the changes, rebase your work. If you added two commits, you would run:

```
git rebase -i HEAD~2
```

Squash the second commit into the first. Write a commit message along the lines of:

```
Made changes asked in review by <reviewer>

- Fixed whitespace errors in foobar
- Reworded the docstring of bar()
```

Finally, push your work back to your GitHub repository. Since you didn't touch the public commits during the rebase, you should not need to force-push:

```
git push origin ticket_xxxxx
```

Your pull request should now contain the new commit too.

Note that the committer is likely to squash the review commit into the previous commit when committing the code.

Working on a patch

One of the ways that developers can contribute to Django is by reviewing patches. Those patches will typically exist as pull requests on GitHub and can be easily integrated into your local repository:

```
git checkout -b pull_xxxxx upstream/master
curl https://github.com/django/django/pull/xxxxx.patch | git am
```

This will create a new branch and then apply the changes from the pull request to it. At this point you can run the tests or do anything else you need to do to investigate the quality of the patch.

For more detail on working with pull requests see the *guidelines for committers*.

Summary

- Work on GitHub if you can.
- Announce your work on the Trac ticket by linking to your GitHub branch.
- When you have something ready, make a pull request.
- Make your pull requests as good as you can.
- When doing fixes to your work, use `git rebase -i` to squash the commits.
- When upstream has changed, do `git fetch upstream; git rebase`.

JavaScript

While most of Django core is Python, the `admin` and `gis` contrib apps contain JavaScript code.

Please follow these coding standards when writing JavaScript code for inclusion in Django.

Code style

- Please conform to the indentation style dictated in the `.editorconfig` file. We recommend using a text editor with EditorConfig support to avoid indentation and whitespace issues. Most of the JavaScript files use 4 spaces for indentation, but there are some exceptions.
- When naming variables, use `camelCase` instead of `underscore_case`. Different JavaScript files sometimes use a different code style. Please try to conform to the code style of each file.
- Use the JSHint code linter to check your code for bugs and style errors. JSHint will be run when you run the JavaScript tests. We also recommended installing a JSHint plugin in your text editor.

JavaScript patches

Django's admin system leverages the jQuery framework to increase the capabilities of the admin interface. In conjunction, there is an emphasis on admin JavaScript performance and minimizing overall admin media file size. Serving compressed or "minified" versions of JavaScript files is considered best practice in this regard.

To that end, patches for JavaScript files should include both the original code for future development (e.g. `foo.js`), and a compressed version for production use (e.g. `foo.min.js`). Any links to the file in the codebase should point to the compressed version.

Compressing JavaScript To simplify the process of providing optimized JavaScript code, Django includes a handy Python script which should be used to create a "minified" version. To run it:

```
$ pip install closure
$ python django/contrib/admin/bin/compress.py
```

Behind the scenes, `compress.py` is a front-end for Google's Closure Compiler which is written in Java. The Closure Compiler library is not bundled with Django, but you can install it using pip as done above. The Closure Compiler library requires Java 7 or higher.

Please don't forget to run `compress.py` and include the `diff` of the minified scripts when submitting patches for Django's JavaScript.

JavaScript tests

Django's JavaScript tests can be run in a browser or from the command line. The tests are located in a top level `js_tests` directory.

Writing tests Django's JavaScript tests use QUnit. Here is an example test module:

```
module('magicTricks', {
    beforeEach: function() {
        var $ = django.jQuery;
        $('#qunit-fixture').append('<button class="button"></button>');
    }
});

test('removeOnClick removes button on click', function(assert) {
    var $ = django.jQuery;
    removeOnClick('.button');
    assert.equal($('.button').length === 1);
    $('.button').click();
    assert.equal($('.button').length === 0);
});

test('copyOnClick adds button on click', function(assert) {
    var $ = django.jQuery;
    copyOnClick('.button');
    assert.equal($('.button').length === 1);
    $('.button').click();
    assert.equal($('.button').length === 2);
});
```

Please consult the QUnit documentation for information on the types of assertions supported by QUnit.

Running tests The JavaScript tests may be run from a web browser or from the command line.

Testing from a web browser To run the tests from a web browser, open up `js_tests/tests.html` in your browser.

To measure code coverage when running the tests, you need to view that file over HTTP. To view code coverage:

- Execute `python -m http.server` (or `python -m SimpleHTTPServer` on Python 2) from the root directory (not from inside `js_tests`).
- Open http://localhost:8000/js_tests/tests.html in your web browser.

Testing from the command line To run the tests from the command line, you need to have Node.js installed.

After installing *Node.js*, install the JavaScript test dependencies by running the following from the root of your Django checkout:

```
$ npm install
```

Then run the tests with:

```
$ npm test
```

10.1.5 Writing documentation

We place a high importance on consistency and readability of documentation. After all, Django was created in a journalism environment! So we treat our documentation like we treat our code: we aim to improve it as often as possible.

Documentation changes generally come in two forms:

- General improvements: typo corrections, error fixes and better explanations through clearer writing and more examples.
- New features: documentation of features that have been added to the framework since the last release.

This section explains how writers can craft their documentation changes in the most useful and least error-prone ways.

Getting the raw documentation

Though Django's documentation is intended to be read as HTML at https://docs.djangoproject.com/, we edit it as a collection of text files for maximum flexibility. These files live in the top-level `docs/` directory of a Django release.

If you'd like to start contributing to our docs, get the development version of Django from the source code repository (see *Installing the development version*). The development version has the latest-and-greatest documentation, just as it has latest-and-greatest code. We also backport documentation fixes and improvements, at the discretion of the committer, to the last release branch. That's because it's highly advantageous to have the docs for the last release be up-to-date and correct (see *Differences between versions*).

Getting started with Sphinx

Django's documentation uses the Sphinx documentation system, which in turn is based on docutils. The basic idea is that lightly-formatted plain-text documentation is transformed into HTML, PDF, and any other output format.

To actually build the documentation locally, you'll currently need to install Sphinx – `pip install Sphinx` should do the trick.

Then, building the HTML is easy; just `make html` (or `make.bat html` on Windows) from the `docs` directory.

To get started contributing, you'll want to read the reStructuredText Primer. After that, you'll want to read about the Sphinx-specific markup that's used to manage metadata, indexing, and cross-references.

How the documentation is organized

The documentation is organized into several categories:

- Tutorials take the reader by the hand through a series of steps to create something.

  The important thing in a tutorial is to help the reader achieve something useful, preferably as early as possible, in order to give them confidence.

  Explain the nature of the problem we're solving, so that the reader understands what we're trying to achieve. Don't feel that you need to begin with explanations of how things work - what matters is what the reader does, not what you explain. It can be helpful to refer back to what you've done and explain afterwards.

- Topic guides aim to explain a concept or subject at a fairly high level.

  Link to reference material rather than repeat it. Use examples and don't be reluctant to explain things that seem very basic to you - it might be the explanation someone else needs.

  Providing background context helps a newcomer connect the topic to things that they already know.

- Reference guides contain technical reference for APIs. They describe the functioning of Django's internal machinery and instruct in its use.

  Keep reference material tightly focused on the subject. Assume that the reader already understands the basic concepts involved but needs to know or be reminded of how Django does it.

  Reference guides aren't the place for general explanation. If you find yourself explaining basic concepts, you may want to move that material to a topic guide.

- How-to guides are recipes that take the reader through steps in key subjects.

  What matters most in a how-to guide is what a user wants to achieve. A how-to should always be result-oriented rather than focused on internal details of how Django implements whatever is being discussed.

  These guides are more advanced than tutorials and assume some knowledge about how Django works. Assume that the reader has followed the tutorials and don't hesitate to refer the reader back to the appropriate tutorial rather than repeat the same material.

Writing style

When using pronouns in reference to a hypothetical person, such as "a user with a session cookie", gender neutral pronouns (they/their/them) should be used. Instead of:

- he or she... use they.
- him or her... use them.
- his or her... use their.
- his or hers... use theirs.
- himself or herself... use themselves.

Commonly used terms

Here are some style guidelines on commonly used terms throughout the documentation:

- **Django** – when referring to the framework, capitalize Django. It is lowercase only in Python code and in the djangoproject.com logo.
- **email** – no hyphen.
- **MySQL**, **PostgreSQL**, **SQLite**
- **SQL** – when referring to SQL, the expected pronunciation should be "Ess Queue Ell" and not "sequel". Thus in a phrase like "Returns an SQL expression", "SQL" should be preceded by "an" and not "a".

- **Python** – when referring to the language, capitalize Python.
- **realize**, **customize**, **initialize**, etc. – use the American "ize" suffix, not "ise."
- **subclass** – it's a single word without a hyphen, both as a verb ("subclass that model") and as a noun ("create a subclass").
- **Web**, **World Wide Web**, **the Web** – note Web is always capitalized when referring to the World Wide Web.
- **website** – use one word, without capitalization.

Django-specific terminology

- **model** – it's not capitalized.
- **template** – it's not capitalized.
- **URLconf** – use three capitalized letters, with no space before "conf."
- **view** – it's not capitalized.

Guidelines for reStructuredText files

These guidelines regulate the format of our reST (reStructuredText) documentation:

- In section titles, capitalize only initial words and proper nouns.
- Wrap the documentation at 80 characters wide, unless a code example is significantly less readable when split over two lines, or for another good reason.
- The main thing to keep in mind as you write and edit docs is that the more semantic markup you can add the better. So:

```
Add ``django.contrib.auth`` to your ``INSTALLED_APPS``...
```

  Isn't nearly as helpful as:

```
Add :mod:`django.contrib.auth` to your :setting:`INSTALLED_APPS`...
```

  This is because Sphinx will generate proper links for the latter, which greatly helps readers.

  You can prefix the target with a ~ (that's a tilde) to get just the "last bit" of that path. So `:mod:`~django.contrib.auth`` will just display a link with the title "auth".

- Use `intersphinx` to reference Python's and Sphinx' documentation.
- Add `.. code-block:: <lang>` to literal blocks so that they get highlighted. Prefer relying on automatic highlighting simply using `::` (two colons). This has the benefit that if the code contains some invalid syntax, it won't be highlighted. Adding `.. code-block:: python`, for example, will force highlighting despite invalid syntax.
- Use these heading styles:

```
===
One
===

Two
===

Three
-----
```

```
    Four
    ~~~~

    Five
    ^^^^
```

Django-specific markup

Besides the Sphinx built-in markup, Django's docs defines some extra description units:

- Settings:

```
    .. setting:: INSTALLED_APPS
```

  To link to a setting, use `:setting:'INSTALLED_APPS'`.

- Template tags:

```
    .. templatetag:: regroup
```

  To link, use `:ttag:'regroup'`.

- Template filters:

```
    .. templatefilter:: linebreaksbr
```

  To link, use `:tfilter:'linebreaksbr'`.

- Field lookups (i.e. `Foo.objects.filter(bar__exact=whatever)`):

```
    .. fieldlookup:: exact
```

  To link, use `:lookup:'exact'`.

- `django-admin` commands:

```
    .. django-admin:: migrate
```

  To link, use `:djadmin:'migrate'`.

- `django-admin` command-line options:

```
    .. django-admin-option:: --traceback
```

  To link, use `:option:'command_name --traceback'` (or omit `command_name` for the options shared by all commands like `--verbosity`).

- Links to Trac tickets (typically reserved for patch release notes):

```
    :ticket:`12345`
```

Documenting new features

Our policy for new features is:

> All documentation of new features should be written in a way that clearly designates the features are only available in the Django development version. Assume documentation readers are using the latest release, not the development version.

Our preferred way for marking new features is by prefacing the features' documentation with: "`.. versionadded:: X.Y`", followed by a mandatory blank line and an optional description (indented).

General improvements, or other changes to the APIs that should be emphasized should use the "`.. versionchanged:: X.Y`" directive (with the same format as the `versionadded` mentioned above.

These `versionadded` and `versionchanged` blocks should be "self-contained." In other words, since we only keep these annotations around for two releases, it's nice to be able to remove the annotation and its contents without having to reflow, reindent, or edit the surrounding text. For example, instead of putting the entire description of a new or changed feature in a block, do something like this:

```
.. class:: Author(first_name, last_name, middle_name=None)

    A person who writes books.

    ``first_name`` is ...

    ...

    ``middle_name`` is ...

    .. versionchanged:: A.B

        The ``middle_name`` argument was added.
```

Put the changed annotation notes at the bottom of a section, not the top.

Also, avoid referring to a specific version of Django outside a `versionadded` or `versionchanged` block. Even inside a block, it's often redundant to do so as these annotations render as "New in Django A.B:" and "Changed in Django A.B", respectively.

If a function, attribute, etc. is added, it's also okay to use a `versionadded` annotation like this:

```
.. attribute:: Author.middle_name

    .. versionadded:: A.B

    An author's middle name.
```

We can simply remove the `.. versionadded:: A.B` annotation without any indentation changes when the time comes.

Minimizing images

Optimize image compression where possible. For PNG files, use OptiPNG and AdvanceCOMP's `advpng`:

```
$ cd docs/
$ optipng -o7 -zm1-9 -i0 -strip all `find . -type f -not -path "./_build/*" -name "*.png"`
$ advpng -z4 `find . -type f -not -path "./_build/*" -name "*.png"`
```

This is based on OptiPNG version 0.7.5. Older versions may complain about the `--strip all` option being lossy.

An example

For a quick example of how it all fits together, consider this hypothetical example:

- First, the `ref/settings.txt` document could have an overall layout like this:

```
========
Settings
========

...

.. _available-settings:

Available settings
==================

...

.. _deprecated-settings:

Deprecated settings
===================

...
```

- Next, the `topics/settings.txt` document could contain something like this:

```
You can access a :ref:`listing of all available settings
<available-settings>`. For a list of deprecated settings see
:ref:`deprecated-settings`.

You can find both in the :doc:`settings reference document
</ref/settings>`.
```

  We use the Sphinx `doc` cross reference element when we want to link to another document as a whole and the `ref` element when we want to link to an arbitrary location in a document.

- Next, notice how the settings are annotated:

```
.. setting:: ADMINS

ADMINS
======

Default: ``[]`` (Empty list)

A list of all the people who get code error notifications. When
``DEBUG=False`` and a view raises an exception, Django will email these people
with the full exception information. Each member of the list should be a tuple
of (Full name, email address). Example::

    [('John', 'john@example.com'), ('Mary', 'mary@example.com')]

Note that Django will email *all* of these people whenever an error happens.
See :doc:`/howto/error-reporting` for more information.
```

  This marks up the following header as the "canonical" target for the setting `ADMINS`. This means any time I talk about `ADMINS`, I can reference it using `:setting:`ADMINS``.

That's basically how everything fits together.

Spelling check

Before you commit your docs, it's a good idea to run the spelling checker. You'll need to install a couple packages first:

- pyenchant (which requires enchant)
- sphinxcontrib-spelling

Then from the `docs` directory, run `make spelling`. Wrong words (if any) along with the file and line number where they occur will be saved to `_build/spelling/output.txt`.

If you encounter false-positives (error output that actually is correct), do one of the following:

- Surround inline code or brand/technology names with grave accents (‘).
- Find synonyms that the spell checker recognizes.
- If, and only if, you are sure the word you are using is correct - add it to `docs/spelling_wordlist` (please keep the list in alphabetical order).

Translating documentation

See *Localizing the Django documentation* if you'd like to help translate the documentation into another language.

`django-admin` man page

Sphinx can generate a manual page for the django-admin command. This is configured in `docs/conf.py`. Unlike other documentation output, this man page should be included in the Django repository and the releases as `docs/man/django-admin.1`. There isn't a need to update this file when updating the documentation, as it's updated once as part of the release process.

To generate an updated version of the man page, run `make man` in the `docs` directory. The new man page will be written in `docs/_build/man/django-admin.1`.

10.1.6 Localizing Django

Various parts of Django, such as the admin site and validation error messages, are internationalized. This means they display differently depending on each user's language or country. For this, Django uses the same internationalization and localization infrastructure available to Django applications, described in the i18n documentation.

Translations

Translations are contributed by Django users worldwide. The translation work is coordinated at Transifex.

If you find an incorrect translation or want to discuss specific translations, go to the Django project page. If you would like to help out with translating or add a language that isn't yet translated, here's what to do:

- Join the *Django i18n mailing list* and introduce yourself.
- Make sure you read the notes about *Specialties of Django translation*.
- Sign up at Transifex and visit the Django project page.
- On the Django project page, choose the language you want to work on, **or** – in case the language doesn't exist yet – request a new language team by clicking on the “Request language” link and selecting the appropriate language.

- Then, click the "Join this Team" button to become a member of this team. Every team has at least one coordinator who is responsible to review your membership request. You can of course also contact the team coordinator to clarify procedural problems and handle the actual translation process.
- Once you are a member of a team choose the translation resource you want to update on the team page. For example the "core" resource refers to the translation catalog that contains all non-contrib translations. Each of the contrib apps also have a resource (prefixed with "contrib").

: For more information about how to use Transifex, read the Transifex User Guide.

Translations from Transifex are only integrated into the Django repository at the time of a new *feature release*. We try to update them a second time during one of the following *patch release*s, but that depends on the translation manager's availability. So don't miss the string freeze period (between the release candidate and the feature release) to take the opportunity to complete and fix the translations for your language!

Formats

You can also review `conf/locale/<locale>/formats.py`. This file describes the date, time and numbers formatting particularities of your locale. See Format localization for details.

The format files aren't managed by the use of Transifex. To change them, you must create a patch against the Django source tree, as for any code change:

- Create a diff against the current Git master branch.
- Open a ticket in Django's ticket system, set its `Component` field to `Translations`, and attach the patch to it.

Documentation

There is also an opportunity to translate the documentation, though this is a huge undertaking to complete entirely (you have been warned!). We use the same Transifex tool. The translations will appear at `https://docs.djangoproject.com/<language_code>/` when at least the `docs/intro/*` files are fully translated in your language.

10.1.7 Committing code

This section is addressed to the *Committers* and to anyone interested in knowing how code gets committed into Django core. If you're a community member who wants to contribute code to Django, have a look at Working with Git and GitHub instead.

Handling pull requests

Since Django is now hosted at GitHub, most patches are provided in the form of pull requests.

When committing a pull request, make sure each individual commit matches the commit guidelines described below. Contributors are expected to provide the best pull requests possible. In practice however, committers - who will likely be more familiar with the commit guidelines - may decide to bring a commit up to standard themselves.

: Before merging, but after reviewing, have Jenkins test the pull request by commenting "buildbot, test this please" on the PR. See our Jenkins wiki page for more details.

An easy way to checkout a pull request locally is to add an alias to your `~/.gitconfig` (`upstream` is assumed to be `django/django`):

```
[alias]
    pr = !sh -c \"git fetch upstream pull/${1}/head:pr/${1} && git checkout pr/${1}\"
```

Now you can simply run `git pr ####` to checkout the corresponding pull request.

At this point, you can work on the code. Use `git rebase -i` and `git commit --amend` to make sure the commits have the expected level of quality. Once you're ready:

```
$ # Pull in the latest changes from master.
$ git checkout master
$ git pull upstream master
$ # Rebase the pull request on master.
$ git checkout pr/####
$ git rebase master
$ git checkout master
$ # Merge the work as "fast-forward" to master to avoid a merge commit.
$ # (in practice, you can omit "--ff-only" since you just rebased)
$ git merge --ff-only pr/XXXX
$ # If you're not sure if you did things correctly, check that only the
$ # changes you expect will be pushed to upstream.
$ git push --dry-run upstream master
$ # Push!
$ git push upstream master
$ # Delete the pull request branch.
$ git branch -d pr/xxxx
```

For changes on your own branches, force push to your fork after rebasing on master but before merging and pushing to upstream. This allows the commit hashes on master and your branch to match which automatically closes the pull request. Since you can't push to other contributors' branches, comment on the pull request "Merged in XXXXXXX" (replacing with the commit hash) after you merge it. Trac checks for this message format to indicate on the ticket page whether or not a pull request is merged.

Avoid using GitHub's "Merge pull request" button on the website as it creates an ugly "merge commit" and makes navigating history more difficult.

When rewriting the commit history of a pull request, the goal is to make Django's commit history as usable as possible:

- If a patch contains back-and-forth commits, then rewrite those into one. For example, if a commit adds some code and a second commit fixes stylistic issues introduced in the first commit, those commits should be squashed before merging.
- Separate changes to different commits by logical grouping: if you do a stylistic cleanup at the same time as you do other changes to a file, separating the changes into two different commits will make reviewing history easier.
- Beware of merges of upstream branches in the pull requests.
- Tests should pass and docs should build after each commit. Neither the tests nor the docs should emit warnings.
- Trivial and small patches usually are best done in one commit. Medium to large work may be split into multiple commits if it makes sense.

Practicality beats purity, so it is up to each committer to decide how much history mangling to do for a pull request. The main points are engaging the community, getting work done, and having a usable commit history.

Committing guidelines

In addition, please follow the following guidelines when committing code to Django's Git repository:

- Never change the published history of django/django branches! **Never force-push your changes to django/django.** If you absolutely must (for security reasons for example) first discuss the situation with the core team.
- For any medium-to-big changes, where "medium-to-big" is according to your judgment, please bring things up on the *django-developers* mailing list before making the change.

  If you bring something up on *django-developers* and nobody responds, please don't take that to mean your idea is great and should be implemented immediately because nobody contested it. Django's core developers don't have a lot of time to read mailing-list discussions immediately, so you may have to wait a couple of days before getting a response.
- Write detailed commit messages in the past tense, not present tense.
  - Good: "Fixed Unicode bug in RSS API."
  - Bad: "Fixes Unicode bug in RSS API."
  - Bad: "Fixing Unicode bug in RSS API."

  The commit message should be in lines of 72 chars maximum. There should be a subject line, separated by a blank line and then paragraphs of 72 char lines. The limits are soft. For the subject line, shorter is better. In the body of the commit message more detail is better than less:

```
Fixed #18307 -- Added git workflow guidelines

Refactored the Django's documentation to remove mentions of SVN
specific tasks. Added guidelines of how to use Git, GitHub, and
how to use pull request together with Trac instead.
```

  If the patch wasn't a pull request, you should credit the contributors in the commit message: "Thanks A for report, B for the patch and C for the review."
- For commits to a branch, prefix the commit message with the branch name. For example: "[1.4.x] Fixed #xxxxx – Added support for mind reading."
- Limit commits to the most granular change that makes sense. This means, use frequent small commits rather than infrequent large commits. For example, if implementing feature X requires a small change to library Y, first commit the change to library Y, then commit feature X in a separate commit. This goes a *long way* in helping all Django core developers follow your changes.
- Separate bug fixes from feature changes. Bugfixes may need to be backported to the stable branch, according to the *backwards-compatibility policy*.
- If your commit closes a ticket in the Django ticket tracker, begin your commit message with the text "Fixed #xxxxx", where "xxxxx" is the number of the ticket your commit fixes. Example: "Fixed #123 – Added whizbang feature.". We've rigged Trac so that any commit message in that format will automatically close the referenced ticket and post a comment to it with the full commit message.

  If your commit closes a ticket and is in a branch, use the branch name first, then the "Fixed #xxxxx." For example: "[1.4.x] Fixed #123 – Added whizbang feature."

  For the curious, we're using a Trac plugin for this.

: Note that the Trac integration doesn't know anything about pull requests. So if you try to close a pull request with the phrase "closes #400" in your commit message, GitHub will close the pull request, but the Trac plugin will also close the same numbered ticket in Trac.

- If your commit references a ticket in the Django ticket tracker but does *not* close the ticket, include the phrase "Refs #xxxxx", where "xxxxx" is the number of the ticket your commit references. This will automatically post

a comment to the appropriate ticket.

- Write commit messages for backports using this pattern:

```
[<Django version>] Fixed <ticket> -- <description>

Backport of <revision> from <branch>.
```

For example:

```
[1.3.x] Fixed #17028 -- Changed diveintopython.org -> diveintopython.net.

Backport of 80c0cbf1c97047daed2c5b41b296bbc56fe1d7e3 from master.
```

There's a script on the wiki to automate this.

Reverting commits

Nobody's perfect; mistakes will be committed.

But try very hard to ensure that mistakes don't happen. Just because we have a reversion policy doesn't relax your responsibility to aim for the highest quality possible. Really: double-check your work, or have it checked by another committer, **before** you commit it in the first place!

When a mistaken commit is discovered, please follow these guidelines:

- If possible, have the original author revert their own commit.
- Don't revert another author's changes without permission from the original author.
- Use git revert – this will make a reverse commit, but the original commit will still be part of the commit history.
- If the original author can't be reached (within a reasonable amount of time – a day or so) and the problem is severe – crashing bug, major test failures, etc. – then ask for objections on the *django-developers* mailing list then revert if there are none.
- If the problem is small (a feature commit after feature freeze, say), wait it out.
- If there's a disagreement between the committer and the reverter-to-be then try to work it out on the *django-developers* mailing list. If an agreement can't be reached then it should be put to a vote.
- If the commit introduced a confirmed, disclosed security vulnerability then the commit may be reverted immediately without permission from anyone.
- The release branch maintainer may back out commits to the release branch without permission if the commit breaks the release branch.
- If you mistakenly push a topic branch to django/django, just delete it. For instance, if you did: `git push upstream feature_antigravity`, just do a reverse push: `git push upstream :feature_antigravity`.

10.2 Mailing lists

: Please report security issues **only** to security@djangoproject.com. This is a private list only open to long-time, highly trusted Django developers, and its archives are not public. For further details, please see our security policies.

Django has several official mailing lists on Google Groups that are open to anyone.

10.2.1 django-users

This is the right place if you are looking to ask any question regarding the installation, usage, or debugging of Django.

: If it's the first time you send an email to this list, your email must be accepted first so don't worry if *your message does not appear* instantly.

- django-users mailing archive
- django-users subscription email address
- django-users posting email

10.2.2 django-core-mentorship

The Django Core Mentorship list is intended to provide a welcoming introductory environment for community members interested in contributing to the Django Project.

- django-core-mentorship mailing archive
- django-core-mentorship subscription email address
- django-core-mentorship posting email

10.2.3 django-developers

The discussion about the development of Django itself takes place here.

: Please make use of *django-users mailing list* if you want to ask for tech support, doing so in this list is inappropriate.

- django-developers mailing archive
- django-developers subscription email address
- django-developers posting email

10.2.4 django-i18n

This is the place to discuss the internationalization and localization of Django's components.

- django-i18n mailing archive
- django-i18n subscription email address
- django-i18n posting email

10.2.5 django-announce

A (very) low-traffic list for announcing *upcoming security releases*, new releases of Django, and security advisories.

- django-announce mailing archive
- django-announce subscription email address
- django-announce posting email

10.2.6 `django-updates`

All the ticket updates are mailed automatically to this list, which is tracked by developers and interested community members.

- django-updates mailing archive
- django-updates subscription email address
- django-updates posting email

10.3 Organization of the Django Project

10.3.1 Principles

The Django Project is managed by a team of volunteers pursuing three goals:

- Driving the development of the Django Web Framework,
- Fostering the ecosystem of Django-related software,
- Leading the Django community in accordance with the values described in the Django Code of Conduct.

The Django Project isn't a legal entity. The Django Software Foundation, a non-profit organization, handles financial and legal matters related to the Django Project. Other than that, the Django Software Foundation lets the Django Project manage the development of the Django framework, its ecosystem and its community.

The Django core team makes the decisions, nominates its new members, and elects its technical board. While it holds decision power in theory, it aims at using it as rarely as possible in practice. Rough consensus should be the norm and formal voting an exception.

10.3.2 Core team

Role

The core team is the group of trusted volunteers who manage the Django Project. They assume many roles required to achieve the project's goals, especially those that require a high level of trust. They make the decisions that shape the future of the project.

Core team members are expected to act as role models for the community and custodians of the project, on behalf of the community and all those who rely on Django.

They will intervene, where necessary, in online discussions or at official Django events on the rare occasions that a situation arises that requires intervention.

They have authority over the Django Project infrastructure, including the Django Project website itself, the Django GitHub organization and repositories, the Trac bug tracker, the mailing lists, IRC channels, etc.

Prerogatives

Core team members may participate in formal votes, typically to nominate new team members and to elect the technical board.

Some contributions don't require commit access. Depending on the reasons why a contributor joins the team, they may or may not have commit permissions to the Django code repository.

However, should the need arise, any team member may ask for commit access by writing to the core team's mailing list. Access will be granted unless the person withdraws their request or the technical board vetoes the proposal.

Core team members who have commit access are referred to as "committers" or "core developers".

Other permissions, such as access to the servers, are granted to those who need them through the same process.

Membership

The core team finds its origins with the *four people* who created Django. It has grown to *a few dozen people* by co-opting volunteers who demonstrate:

- a good grasp of the philosophy of the Django Project
- a solid track record of being constructive and helpful
- significant contributions to the project's goals, in any form
- willingness to dedicate some time to improving Django

As the project matures, contributions go way beyond code. Here's an incomplete list of areas where contributions may be considered for joining the core team, in no particular order:

- Working on community management and outreach
- Providing support on the mailing-lists and on IRC
- Triaging tickets
- Writing patches (code, docs, or tests)
- Reviewing patches (code, docs, or tests)
- Participating in design decisions
- Providing expertise in a particular domain (security, i18n, etc.)
- Managing the continuous integration infrastructure
- Managing the servers (website, tracker, documentation, etc.)
- Maintaining related projects (djangoproject.com site, ex-contrib apps, etc.)
- Creating visual designs

Very few areas are reserved to core team members:

- Reviewing security reports
- Merging patches (code, docs, or tests)
- Packaging releases

Core team membership acknowledges sustained and valuable efforts that align well with the philosophy and the goals of the Django Project.

It is granted by a four fifths majority of votes cast in a core team vote and no veto by the technical board.

Core team members are always looking for promising contributors, teaching them how the project is managed, and submitting their names to the core team's vote when they're ready. If you would like to join the core team, you can contact a core team member privately or ask for guidance on the *Django Core Mentorship mailing-list*.

There's no time limit on core team membership. However, in order to provide the general public with a reasonable idea of how many people maintain Django, core team members who have stopped contributing are encouraged to declare themselves as "past team members". Those who haven't made any non-trivial contribution in two years may be asked

to move themselves to this category, and moved there if they don't respond. Past team members lose their privileges such as voting rights and commit access.

10.3.3 Technical board

Role

The technical board is a group of experienced and active committers who steer technical choices. Their main concern is to maintain the quality and stability of the Django Web Framework.

Prerogatives

The technical board holds two prerogatives:

- Making major technical decisions when no consensus is found otherwise. This happens on the *django-developers* mailing-list.
- Veto a grant of commit access or remove commit access. This happens on the django-core mailing-list.

In both cases, the technical board is a last resort. In these matters, it fulfills a similar function to the former Benevolent Dictators For Life.

When the board wants to exercise one of these prerogatives, it must hold a private, simple majority vote on the resolution. The quorum is the full committee — each member must cast a vote or abstain explicitly. Then the board communicates the result, and if possible the reasons, on the appropriate mailing-list. There's no appeal for such decisions.

In addition, at its discretion, the technical board may act in an advisory capacity on non-technical decisions.

Membership

The technical board is an elected group of five committers. They're expected to be experienced but there's no formal seniority requirement. Its current composition is published *here*.

A new board is elected after each feature release of Django. The election process is managed by a returns officer nominated by the outgoing technical board. The election process works as follows:

1. Candidates advertise their application for the technical board to the team.

   They must be committers already. There's no term limit for technical board members.

2. Each team member can vote for zero to five people among the candidates. Candidates are ranked by the total number of votes they received.

   In case of a tie, the person who joined the core team earlier wins.

Both the application and the voting period last between one and two weeks, at the outgoing board's discretion.

10.3.4 Changing the organization

Changes to this document require a four fifths majority of votes cast in a core team vote and no veto by the technical board.

10.4 Django team

10.4.1 The original team

Django originally started at World Online, the Web department of the Lawrence Journal-World of Lawrence, Kansas, USA.

Adrian Holovaty Adrian is a Web developer with a background in journalism. He's known in journalism circles as one of the pioneers of "journalism via computer programming", and in technical circles as "the guy who invented Django."

He was lead developer at World Online for 2.5 years, during which time Django was developed and implemented on World Online's sites. He was the leader and founder of EveryBlock, a "news feed for your block." He now develops Soundslice.

Adrian lives in Chicago, USA.

Simon Willison Simon is a well-respected Web developer from England. He had a one-year internship at World Online, during which time he and Adrian developed Django from scratch. The most enthusiastic Brit you'll ever meet, he's passionate about best practices in Web development and maintains a well-read web-development blog.

Simon lives in Brighton, England.

Jacob Kaplan-Moss Jacob is Director of Platform Security at Heroku. He worked at World Online for four years, where he helped open source Django and found the Django Software Foundation. Jacob lives on a hobby farm outside of Lawrence where he spends his weekends playing with dirt and power tools.

Wilson Miner Wilson's design-fu is what makes Django look so nice. He created the design that was used for nearly the first ten years on the Django Project website, as well as the current design for Django's acclaimed admin interface. Wilson was the designer for EveryBlock and Rdio. He now designs for Facebook.

Wilson lives in San Francisco, USA.

10.4.2 The current team

These are the folks who have a long history of contributions, a solid track record of being helpful on the mailing lists, and a proven desire to dedicate serious time to Django. In return, they've been invited to join the *core team*.

Luke Plant At University Luke studied physics and Materials Science and also met Michael Meeks who introduced him to Linux and Open Source, re-igniting an interest in programming. Since then he has contributed to a number of Open Source projects and worked professionally as a developer.

Luke has contributed many excellent improvements to Django, including database-level improvements, the CSRF middleware and many unit tests.

Luke currently works for a church in Bradford, UK, and part-time as a freelance developer.

Russell Keith-Magee Russell studied physics as an undergraduate, and studied neural networks for his PhD. His first job was with a startup in the defense industry developing simulation frameworks. Over time, mostly through work with Django, he's become more involved in Web development.

Russell has helped with several major aspects of Django, including a couple major internal refactorings, creation of the test system, and more.

Russell lives in the most isolated capital city in the world — Perth, Australia.

James Bennett James has been one of Django's release managers, and also contributes to the documentation and provide the occasional bugfix.

James came to Web development from philosophy when he discovered that programmers get to argue just as much while collecting much better pay. He lives in San Mateo, California and previously worked at World Online and Mozilla; currently, he's part of the Web engineering team at Clover.

He keeps a blog, and enjoys fine port and talking to his car.

Justin Bronn Justin Bronn is a computer scientist and attorney specializing in legal topics related to intellectual property and spatial law.

In 2007, Justin began developing `django.contrib.gis` in a branch, a.k.a. GeoDjango, which was merged in time for Django 1.0. While implementing GeoDjango, Justin obtained a deep knowledge of Django's internals including the ORM, the admin, and Oracle support.

Justin lives in Houston, TX.

Karen Tracey Karen has a background in distributed operating systems (graduate school), communications software (industry) and crossword puzzle construction (freelance). The last of these brought her to Django, in late 2006, when she set out to put a Web front-end on her crossword puzzle database. That done, she stuck around in the community answering questions, debugging problems, etc. – because coding puzzles are as much fun as word puzzles.

Karen lives in Apex, NC, USA.

Jannis Leidel Jannis graduated in media design from Bauhaus-University Weimar, is the author of a number of pluggable Django apps and likes to contribute to Open Source projects like virtualenv and pip.

He has worked on Django's auth, admin and staticfiles apps as well as the form, core, internationalization and test systems. He currently works at Mozilla.

Jannis lives in Berlin, Germany.

Andrew Godwin Andrew is a freelance Python developer and tinkerer, and has been developing against Django since 2007. He graduated from Oxford University with a degree in Computer Science, and has become most well known in the Django community for his work on South, the schema migrations library.

Andrew lives in San Francisco, CA, USA.

Carl Meyer Carl has been a Django user since 2007 (long enough to remember queryset-refactor, but not magic-removal), and builds web apps at OddBird. He became a Django contributor by accident, because fixing bugs is more interesting than working around them.

Carl lives in Rapid City, SD, USA.

Ramiro Morales Ramiro has been reading Django source code and submitting patches since mid-2006 after researching for a Python Web tool with matching awesomeness and being pointed to it by an old ninja.

A software developer in the electronic transactions industry, he is a living proof of the fact that anyone with enough enthusiasm can contribute to Django, learning a lot and having fun in the process.

Ramiro lives in Córdoba, Argentina.

Chris Beaven Chris has been submitting patches and suggesting crazy ideas for Django since early 2006. An advocate for community involvement and a long-term triager, he is still often found answering questions in the #django IRC channel.

Chris lives in Napier, New Zealand (adding to the pool of Oceanic core developers). He works remotely as a developer for Lincoln Loop.

Honza Král Honza first discovered Django in 2006 and started using it right away, first for school and personal projects and later in his full-time job. He contributed various patches and fixes mostly to the newforms library, newforms admin and, through participation in the Google Summer of Code project, assisted in creating the *model validation* functionality.

He is currently working for Whiskey Media in San Francisco developing awesome sites running on pure Django.

Tim Graham When exploring Web frameworks for an independent study project in the fall of 2008, Tim discovered Django and was lured to it by the documentation. He enjoys contributing to the docs because they're awesome.

Tim works as a software engineer and lives in Philadelphia, PA, USA.

Paul McMillan Paul found Django in 2008 while looking for a more structured approach to web programming. He stuck around after figuring out that the developers of Django had already invented many of the wheels he needed. His passion for breaking (and then fixing) things led to his current role working to maintain and improve the security of Django.

Julien Phalip Julien has a background in software engineering and human-computer interaction. As a Web developer, he enjoys tinkering with the backend as much as designing and coding user interfaces. Julien discovered Django in 2007 while doing his PhD in Computing Sciences. Since then he has contributed patches to various components of the framework, in particular the admin. Julien was a co-founder of the Interaction Consortium. He now works at Odopod, a digital agency based in San Francisco, CA, USA.

Aymeric Augustin Aymeric is an engineer with a background in mathematics and computer science. He chose Django because he believes that software should be simple, explicit and tested. His perfectionist tendencies quickly led him to triage tickets and contribute patches.

Aymeric has a pragmatic approach to software engineering, can't live without a continuous integration server, and likes proving that Django is a good choice for enterprise software.

Claude Paroz Claude is a former teacher who fell in love with free software at the beginning of the 21st century. He's now working as freelancer in Web development in his native Switzerland. He has found in Django a perfect match for his needs of a stable, clean, documented and well-maintained Web framework.

He's also helping the GNOME Translation Project as maintainer of the Django-based l10n.gnome.org.

Anssi Kääriäinen Anssi works as a developer at Finnish National Institute for Health and Welfare. He is also a computer science student at Aalto University. In his work he uses Django for developing internal business applications and sees Django as a great match for that use case.

Anssi is interested in developing the object relational mapper (ORM) and all related features. He's also a fan of benchmarking and he tries keep Django as fast as possible.

Florian Apolloner Florian is currently studying Physics at the Graz University of Technology. Soon after he started using Django he joined the Ubuntuusers webteam to work on *Inyoka*, the software powering the whole Ubuntuusers site.

For the time being he lives in Graz, Austria (not Australia ;)).

Jeremy Dunck Jeremy was rescued from corporate IT drudgery by Free Software and, in part, Django. Many of Jeremy's interests center around access to information.

Jeremy was the lead developer of Pegasus News, one of the first uses of Django outside World Online, and has since joined Votizen, a startup intent on reducing the influence of money in politics.

He serves as DSF Secretary, organizes and helps organize sprints, cares about the health and equity of the Django community. He has gone an embarrassingly long time without a working blog.

Jeremy lives in Mountain View, CA, USA.

Bryan Veloso Bryan found Django 0.96 through a fellow designer who was evangelizing its use. It was his first foray outside of the land that was PHP-based templating. Although he has only ever used Django for personal projects, it is the very reason he considers himself a designer/developer hybrid and is working to further design within the Django community.

Bryan works as a designer at GitHub by day, and masquerades as a vlogger and shoutcaster in the after-hours. Bryan lives in Los Angeles, CA, USA.

Simon Charette Simon is a mathematics student who discovered Django while searching for a replacement framework to an in-house PHP entity. Since that faithful day Django has been a big part of his life. So far, he's been involved in some ORM and forms API fixes.

Apart from contributing to multiple open source projects he spends most of his spare-time playing Ultimate Frisbee and working part-time at this awesome place called Reptiletech.

Simon lives in Montréal, Québec, Canada.

Donald Stufft Donald found Python and Django in 2007 while trying to find a language, and web framework that he really enjoyed using after many years of PHP. He fell in love with the beauty of Python and the way Django made tasks simple and easy. His contributions to Django focus primarily on ensuring that it is and remains a secure web framework.

Donald currently works at Nebula Inc as a Software Engineer for their security team and lives in the Greater Philadelphia Area.

Marc Tamlyn Marc started life on the web using Django 1.2 back in 2010, and has never looked back. He was involved with rewriting the class-based view documentation at DjangoCon EU 2012, and also helped to develop CCBV, an additional class-based view reference tool.

Marc is currently a full-time parent, part-time developer, and lives in Oxford, UK.

Shai Berger Shai started working with Python back in 1998, and with Django just before 1.0. He is a Free Software enthusiast, but life happens, and he was driven by consulting gigs to contribute to the Oracle and SQL Server backends of South, and then the Oracle backend of Django itself. Finally, he joined core to help maintain the Oracle backend.

Shai works for Platonix, a small consulting company he started with a few friends in 1996, and lives near Tel Aviv, Israel.

Baptiste Mispelon Baptiste discovered Django around the 1.2 version and promptly switched away from his home-grown PHP framework. He started getting more involved in the project after attending DjangoCon EU 2012, mostly by triaging tickets and submitting small patches.

Baptiste currently lives in Budapest, Hungary and works for M2BPO, a small French company providing services to architects.

Daniele Procida Daniele unexpectedly became a Django developer on 29th April 2009. Since then he has relied daily on Django's documentation, which has been a constant companion to him. More recently he has been able to contribute back to the project by helping improve the documentation itself.

He is the author of Arkestra and Don't be afraid to commit. He lives in Cardiff, Wales, and works for Divio.

Erik Romijn Erik started using Django in the days of 1.2. His largest contribution to Django was `GenericIPAddressField`, and he has worked on all sorts of patches since. While developing with Django, he always keeps a little list of even the slightest Django frustrations, to tackle them at a later time and prevent other developers from having to deal with the same issues.

Erik is an independent app maker, mostly developing web and mobile apps, as Solid Links. He also enjoys helping ordinary developers to build safer web apps, for which Django is already a great start, and developed Erik's Pony Checkup with that goal in mind. Erik lives in Amsterdam, The Netherlands.

Loïc Bistuer Loïc studied telecommunications engineering and works as an independent software developer and consultant.

He discovered Django in 2008 shortly before the 1.0 release and has been hooked ever since. He contributes mostly to Django's ORM and Form components. His main contributions include advanced query prefetching, streamlining QuerySet and Manager to improve query reusability, and a significant refactor of forms error handling.

Loïc is originally from the South of France and currently lives in Bangkok, Thailand.

Michael Manfre Michael started running Django on Windows against a Microsoft SQL Server (MSSQL) database in 2008. He quickly became the maintainer of the django-mssql 3rd party database backend. Much of his involvement with Django relates to the ORM, the private 3rd party database API, and using Django on Windows.

Michael lives in Cary, NC, USA.

Collin Anderson Collin found Django in November 2006. He was in awe of the admin and ORM and was amazed that the documentation was teaching him best web practices like redirecting after a successful POST request. Why had he never learned this before? No one knows to this day.

He enjoys helping people on the *django-users* mailing list and making Django simple and easy for newcomers.

Collin lives in South Bend, IN, USA where he uses Django to increase unity.

Tom Christie Tom has background in speech recognition, networking, and web development. He has a particular interest in Web API design and is the original author of Django REST framework.

Tom lives in the seaside city of Brighton, UK.

Curtis Maloney Curtis is a self-taught programmer from Melbourne, Australia, who eschews specialization. Upon finding Django when it was first open sourced, he realized it was possible to enjoy web development.

He spends a lot of time helping people on the #django IRC channel, and has authored and released a number of smaller django apps.

Markus Holtermann Markus is a senior backend developer at LaterPay in Munich. He studied Computer Science at the Technical University of Berlin. He started working with Django in 2010 when he joined the ubuntuusers.de web team to work on *Inyoka*. Markus made his first contribution to the Django project during DjangoCon Europe 2013 in Warsaw. He was the web team leader for the EuroPython 2014 website and started regular contributions to Django after that.

Markus lives in Berlin, Germany.

Josh Smeaton Josh was given the opportunity to work on a new Django app around version 1.1 after working with a homegrown PHP reporting framework. The simplicity of the ORM and the power of the Admin were extremely liberating.

Still being involved with custom reporting applications, he decided to try his hand at improving the ORM support for analytics. His contributions focus on giving more power to users of the ORM.

Josh lives in Melbourne, Australia where he heads up development for a SaaS telecommunications company.

Preston Timmons Preston is a software developer with a background in mathematics. He enjoys Django because it enables consistent, simple, and tested systems to be built that even new programmers can quickly dive into. Preston lives in Dallas, TX.

Tomek Paczkowski Tomek started using Django in 2007 as a tool for quickly dealing with university projects. Since then, he worked with various technologies like Ruby on Rails, JavaScript and Android but always returned to Python and Django.

Tomek loves the Django community. He organized multiple Django sprints, co-organized DjangoCon Europe 2013 and has mentored at many Django Girls events.

Originally from Poland, Tomek currently lives in London, where he works at Squirrel.

Ola Sitarska Ola started working with Django in 2009, when she discovered the power of the Django admin and quickly fell in love with the beauty of Python.

She co-organized DjangoCon Europe 2013 in Warsaw and co-authored the Django Girls Tutorial, the most beginner friendly Django tutorial out there. Together with Ola Sendecka, she started Django Girls, a community and series of Django workshops for women who've never programmed before.

In 2015, she became a Django Software Foundation board member. Ola was also a part of the team responsible for shipping the djangoproject.com redesign.

Originally from Poland, Ola currently lives in London, where she works with friends at Potato.

Ola Sendecka Ola started her adventure with Django when writing a university project in 2009. After that she attended her first DjangoCon Europe which defined her future life as a Django professional.

Since then she has been an active community member. Ola co-organized a number of Django sprints and conferences including: DjangoCon Europe 2013, Django Under the Hood 2015 and DjangoCon Europe 2016. Together with Ola Sitarska she founded Django Girls, co-authored the Django Girls Tutorial and is a member of the Django Girls Foundation. She is also an author of the Coding is for Girls YouTube series teaching programming and Django to beginners.

Originally from Poland, Ola currently lives in London, where she works for Potato.

10.4.3 Past team members

Georg "Hugo" Bauer Georg created Django's internationalization system, managed i18n contributions and made a ton of excellent tweaks, feature additions and bug fixes.

Robert Wittams Robert was responsible for the *first* refactoring of Django's admin application to allow for easier reuse and has made a ton of excellent tweaks, feature additions and bug fixes.

Alex Gaynor Alex was involved in many parts of Django, he contributed to the ORM, forms, admin, amongst others; he is most known for his work on multiple-database support in Django.

Alex lives in Washington, DC, USA.

Simon Meers Simon discovered Django 0.96 during his Computer Science PhD research and has been developing with it full-time ever since. His core code contributions are mostly in Django's admin application.

Simon works as a freelance developer based in Wollongong, Australia.

Gabriel Hurley Gabriel has been working with Django since 2008, shortly after the 1.0 release. Convinced by his business partner that Python and Django were the right direction for the company, he couldn't have been more happy with the decision. His contributions range across many areas in Django, but years of copy-editing and an eye for detail lead him to be particularly at home while working on Django's documentation.

Gabriel works as a developer in the SF Bay Area, CA, USA.

Malcolm Tredinnick Malcolm originally wanted to be a mathematician and somehow ended up a software developer. He contributed to many Open Source projects, served on the board of the GNOME foundation, and was a great chess player.

Malcolm was deeply involved in many part of Django - most notably, the ORM, but many other internals bear his fingerprints. Django's support for unicode and autoescaping in templates can both be almost entirely attributed to Malcolm.

He was an International Man of Mystery and lived in Sydney, Australia.

Malcolm passed away on March 17, 2013.

Preston Holmes Preston is a recovering neuroscientist who originally discovered Django as part of a sweeping move to Python from a grab bag of half a dozen languages. He was drawn to Django's balance of practical batteries included philosophy, care and thought in code design, and strong open source community. Currently working in the rent-your-infra space (aka Cloud), he is always looking for opportunities to volunteer for community oriented education projects, such as for kids and scientists (e.g. Software Carpentry).

Preston lives with his family and animal menagerie in Santa Barbara, CA, USA.

Idan Gazit As a self-professed design geek, Idan was initially attracted to Django sometime between magic-removal and queryset-refactor. Formally trained as a software engineer, Idan straddles the worlds of design and code, jack of two trades and master of none. He is passionate about usability and finding novel ways to extract meaning from data, and is a longtime photographer.

Idan is currently hacking on all things data and visualization at Heroku.

Matt Boersma Matt helped with Django's Oracle support.

Ian Kelly Ian also helped with Oracle support.

Joseph Kocherhans Joseph was the director of lead development at EveryBlock and previously developed at the Lawrence Journal-World. He often disappears for several days into the woods, attempts to teach himself computational linguistics, and annoys his neighbors with his Charango playing.

Joseph's first contribution to Django was a series of improvements to the authorization system leading up to support for pluggable authorization. Since then, he's worked on the new forms system, its use in the admin, and many other smaller improvements.

Joseph lives in Chicago, USA.

Gary Wilson Gary starting contributing patches to Django in 2006 while developing Web applications for The University of Texas (UT). Since, he has made contributions to the email and forms systems, as well as many other improvements and code cleanups throughout the code base.

Gary lives in Austin, Texas, USA.

Brian Rosner Brian enjoys learning more about programming languages and system architectures and contributing to open source projects.

He helped immensely in getting Django's "newforms-admin" branch finished in time for Django 1.0.

Brian lives in Denver, Colorado, USA.

James Tauber James is the lead developer of Pinax and the CEO and founder of Eldarion. He has been doing open source software since 1993, Python since 1998 and Django since 2006. He serves on the board of the Python Software Foundation and is currently on a leave of absence from a PhD in linguistics.

James currently lives in Boston, MA, USA but originally hails from Perth, Western Australia where he attended the same high school as Russell Keith-Magee.

10.5 Roles

10.5.1 Technical board

The technical board for the 1.10 release cycle is:

- James Bennett
- Andrew Godwin
- Russell Keith-Magee
- Carl Meyer
- Marc Tamlyn

10.5.2 Committers

Most *core team* members have commit access. They're called "committers" or "core developers".

Being part of the core team is a pre-requisite for having commit access.

10.5.3 Security team

The security team is responsible for Django's security policies. It handles private reports of security issues.

The current security team members are:

- Florian Apolloner
- James Bennett
- Tim Graham
- Adrian Holovaty
- Markus Holtermann
- Paul McMillan
- Carl Meyer

10.5.4 Releasers

Releasers take care of building Django releases.

The current releasers are:

- James Bennett
- Jacob Kaplan-Moss
- Tim Graham

10.5.5 Ops team

The ops team maintains Django's infrastructure like the Django Project server, Trac instance, and continuous integration infrastructure.

- Florian Apolloner
- Aymeric Augustin
- Tim Graham
- Markus Holtermann
- Jannis Leidel

10.6 Django's security policies

Django's development team is strongly committed to responsible reporting and disclosure of security-related issues. As such, we've adopted and follow a set of policies which conform to that ideal and are geared toward allowing us to deliver timely security updates to the official distribution of Django, as well as to third-party distributions.

10.6.1 Reporting security issues

Short version: please report security issues by emailing security@djangoproject.com.

Most normal bugs in Django are reported to our public Trac instance, but due to the sensitive nature of security issues, we ask that they **not** be publicly reported in this fashion.

Instead, if you believe you've found something in Django which has security implications, please send a description of the issue via email to `security@djangoproject.com`. Mail sent to that address reaches a *subset of the core team*, who can forward security issues into the private team's mailing list for broader discussion if needed.

Once you've submitted an issue via email, you should receive an acknowledgment from a member of the security team within 48 hours, and depending on the action to be taken, you may receive further followup emails.

Sending encrypted reports

If you want to send an encrypted email (*optional*), the public key ID for `security@djangoproject.com` is `0xfcb84b8d1d17f80b`, and this public key is available from most commonly-used keyservers.

10.6.2 Supported versions

At any given time, the Django team provides official security support for several versions of Django:

- The master development branch, hosted on GitHub, which will become the next major release of Django, receives security support. Security issues that only affect the master development branch and not any stable released versions are fixed in public without going through the *disclosure process*.
- The two most recent Django release series receive security support. For example, during the development cycle leading to the release of Django 1.5, support will be provided for Django 1.4 and Django 1.3. Upon the release of Django 1.5, Django 1.3's security support will end.
- *Long-term support release*s will receive security updates for a specified period.

When new releases are issued for security reasons, the accompanying notice will include a list of affected versions. This list is comprised solely of *supported* versions of Django: older versions may also be affected, but we do not investigate to determine that, and will not issue patches or new releases for those versions.

10.6.3 How Django discloses security issues

Our process for taking a security issue from private discussion to public disclosure involves multiple steps.

Approximately one week before public disclosure, we send two notifications:

First, we notify *django-announce* of the date and approximate time of the upcoming security release, as well as the severity of the issues. This is to aid organizations that need to ensure they have staff available to handle triaging our announcement and upgrade Django as needed. Severity levels are:

High:

- Remote code execution
- SQL injection

Moderate:

- Cross site scripting (XSS)
- Cross site request forgery (CSRF)

- Broken authentication

Low:

- Sensitive data exposure
- Broken session management
- Unvalidated redirects/forwards
- Issues requiring an uncommon configuration option

Second, we notify a list of *people and organizations*, primarily composed of operating-system vendors and other distributors of Django. This email is signed with the PGP key of someone from *Django's release team* and consists of:

- A full description of the issue and the affected versions of Django.
- The steps we will be taking to remedy the issue.
- The patch(es), if any, that will be applied to Django.
- The date on which the Django team will apply these patches, issue new releases and publicly disclose the issue.

On the day of disclosure, we will take the following steps:

1. Apply the relevant patch(es) to Django's codebase.
2. Issue the relevant release(s), by placing new packages on the Python Package Index and on the Django website, and tagging the new release(s) in Django's git repository.
3. Post a public entry on the official Django development blog, describing the issue and its resolution in detail, pointing to the relevant patches and new releases, and crediting the reporter of the issue (if the reporter wishes to be publicly identified).
4. Post a notice to the *django-announce* and oss-security@lists.openwall.com mailing lists that links to the blog post.

If a reported issue is believed to be particularly time-sensitive – due to a known exploit in the wild, for example – the time between advance notification and public disclosure may be shortened considerably.

Additionally, if we have reason to believe that an issue reported to us affects other frameworks or tools in the Python/web ecosystem, we may privately contact and discuss those issues with the appropriate maintainers, and coordinate our own disclosure and resolution with theirs.

The Django team also maintains an archive of security issues disclosed in Django.

10.6.4 Who receives advance notification

The full list of people and organizations who receive advance notification of security issues is not and will not be made public.

We also aim to keep this list as small as effectively possible, in order to better manage the flow of confidential information prior to disclosure. As such, our notification list is *not* simply a list of users of Django, and merely being a user of Django is not sufficient reason to be placed on the notification list.

In broad terms, recipients of security notifications fall into three groups:

1. Operating-system vendors and other distributors of Django who provide a suitably-generic (i.e., *not* an individual's personal email address) contact address for reporting issues with their Django package, or for general security reporting. In either case, such addresses **must not** forward to public mailing lists or bug trackers. Addresses which forward to the private email of an individual maintainer or security-response contact are acceptable, although private security trackers or security-response groups are strongly preferred.

2. On a case-by-case basis, individual package maintainers who have demonstrated a commitment to responding to and responsibly acting on these notifications.
3. On a case-by-case basis, other entities who, in the judgment of the Django development team, need to be made aware of a pending security issue. Typically, membership in this group will consist of some of the largest and/or most likely to be severely impacted known users or distributors of Django, and will require a demonstrated ability to responsibly receive, keep confidential and act on these notifications.

10.6.5 Requesting notifications

If you believe that you, or an organization you are authorized to represent, fall into one of the groups listed above, you can ask to be added to Django's notification list by emailing `security@djangoproject.com`. Please use the subject line "Security notification request".

Your request **must** include the following information:

- Your full, real name and the name of the organization you represent, if applicable, as well as your role within that organization.
- A detailed explanation of how you or your organization fit at least one set of criteria listed above.
- A detailed explanation of why you are requesting security notifications. Again, please keep in mind that this is *not* simply a list for users of Django, and the overwhelming majority of users should subscribe to *django-announce* to receive advanced notice of when a security release will happen, without the details of the issues, rather than request detailed notifications.
- The email address you would like to have added to our notification list.
- An explanation of who will be receiving/reviewing mail sent to that address, as well as information regarding any automated actions that will be taken (i.e., filing of a confidential issue in a bug tracker).
- For individuals, the ID of a public key associated with your address which can be used to verify email received from you and encrypt email sent to you, as needed.

Once submitted, your request will be considered by the Django development team; you will receive a reply notifying you of the result of your request within 30 days.

Please also bear in mind that for any individual or organization, receiving security notifications is a privilege granted at the sole discretion of the Django development team, and that this privilege can be revoked at any time, with or without explanation.

10.7 Django's release process

10.7.1 Official releases

Since version 1.0, Django's release numbering works as follows:

- Versions are numbered in the form `A.B` or `A.B.C`.
- `A.B` is the *feature release* version number. Each version will be mostly backwards compatible with the previous release. Exceptions to this rule will be listed in the release notes.
- `C` is the *patch release* version number, which is incremented for bugfix and security releases. These releases will be 100% backwards-compatible with the previous patch release. The only exception is when a security or data loss issue can't be fixed without breaking backwards-compatibility. If this happens, the release notes will provide detailed upgrade instructions.

- Before a new feature release, we'll make alpha, beta, and release candidate releases. These are of the form `A.B alpha/beta/rc N`, which means the `Nth` alpha/beta/release candidate of version `A.B`.

In git, each Django release will have a tag indicating its version number, signed with the Django release key. Additionally, each release series has its own branch, called `stable/A.B.x`, and bugfix/security releases will be issued from those branches.

For more information about how the Django project issues new releases for security purposes, please see our security policies.

Feature release Feature releases (A.B, A.B+1, etc.) will happen roughly every eight months – see *release process* for details. These releases will contain new features, improvements to existing features, and such.

Patch release Patch releases (A.B.C, A.B.C+1, etc.) will be issued as needed, to fix bugs and/or security issues.

These releases will be 100% compatible with the associated feature release, unless this is impossible for security reasons or to prevent data loss. So the answer to "should I upgrade to the latest patch release?" will always be "yes."

Long-term support release Certain feature releases will be designated as long-term support (LTS) releases. These releases will get security and data loss fixes applied for a guaranteed period of time, typically three years.

See the download page for the releases that have been designated for long-term support.

10.7.2 Release cadence

Starting with Django 2.0, version numbers will use a loose form of semantic versioning such that each version following an LTS will bump to the next "dot zero" version. For example: 2.0, 2.1, 2.2 (LTS), 3.0, 3.1, 3.2 (LTS), etc.

SemVer makes it easier to see at a glance how compatible releases are with each other. It also helps to anticipate when compatibility shims will be removed. It's not a pure form of SemVer as each feature release will continue to have a few documented backwards incompatibilities where a deprecation path isn't possible or not worth the cost. Also, deprecations started in an LTS release (X.2) will be dropped in a non-dot-zero release (Y.1) to accommodate our policy of keeping deprecation shims for at least two feature releases. Read on to the next section for an example.

10.7.3 Deprecation policy

A feature release may deprecate certain features from previous releases. If a feature is deprecated in feature release A.x, it will continue to work in all A.x versions (for all versions of x) but raise warnings. Deprecated features will be removed in the B.0 release, or B.1 for features deprecated in the last A.x feature release to ensure deprecations are done over at least 2 feature releases.

So, for example, if we decided to start the deprecation of a function in Django 4.2:

- Django 4.2 will contain a backwards-compatible replica of the function which will raise a `RemovedInDjango51Warning`.
- Django 5.0 (the version that follows 4.2) will still contain the backwards-compatible replica.
- Django 5.1 will remove the feature outright.

The warnings are silent by default. You can turn on display of these warnings with the `python -Wd` option.

A more generic example:

- X.0
- X.1
- X.2 LTS

- Y.0: Drop deprecation shims added in X.0 and X.1.
- Y.1: Drop deprecation shims added in X.2.
- Y.2 LTS: No deprecation shims dropped (while Y.0 is no longer supported, third-party apps need to maintain compatibility back to X.2 LTS to ease LTS to LTS upgrades).
- Z.0: Drop deprecation shims added in Y.0 and Y.1.

10.7.4 Supported versions

At any moment in time, Django's developer team will support a set of releases to varying levels. See the supported versions section of the download page for the current state of support for each version.

- The current development master will get new features and bug fixes requiring non-trivial refactoring.
- Patches applied to the master branch must also be applied to the last feature release branch, to be released in the next patch release of that feature series, when they fix critical problems:
  - Security issues.
  - Data loss bugs.
  - Crashing bugs.
  - Major functionality bugs in newly-introduced features.
  - Regressions from older versions of Django.

  The rule of thumb is that fixes will be backported to the last feature release for bugs that would have prevented a release in the first place (release blockers).
- Security fixes and data loss bugs will be applied to the current master, the last two feature release branches, and any other supported long-term support release branches.
- Documentation fixes generally will be more freely backported to the last release branch. That's because it's highly advantageous to have the docs for the last release be up-to-date and correct, and the risk of introducing regressions is much less of a concern.

As a concrete example, consider a moment in time halfway between the release of Django 5.1 and 5.2. At this point in time:

- Features will be added to development master, to be released as Django 5.2.
- Critical bug fixes will be applied to the `stable/5.1.x` branch, and released as 5.1.1, 5.1.2, etc.
- Security fixes and bug fixes for data loss issues will be applied to `master` and to the `stable/5.1.x`, `stable/5.0.x`, and `stable/4.2.x` (LTS) branches. They will trigger the release of `5.1.1`, `5.0.5`, `4.2.8`, etc.
- Documentation fixes will be applied to master, and, if easily backported, to the latest stable branch, `5.1.x`.

10.7.5 Release process

Django uses a time-based release schedule, with feature releases every eight months or so.

After each feature release, the release manager will announce a timeline for the next feature release.

Release cycle

Each release cycle consists of three parts:

Phase one: feature proposal

The first phase of the release process will include figuring out what major features to include in the next version. This should include a good deal of preliminary work on those features – working code trumps grand design.

Major features for an upcoming release will be added to the wiki roadmap page, e.g. https://code.djangoproject.com/wiki/Version1.9Roadmap.

Phase two: development

The second part of the release schedule is the "heads-down" working period. Using the roadmap produced at the end of phase one, we'll all work very hard to get everything on it done.

At the end of phase two, any unfinished features will be postponed until the next release.

Phase two will culminate with an alpha release. At this point, the `stable/A.B.x` branch will be forked from `master`.

Phase three: bugfixes

The last part of a release cycle is spent fixing bugs – no new features will be accepted during this time. We'll try to release a beta release one month after the alpha and a release candidate one month after the beta.

The release candidate marks the string freeze, and it happens at least two weeks before the final release. After this point, new translatable strings must not be added.

During this phase, committers will be more and more conservative with backports, to avoid introducing regressions. After the release candidate, only release blockers and documentation fixes should be backported.

In parallel to this phase, `master` can receive new features, to be released in the `A.B+1` cycle.

Bug-fix releases

After a feature release (e.g. A.B), the previous release will go into bugfix mode.

The branch for the previous feature release (e.g. `stable/A.B-1.x`) will include bugfixes. Critical bugs fixed on master must *also* be fixed on the bugfix branch; this means that commits need to cleanly separate bug fixes from feature additions. The developer who commits a fix to master will be responsible for also applying the fix to the current bugfix branch.

10.8 Django Deprecation Timeline

This document outlines when various pieces of Django will be removed or altered in a backward incompatible way, following their deprecation, as per the *deprecation policy*. More details about each item can often be found in the release notes of two versions prior.

10.8.1 2.0

See the *Django 1.9 release notes* for more details on these changes.

- The `weak` argument to `django.dispatch.signals.Signal.disconnect()` will be removed.
- The `django.forms.extras` package will be removed.

- The `assignment_tag` helper will be removed.
- The `host` argument to `assertsRedirects` will be removed. The compatibility layer which allows absolute URLs to be considered equal to relative ones when the path is identical will also be removed.
- `Field.rel` will be removed.
- `Field.remote_field.to` attribute will be removed.
- The `on_delete` argument for `ForeignKey` and `OneToOneField` will be required.
- `django.db.models.fields.add_lazy_relation()` will be removed.
- When time zone support is enabled, database backends that don't support time zones won't convert aware datetimes to naive values in UTC anymore when such values are passed as parameters to SQL queries executed outside of the ORM, e.g. with `cursor.execute()`.
- The `django.contrib.auth.tests.utils.skipIfCustomUser()` decorator will be removed.
- The `GeoManager` and `GeoQuerySet` classes will be removed.
- The `django.contrib.gis.geoip` module will be removed.
- The `supports_recursion` check for template loaders will be removed from:
  - `django.template.engine.Engine.find_template()`
  - `django.template.loader_tags.ExtendsNode.find_template()`
  - `django.template.loaders.base.Loader.supports_recursion()`
  - `django.template.loaders.cached.Loader.supports_recursion()`
- The `load_template()` and `load_template_sources()` template loader methods will be removed.
- The `template_dirs` argument for template loaders will be removed:
  - `django.template.loaders.base.Loader.get_template()`
  - `django.template.loaders.cached.Loader.cache_key()`
  - `django.template.loaders.cached.Loader.get_template()`
  - `django.template.loaders.cached.Loader.get_template_sources()`
  - `django.template.loaders.filesystem.Loader.get_template_sources()`
- The `django.template.loaders.base.Loader.__call__()` method will be removed.
- Support for custom error views with a single positional parameter will be dropped.
- The `mime_type` attribute of `django.utils.feedgenerator.Atom1Feed` and `django.utils.feedgenerator.RssFeed` will be removed in favor of `content_type`.
- The `app_name` argument to *`include()`* will be removed.
- Support for passing a 3-tuple as the first argument to `include()` will be removed.
- Support for setting a URL instance namespace without an application namespace will be removed.
- `Field._get_val_from_obj()` will be removed in favor of `Field.value_from_object()`.
- `django.template.loaders.eggs.Loader` will be removed.
- The `current_app` parameter to the `contrib.auth` views will be removed.
- The `callable_obj` keyword argument to `SimpleTestCase.assertRaisesMessage()` will be removed.
- Support for the `allow_tags` attribute on `ModelAdmin` methods will be removed.

- The `enclosure` keyword argument to `SyndicationFeed.add_item()` will be removed.
- The `django.template.loader.LoaderOrigin` and `django.template.base.StringOrigin` aliases for `django.template.base.Origin` will be removed.

See the *Django 1.10 release notes* for more details on these changes.

- The `makemigrations --exit` option will be removed.
- Support for direct assignment to a reverse foreign key or many-to-many relation will be removed.
- The `get_srid()` and `set_srid()` methods of `django.contrib.gis.geos.GEOSGeometry` will be removed.
- The `get_x()`, `set_x()`, `get_y()`, `set_y()`, `get_z()`, and `set_z()` methods of `django.contrib.gis.geos.Point` will be removed.
- The `get_coords()` and `set_coords()` methods of `django.contrib.gis.geos.Point` will be removed.
- The `cascaded_union` property of `django.contrib.gis.geos.MultiPolygon` will be removed.
- `django.utils.functional.allow_lazy()` will be removed.
- The `shell --plain` option will be removed.
- The `django.core.urlresolvers` module will be removed.
- The model `CommaSeparatedIntegerField` will be removed. A stub field will remain for compatibility with historical migrations.
- Support for the template `Context.has_key()` method will be removed.
- Support for the `django.core.files.storage.Storage.accessed_time()`, `created_time()`, and `modified_time()` methods will be removed.
- Support for query lookups using the model name when `Meta.default_related_name` is set will be removed.
- The `__search` query lookup and the `DatabaseOperations.fulltext_search_sql()` method will be removed.
- The shim for supporting custom related manager classes without a `_apply_rel_filters()` method will be removed.
- Using `User.is_authenticated()` and `User.is_anonymous()` as methods will no longer be supported.
- The private attribute `virtual_fields` of `Model._meta` will be removed.
- The private keyword arguments `virtual_only` in `Field.contribute_to_class()` and `virtual` in `Model._meta.add_field()` will be removed.
- The `javascript_catalog()` and `json_catalog()` views will be removed.
- The `django.contrib.gis.utils.precision_wkt()` function will be removed.
- In multi-table inheritance, implicit promotion of a `OneToOneField` to a `parent_link` will be removed.
- Support for `Widget._format_value()` will be removed.
- `FileField` methods `get_directory_name()` and `get_filename()` will be removed.
- The `mark_for_escaping()` function and the classes it uses: `EscapeData`, `EscapeBytes`, `EscapeText`, `EscapeString`, and `EscapeUnicode` will be removed.
- The `escape` filter will change to use `django.utils.html.conditional_escape()`.

- `Manager.use_for_related_fields` will be removed.
- Model `Manager` inheritance will follow MRO inheritance rules and the `Meta.manager_inheritance_from_future` to opt-in to this behavior will be removed.
- Support for old-style middleware using `settings.MIDDLEWARE_CLASSES` will be removed.

10.8.2 1.10

See the *Django 1.8 release notes* for more details on these changes.

- Support for calling a `SQLCompiler` directly as an alias for calling its `quote_name_unless_alias` method will be removed.
- `cycle` and `firstof` template tags will be removed from the `future` template tag library (used during the 1.6/1.7 deprecation period).
- `django.conf.urls.patterns()` will be removed.
- Support for the `prefix` argument to `django.conf.urls.i18n.i18n_patterns()` will be removed.
- `SimpleTestCase.urls` will be removed.
- Using an incorrect count of unpacked values in the `for` template tag will raise an exception rather than fail silently.
- The ability to reverse URLs using a dotted Python path will be removed.
- The ability to use a dotted Python path for the `LOGIN_URL` and `LOGIN_REDIRECT_URL` settings will be removed.
- Support for `optparse` will be dropped for custom management commands (replaced by `argparse`).
- The class `django.core.management.NoArgsCommand` will be removed. Use *`BaseCommand`* instead, which takes no arguments by default.
- `django.core.context_processors` module will be removed.
- `django.db.models.sql.aggregates` module will be removed.
- `django.contrib.gis.db.models.sql.aggregates` module will be removed.
- The following methods and properties of `django.db.sql.query.Query` will be removed:
  - Properties: `aggregates` and `aggregate_select`
  - Methods: `add_aggregate`, `set_aggregate_mask`, and `append_aggregate_mask`.
- `django.template.resolve_variable` will be removed.
- The following private APIs will be removed from *`django.db.models.options.Options`* (`Model._meta`):
  - `get_field_by_name()`
  - `get_all_field_names()`
  - `get_fields_with_model()`
  - `get_concrete_fields_with_model()`
  - `get_m2m_with_model()`
  - `get_all_related_objects()`
  - `get_all_related_objects_with_model()`

  - `get_all_related_many_to_many_objects()`
  - `get_all_related_m2m_objects_with_model()`
- The `error_message` argument of `django.forms.RegexField` will be removed.
- The `unordered_list` filter will no longer support old style lists.
- Support for string `view` arguments to `url()` will be removed.
- The backward compatible shim to rename `django.forms.Form._has_changed()` to `has_changed()` will be removed.
- The `removetags` template filter will be removed.
- The `remove_tags()` and `strip_entities()` functions in `django.utils.html` will be removed.
- The `is_admin_site` argument to `django.contrib.auth.views.password_reset()` will be removed.
- `django.db.models.field.subclassing.SubfieldBase` will be removed.
- `django.utils.checksums` will be removed; its functionality is included in django-localflavor 1.1+.
- The `original_content_type_id` attribute on `django.contrib.admin.helpers.InlineAdminForm` will be removed.
- The backwards compatibility shim to allow `FormMixin.get_form()` to be defined with no default value for its `form_class` argument will be removed.
- The following settings will be removed:
  - `ALLOWED_INCLUDE_ROOTS`
  - `TEMPLATE_CONTEXT_PROCESSORS`
  - `TEMPLATE_DEBUG`
  - `TEMPLATE_DIRS`
  - `TEMPLATE_LOADERS`
  - `TEMPLATE_STRING_IF_INVALID`
- The backwards compatibility alias `django.template.loader.BaseLoader` will be removed.
- Django template objects returned by *`get_template()`* and *`select_template()`* won't accept a *`Context`* in their *`render()`* method anymore.
- Template response APIs will enforce the use of `dict` and backend-dependent template objects instead of *`Context`* and *`Template`* respectively.
- The `current_app` parameter for the following function and classes will be removed:
  - `django.shortcuts.render()`
  - `django.template.Context()`
  - `django.template.RequestContext()`
  - `django.template.response.TemplateResponse()`
- The `dictionary` and `context_instance` parameters for the following functions will be removed:
  - `django.shortcuts.render()`
  - `django.shortcuts.render_to_response()`
  - `django.template.loader.render_to_string()`

- The `dirs` parameter for the following functions will be removed:
  - `django.template.loader.get_template()`
  - `django.template.loader.select_template()`
  - `django.shortcuts.render()`
  - `django.shortcuts.render_to_response()`
- Session verification will be enabled regardless of whether or not `'django.contrib.auth.middleware.SessionAuthenticationMiddleware'` is in `MIDDLEWARE_CLASSES`.
- Private attribute `django.db.models.Field.related` will be removed.
- The `--list` option of the `migrate` management command will be removed.
- The `ssi` template tag will be removed.
- Support for the = comparison operator in the `if` template tag will be removed.
- The backwards compatibility shims to allow `Storage.get_available_name()` and `Storage.save()` to be defined without a `max_length` argument will be removed.
- Support for the legacy `%(<foo>)s` syntax in `ModelFormMixin.success_url` will be removed.
- `GeoQuerySet` aggregate methods `collect()`, `extent()`, `extent3d()`, `make_line()`, and `unionagg()` will be removed.
- Ability to specify `ContentType.name` when creating a content type instance will be removed.
- Support for the old signature of `allow_migrate` will be removed. It changed from `allow_migrate(self, db, model)` to `allow_migrate(self, db, app_label, model_name=None, **hints)`.
- Support for the syntax of `{% cycle %}` that uses comma-separated arguments will be removed.
- The warning that *`Signer`* issues when given an invalid separator will become an exception.

10.8.3 1.9

See the *Django 1.7 release notes* for more details on these changes.

- `django.utils.dictconfig` will be removed.
- `django.utils.importlib` will be removed.
- `django.utils.tzinfo` will be removed.
- `django.utils.unittest` will be removed.
- The `syncdb` command will be removed.
- `django.db.models.signals.pre_syncdb` and `django.db.models.signals.post_syncdb` will be removed.
- `allow_syncdb` on database routers will no longer automatically become `allow_migrate`.
- Automatic syncing of apps without migrations will be removed. Migrations will become compulsory for all apps unless you pass the `--run-syncdb` option to `migrate`.
- The SQL management commands for apps without migrations, `sql`, `sqlall`, `sqlclear`, `sqldropindexes`, and `sqlindexes`, will be removed.
- Support for automatic loading of `initial_data` fixtures and initial SQL data will be removed.

- All models will need to be defined inside an installed application or declare an explicit *app_label*. Furthermore, it won't be possible to import them before their application is loaded. In particular, it won't be possible to import models inside the root package of their application.
- The model and form `IPAddressField` will be removed. A stub field will remain for compatibility with historical migrations.
- `AppCommand.handle_app()` will no longer be supported.
- `RequestSite` and `get_current_site()` will no longer be importable from `django.contrib.sites.models`.
- FastCGI support via the `runfcgi` management command will be removed. Please deploy your project using WSGI.
- `django.utils.datastructures.SortedDict` will be removed. Use `collections.OrderedDict` from the Python standard library instead.
- `ModelAdmin.declared_fieldsets` will be removed.
- Instances of `util.py` in the Django codebase have been renamed to `utils.py` in an effort to unify all util and utils references. The modules that provided backwards compatibility will be removed:
  - `django.contrib.admin.util`
  - `django.contrib.gis.db.backends.util`
  - `django.db.backends.util`
  - `django.forms.util`
- `ModelAdmin.get_formsets` will be removed.
- The backward compatibility shim introduced to rename the `BaseMemcachedCache._get_memcache_timeout()` method to `get_backend_timeout()` will be removed.
- The `--natural` and `-n` options for *dumpdata* will be removed.
- The `use_natural_keys` argument for `serializers.serialize()` will be removed.
- Private API `django.forms.forms.get_declared_fields()` will be removed.
- The ability to use a `SplitDateTimeWidget` with `DateTimeField` will be removed.
- The `WSGIRequest.REQUEST` property will be removed.
- The class `django.utils.datastructures.MergeDict` will be removed.
- The `zh-cn` and `zh-tw` language codes will be removed and have been replaced by the `zh-hans` and `zh-hant` language code respectively.
- The internal `django.utils.functional.memoize` will be removed.
- `django.core.cache.get_cache` will be removed. Add suitable entries to *CACHES* and use *django.core.cache.caches* instead.
- `django.db.models.loading` will be removed.
- Passing callable arguments to querysets will no longer be possible.
- `BaseCommand.requires_model_validation` will be removed in favor of `requires_system_checks`. Admin validators will be replaced by admin checks.
- The `ModelAdmin.validator_class` and `default_validator_class` attributes will be removed.
- `ModelAdmin.validate()` will be removed.

- `django.db.backends.DatabaseValidation.validate_field` will be removed in favor of the `check_field` method.
- The `validate` management command will be removed.
- `django.utils.module_loading.import_by_path` will be removed in favor of `django.utils.module_loading.import_string`.
- `ssi` and `url` template tags will be removed from the `future` template tag library (used during the 1.3/1.4 deprecation period).
- `django.utils.text.javascript_quote` will be removed.
- Database test settings as independent entries in the database settings, prefixed by `TEST_`, will no longer be supported.
- The *cache_choices* option to *ModelChoiceField* and *ModelMultipleChoiceField* will be removed.
- The default value of the *RedirectView.permanent* attribute will change from `True` to `False`.
- `django.contrib.sitemaps.FlatPageSitemap` will be removed in favor of `django.contrib.flatpages.sitemaps.FlatPageSitemap`.
- Private API `django.test.utils.TestTemplateLoader` will be removed.
- The `django.contrib.contenttypes.generic` module will be removed.
- Private APIs `django.db.models.sql.where.WhereNode.make_atom()` and `django.db.models.sql.where.Constraint` will be removed.

10.8.4 1.8

See the *Django 1.6 release notes* for more details on these changes.

- `django.contrib.comments` will be removed.
- The following transaction management APIs will be removed:
  - `TransactionMiddleware`,
  - the decorators and context managers `autocommit`, `commit_on_success`, and `commit_manually`, defined in `django.db.transaction`,
  - the functions `commit_unless_managed` and `rollback_unless_managed`, also defined in `django.db.transaction`,
  - the `TRANSACTIONS_MANAGED` setting.
- The *cycle* and *firstof* template tags will auto-escape their arguments. In 1.6 and 1.7, this behavior is provided by the version of these tags in the `future` template tag library.
- The `SEND_BROKEN_LINK_EMAILS` setting will be removed. Add the *django.middleware.common.BrokenLinkEmailsMiddleware* middleware to your *MIDDLEWARE_CLASSES* setting instead.
- `django.middleware.doc.XViewMiddleware` will be removed. Use `django.contrib.admindocs.middleware.XViewMiddleware` instead.
- `Model._meta.module_name` was renamed to `model_name`.
- Remove the backward compatible shims introduced to rename `get_query_set` and similar queryset methods. This affects the following classes: `BaseModelAdmin`, `ChangeList`, `BaseCommentNode`, `GenericForeignKey`, `Manager`, `SingleRelatedObjectDescriptor` and `ReverseSingleRelatedObjectDescriptor`.

- Remove the backward compatible shims introduced to rename the attributes `ChangeList.root_query_set` and `ChangeList.query_set`.
- `django.views.defaults.shortcut` will be removed, as part of the goal of removing all `django.contrib` references from the core Django codebase. Instead use `django.contrib.contenttypes.views.shortcut`. `django.conf.urls.shortcut` will also be removed.
- Support for the Python Imaging Library (PIL) module will be removed, as it no longer appears to be actively maintained & does not work on Python 3. You are advised to install Pillow, which should be used instead.
- The following private APIs will be removed:
  - `django.db.backend`
  - `django.db.close_connection()`
  - `django.db.backends.creation.BaseDatabaseCreation.set_autocommit()`
  - `django.db.transaction.is_managed()`
  - `django.db.transaction.managed()`
- `django.forms.widgets.RadioInput` will be removed in favor of `django.forms.widgets.RadioChoiceInput`.
- The module `django.test.simple` and the class `django.test.simple.DjangoTestSuiteRunner` will be removed. Instead use `django.test.runner.DiscoverRunner`.
- The module `django.test._doctest` will be removed. Instead use the doctest module from the Python standard library.
- The `CACHE_MIDDLEWARE_ANONYMOUS_ONLY` setting will be removed.
- Usage of the hard-coded *Hold down "Control", or "Command" on a Mac, to select more than one.* string to override or append to user-provided `help_text` in forms for ManyToMany model fields will not be performed by Django anymore either at the model or forms layer.
- The `Model._meta.get_(add|change|delete)_permission` methods will be removed.
- The session key `django_language` will no longer be read for backwards compatibility.
- Geographic Sitemaps will be removed (`django.contrib.gis.sitemaps.views.index` and `django.contrib.gis.sitemaps.views.sitemap`).
- `django.utils.html.fix_ampersands`, the `fix_ampersands` template filter and `django.utils.html.clean_html` will be removed following an accelerated deprecation.

10.8.5 1.7

See the *Django 1.5 release notes* for more details on these changes.

- The module `django.utils.simplejson` will be removed. The standard library provides `json` which should be used instead.
- The function `django.utils.itercompat.product` will be removed. The Python builtin version should be used instead.
- Auto-correction of INSTALLED_APPS and TEMPLATE_DIRS settings when they are specified as a plain string instead of a tuple will be removed and raise an exception.
- The `mimetype` argument to the `__init__` methods of *`HttpResponse`*, *`SimpleTemplateResponse`*, and *`TemplateResponse`*, will be removed. `content_type` should be used instead. This also applies to the `render_to_response()` shortcut and the sitemap views, *`index()`* and *`sitemap()`*.

- When *HttpResponse* is instantiated with an iterator, or when *content* is set to an iterator, that iterator will be immediately consumed.
- The AUTH_PROFILE_MODULE setting, and the get_profile() method on the User model, will be removed.
- The cleanup management command will be removed. It's replaced by clearsessions.
- The daily_cleanup.py script will be removed.
- The depth keyword argument will be removed from *select_related()*.
- The undocumented get_warnings_state()/restore_warnings_state() functions from *django.test.utils* and the save_warnings_state()/ restore_warnings_state() *django.test.*TestCase* methods are deprecated. Use the warnings.catch_warnings context manager available starting with Python 2.6 instead.
- The undocumented check_for_test_cookie method in *AuthenticationForm* will be removed following an accelerated deprecation. Users subclassing this form should remove calls to this method, and instead ensure that their auth related views are CSRF protected, which ensures that cookies are enabled.
- The version of *django.contrib.auth.views.password_reset_confirm()* that supports base36 encoded user IDs (django.contrib.auth.views.password_reset_confirm_uidb36) will be removed. If your site has been running Django 1.6 for more than *PASSWORD_RESET_TIMEOUT_DAYS*, this change will have no effect. If not, then any password reset links generated before you upgrade to Django 1.7 won't work after the upgrade.
- The django.utils.encoding.StrAndUnicode mix-in will be removed. Define a __str__ method and apply the *python_2_unicode_compatible()* decorator instead.

10.8.6 1.6

See the *Django 1.4 release notes* for more details on these changes.

- django.contrib.databrowse will be removed.
- django.contrib.localflavor will be removed following an accelerated deprecation.
- django.contrib.markup will be removed following an accelerated deprecation.
- The compatibility modules django.utils.copycompat and django.utils.hashcompat as well as the functions django.utils.itercompat.all and django.utils.itercompat.any will be removed. The Python builtin versions should be used instead.
- The csrf_response_exempt and csrf_view_exempt decorators will be removed. Since 1.4 csrf_response_exempt has been a no-op (it returns the same function), and csrf_view_exempt has been a synonym for django.views.decorators.csrf.csrf_exempt, which should be used to replace it.
- The django.core.cache.backends.memcached.CacheClass backend was split into two in Django 1.3 in order to introduce support for PyLibMC. The historical CacheClass will be removed in favor of django.core.cache.backends.memcached.MemcachedCache.
- The UK-prefixed objects of django.contrib.localflavor.uk will only be accessible through their GB-prefixed names (GB is the correct ISO 3166 code for United Kingdom).
- The IGNORABLE_404_STARTS and IGNORABLE_404_ENDS settings have been superseded by *IGNORABLE_404_URLS* in the 1.4 release. They will be removed.
- The form wizard has been refactored to use class-based views with pluggable backends in 1.4. The previous implementation will be removed.

- Legacy ways of calling *cache_page()* will be removed.
- The backward-compatibility shim to automatically add a debug-false filter to the `'mail_admins'` logging handler will be removed. The *LOGGING* setting should include this filter explicitly if it is desired.
- The builtin truncation functions `django.utils.text.truncate_words()` and `django.utils.text.truncate_html_words()` will be removed in favor of the `django.utils.text.Truncator` class.
- The *GeoIP* class was moved to *django.contrib.gis.geoip* in 1.4 – the shortcut in *django.contrib.gis.utils* will be removed.
- `django.conf.urls.defaults` will be removed. The functions *include()*, `patterns()` and *url()* plus *handler404*, *handler500*, are now available through *django.conf.urls*.
- The functions `setup_environ()` and `execute_manager()` will be removed from *django.core.management*. This also means that the old (pre-1.4) style of `manage.py` file will no longer work.
- Setting the `is_safe` and `needs_autoescape` flags as attributes of template filter functions will no longer be supported.
- The attribute `HttpRequest.raw_post_data` was renamed to `HttpRequest.body` in 1.4. The backward compatibility will be removed – `HttpRequest.raw_post_data` will no longer work.
- The value for the `post_url_continue` parameter in `ModelAdmin.response_add()` will have to be either `None` (to redirect to the newly created object's edit page) or a pre-formatted url. String formats, such as the previous default `'../%s/'`, will not be accepted any more.

10.8.7 1.5

See the *Django 1.3 release notes* for more details on these changes.

- Starting Django without a *SECRET_KEY* will result in an exception rather than a `DeprecationWarning`. (This is accelerated from the usual deprecation path; see the Django 1.4 release notes.)
- The `mod_python` request handler will be removed. The `mod_wsgi` handler should be used instead.
- The `template` attribute on `django.test.client.Response` objects returned by the *test client* will be removed. The *templates* attribute should be used instead.
- The `django.test.simple.DjangoTestRunner` will be removed. Instead use a unittest-native class. The features of the `django.test.simple.DjangoTestRunner` (including fail-fast and Ctrl-C test termination) can currently be provided by the unittest-native `TextTestRunner`.
- The undocumented function `django.contrib.formtools.utils.security_hash` will be removed, instead use `django.contrib.formtools.utils.form_hmac`
- The function-based generic view modules will be removed in favor of their class-based equivalents, outlined here.
- The `django.core.servers.basehttp.AdminMediaHandler` will be removed. In its place use `django.contrib.staticfiles.handlers.StaticFilesHandler`.
- The template tags library `adminmedia` and the template tag `{% admin_media_prefix %}` will be removed in favor of the generic static files handling. (This is faster than the usual deprecation path; see the Django 1.4 release notes.)
- The `url` and `ssi` template tags will be modified so that the first argument to each tag is a template variable, not an implied string. In 1.4, this behavior is provided by a version of the tag in the `future` template tag library.
- The `reset` and `sqlreset` management commands will be removed.

- Authentication backends will need to support an inactive user being passed to all methods dealing with permissions. The `supports_inactive_user` attribute will no longer be checked and can be removed from custom backends.
- *`transform()`* will raise a *`GEOSException`* when called on a geometry with no SRID value.
- `django.http.CompatCookie` will be removed in favor of `django.http.SimpleCookie`.
- `django.core.context_processors.PermWrapper` and `django.core.context_processors.PermLookup` will be removed in favor of the corresponding `django.contrib.auth.context_processors.PermWrapper` and `django.contrib.auth.context_processors.PermLookupDict`, respectively.
- The *`MEDIA_URL`* or *`STATIC_URL`* settings will be required to end with a trailing slash to ensure there is a consistent way to combine paths in templates.
- `django.db.models.fields.URLField.verify_exists` will be removed. The feature was deprecated in 1.3.1 due to intractable security and performance issues and will follow a slightly accelerated deprecation timeframe.
- Translations located under the so-called *project path* will be ignored during the translation building process performed at runtime. The *`LOCALE_PATHS`* setting can be used for the same task by including the filesystem path to a `locale` directory containing non-app-specific translations in its value.
- The Markup contrib app will no longer support versions of Python-Markdown library earlier than 2.1. An accelerated timeline was used as this was a security related deprecation.
- The `CACHE_BACKEND` setting will be removed. The cache backend(s) should be specified in the *`CACHES`* setting.

10.8.8 1.4

See the *Django 1.2 release notes* for more details on these changes.

- `CsrfResponseMiddleware` and `CsrfMiddleware` will be removed. Use the `{% csrf_token %}` template tag inside forms to enable CSRF protection. `CsrfViewMiddleware` remains and is enabled by default.
- The old imports for CSRF functionality (`django.contrib.csrf.*`), which moved to core in 1.2, will be removed.
- The `django.contrib.gis.db.backend` module will be removed in favor of the specific backends.
- `SMTPConnection` will be removed in favor of a generic Email backend API.
- The many to many SQL generation functions on the database backends will be removed.
- The ability to use the `DATABASE_*` family of top-level settings to define database connections will be removed.
- The ability to use shorthand notation to specify a database backend (i.e., `sqlite3` instead of `django.db.backends.sqlite3`) will be removed.
- The `get_db_prep_save`, `get_db_prep_value` and `get_db_prep_lookup` methods will have to support multiple databases.
- The `Message` model (in `django.contrib.auth`), its related manager in the `User` model (`user.message_set`), and the associated methods (`user.message_set.create()` and `user.get_and_delete_messages()`), will be removed. The messages framework should be used instead. The related `messages` variable returned by the auth context processor will also be removed. Note that this means that the admin application will depend on the messages context processor.

- Authentication backends will need to support the `obj` parameter for permission checking. The `supports_object_permissions` attribute will no longer be checked and can be removed from custom backends.
- Authentication backends will need to support the `AnonymousUser` class being passed to all methods dealing with permissions. The `supports_anonymous_user` variable will no longer be checked and can be removed from custom backends.
- The ability to specify a callable template loader rather than a `Loader` class will be removed, as will the `load_template_source` functions that are included with the built in template loaders for backwards compatibility.
- `django.utils.translation.get_date_formats()` and `django.utils.translation.get_partial_dat` These functions will be removed; use the locale-aware `django.utils.formats.get_format()` to get the appropriate formats.
- In `django.forms.fields`, the constants: `DEFAULT_DATE_INPUT_FORMATS`, `DEFAULT_TIME_INPUT_FORMATS` and `DEFAULT_DATETIME_INPUT_FORMATS` will be removed. Use `django.utils.formats.get_format()` to get the appropriate formats.
- The ability to use a function-based test runner will be removed, along with the `django.test.simple.run_tests()` test runner.
- The `views.feed()` view and `feeds.Feed` class in `django.contrib.syndication` will be removed. The class-based view `views.Feed` should be used instead.
- `django.core.context_processors.auth`. This release will remove the old method in favor of the new method in `django.contrib.auth.context_processors.auth`.
- The `postgresql` database backend will be removed, use the `postgresql_psycopg2` backend instead.
- The `no` language code will be removed and has been replaced by the `nb` language code.
- Authentication backends will need to define the boolean attribute `supports_inactive_user` until version 1.5 when it will be assumed that all backends will handle inactive users.
- `django.db.models.fields.XMLField` will be removed. This was deprecated as part of the 1.3 release. An accelerated deprecation schedule has been used because the field hasn't performed any role beyond that of a simple `TextField` since the removal of `oldforms`. All uses of `XMLField` can be replaced with `TextField`.
- The undocumented `mixin` parameter to the `open()` method of `django.core.files.storage.Storage` (and subclasses) will be removed.

10.8.9 1.3

See the *Django 1.1 release notes* for more details on these changes.

- `AdminSite.root()`. This method of hooking up the admin URLs will be removed in favor of including `admin.site.urls`.
- Authentication backends need to define the boolean attributes `supports_object_permissions` and `supports_anonymous_user` until version 1.4, at which point it will be assumed that all backends will support these options.

10.9 The Django source code repository

When deploying a Django application into a real production environment, you will almost always want to use an official packaged release of Django.

However, if you'd like to try out in-development code from an upcoming release or contribute to the development of Django, you'll need to obtain a clone of Django's source code repository.

This document covers the way the code repository is laid out and how to work with and find things in it.

10.9.1 High-level overview

The Django source code repository uses Git to track changes to the code over time, so you'll need a copy of the Git client (a program called `git`) on your computer, and you'll want to familiarize yourself with the basics of how Git works.

Git's website offers downloads for various operating systems. The site also contains vast amounts of documentation.

The Django Git repository is located online at github.com/django/django. It contains the full source code for all Django releases, which you can browse online.

The Git repository includes several branches:

- `master` contains the main in-development code which will become the next packaged release of Django. This is where most development activity is focused.
- `stable/A.B.x` are the branches where release preparation work happens. They are also used for bugfix and security releases which occur as necessary after the initial release of a feature version.
- `soc20XX/<project>` branches were used by students who worked on Django during the 2009 and 2010 Google Summer of Code programs.
- `attic/<project>` branches were used to develop major or experimental new features without affecting the rest of Django's code.

The Git repository also contains tags. These are the exact revisions from which packaged Django releases were produced, since version 1.0.

The source code for the Djangoproject.com website can be found at github.com/django/djangoproject.com.

10.9.2 The master branch

If you'd like to try out the in-development code for the next release of Django, or if you'd like to contribute to Django by fixing bugs or developing new features, you'll want to get the code from the master branch.

Note that this will get *all* of Django: in addition to the top-level `django` module containing Python code, you'll also get a copy of Django's documentation, test suite, packaging scripts and other miscellaneous bits. Django's code will be present in your clone as a directory named `django`.

To try out the in-development code with your own applications, simply place the directory containing your clone on your Python import path. Then `import` statements which look for Django will find the `django` module within your clone.

If you're going to be working on Django's code (say, to fix a bug or develop a new feature), you can probably stop reading here and move over to the documentation for contributing to Django, which covers things like the preferred coding style and how to generate and submit a patch.

10.9.3 Other branches

Django uses branches to prepare for releases of Django.

In the past when Django was hosted on Subversion, branches were also used for feature development. Now Django is hosted on Git and feature development is done on contributor's forks, but the Subversion feature branches remain in Git for historical reference.

Stable branches

These branches can be found in the repository as `stable/A.B.x` branches and will be created right after the first alpha is tagged.

For example, immediately after *Django 1.5 alpha 1* was tagged, the branch `stable/1.5.x` was created and all further work on preparing the code for the final 1.5 release was done there.

These branches also provide limited bugfix support for the most recent released version of Django and security support for the two most recently-released versions of Django.

For example, after the release of Django 1.5, the branch `stable/1.5.x` receives only fixes for security and critical stability bugs, which are eventually released as Django 1.5.1 and so on, `stable/1.4.x` receives only security fixes, and `stable/1.3.x` no longer receives any updates.

Historical information

This policy for handling `stable/A.B.x` branches was adopted starting with the Django 1.5 release cycle.

Previously, these branches weren't created until right after the releases and the stabilization work occurred on the main repository branch. Thus, no new features development work for the next release of Django could be committed until the final release happened.

For example, shortly after the release of Django 1.3 the branch `stable/1.3.x` was created. Official support for that release has expired, and so it no longer receives direct maintenance from the Django project. However, that and all other similarly named branches continue to exist and interested community members have occasionally used them to provide unofficial support for old Django releases.

Feature-development branches

Historical information

Since Django moved to Git in 2012, anyone can clone the repository and create their own branches, alleviating the need for official branches in the source code repository.

The following section is mostly useful if you're exploring the repository's history, for example if you're trying to understand how some features were designed.

Feature-development branches tend by their nature to be temporary. Some produce successful features which are merged back into Django's master to become part of an official release, but others do not; in either case there comes a time when the branch is no longer being actively worked on by any developer. At this point the branch is considered closed.

Unfortunately, Django used to be maintained with the Subversion revision control system, that has no standard way of indicating this. As a workaround, branches of Django which are closed and no longer maintained were moved into `attic`.

For reference, the following are branches whose code eventually became part of Django itself, and so are no longer separately maintained:

- `boulder-oracle-sprint`: Added support for Oracle databases to Django's object-relational mapper. This has been part of Django since the 1.0 release.
- `gis`: Added support for geographic/spatial queries to Django's object-relational mapper. This has been part of Django since the 1.0 release, as the bundled application `django.contrib.gis`.
- `i18n`: Added internationalization support to Django. This has been part of Django since the 0.90 release.
- `magic-removal`: A major refactoring of both the internals and public APIs of Django's object-relational mapper. This has been part of Django since the 0.95 release.
- `multi-auth`: A refactoring of Django's bundled authentication framework which added support for *authentication backends*. This has been part of Django since the 0.95 release.
- `new-admin`: A refactoring of Django's bundled administrative application. This became part of Django as of the 0.91 release, but was superseded by another refactoring (see next listing) prior to the Django 1.0 release.
- `newforms-admin`: The second refactoring of Django's bundled administrative application. This became part of Django as of the 1.0 release, and is the basis of the current incarnation of `django.contrib.admin`.
- `queryset-refactor`: A refactoring of the internals of Django's object-relational mapper. This became part of Django as of the 1.0 release.
- `unicode`: A refactoring of Django's internals to consistently use Unicode-based strings in most places within Django and Django applications. This became part of Django as of the 1.0 release.

When Django moved from SVN to Git, the information about branch merges wasn't preserved in the source code repository. This means that the `master` branch of Django doesn't contain merge commits for the above branches.

However, this information is available as a grafts file. You can restore it by putting the following lines in `.git/info/grafts` in your local clone:

```
ac64e91a0cadc57f4bc5cd5d66955832320ca7a1 553a20075e6991e7a60baee51ea68c8adc520d9a 0cb8e31823b2e9f05c
79e68c225b926302ebb29c808dda8afa49856f5c d0f57e7c7385a112cb9e19d314352fc5ed5b0747 aa239e3e5405933af6
5cf8f684237ab5addaf3549b2347c3adf107c0a7 cb45fd0ae20597306cd1f877efc99d9bd7cbee98 e27211a0deae2f1d40
f69cf70ed813a8cd7e1f963a14ae39103e8d5265 d5dbeaa9be359a4c794885c2e9f1b5a7e5e51fb8 d2fcbcf9d76d5bb8a6
aab3a418ac9293bb4abd7670f65d930cb0426d58 4ea7a11659b8a0ab07b0d2e847975f7324664f10 adf4b9311d5d64a2bd
ff60c5f9de3e8690d1e86f3e9e3f7248a15397c8 7ef212af149540aa2da577a960d0d87029fd1514 45b4288bb66a3cda40
9dda4abee1225db7a7b195b84c915fdd141a7260 4fe5c9b7ee09dc25921918a6dbb7605edb374bc9 3a7c14b583621272d4
a19ed8aea395e8e07164ff7d85bd7dff2f24edca dc375fb0f3b7fbae740e8cfcd791b8bccb8a4e66 42ea7a5ce8aece67d1
9c52d56f6f8a9cdafb231adf9f4110473099c9b5 c91a30f00fd182faf8ca5c03cd7dbcf8b735b458 4a5c5c78f2ecd4ed88
953badbea5a04159adbfa970f5805c0232b6a401 4c958b15b250866b70ded7d82aa532f1e57f96ae 5664a678b29ab04cad
471596fc1afcb9c6258d317c619eaf5fd394e797 4e89105d64bb9e04c409139a41e9c7aac263df4c 3e9035a9625c8a8a5e
9233d0426537615e06b78d28010d17d5a66adf44 6632739e94c6c38b4c5a86cf5c80c48ae50ac49f 18e151bc3f8a85f276
```

Additionally, the following branches are closed, but their code was never merged into Django and the features they aimed to implement were never finished:

- `full-history`
- `generic-auth`
- `multiple-db-support`
- `per-object-permissions`
- `schema-evolution`
- `schema-evolution-ng`
- `search-api`

- `sqlalchemy`

All of the above-mentioned branches now reside in `attic`.

Finally, the repository contains `soc2009/xxx` and `soc2010/xxx` feature branches, used for Google Summer of Code projects.

10.9.4 Tags

Each Django release is tagged and signed by a *releaser*.

The tags can be found on GitHub's tags page.

10.10 How is Django Formed?

This document explains how to release Django.

Please, keep these instructions up-to-date if you make changes! The point here is to be descriptive, not prescriptive, so feel free to streamline or otherwise make changes, but **update this document accordingly!**

10.10.1 Overview

There are three types of releases that you might need to make:

- Security releases: disclosing and fixing a vulnerability. This'll generally involve two or three simultaneous releases – e.g. 1.5.x, 1.6.x, and, depending on timing, perhaps a 1.7 alpha/beta/rc.
- Regular version releases: either a final release (e.g. 1.5) or a bugfix update (e.g. 1.5.1).
- Pre-releases: e.g. 1.6 alpha, beta, or rc.

The short version of the steps involved is:

1. If this is a security release, pre-notify the security distribution list one week before the actual release.
2. Proofread the release notes, looking for organization and writing errors. Draft a blog post and email announcement.
3. Update version numbers and create the release package(s).
4. Upload the package(s) to the `djangoproject.com` server.
5. Upload the new version(s) to PyPI.
6. Declare the new version in the admin on `djangoproject.com`.
7. Post the blog entry and send out the email announcements.
8. Update version numbers post-release.

There are a lot of details, so please read on.

10.10.2 Prerequisites

You'll need a few things before getting started:

- A GPG key. If the key you want to use is not your default signing key, you'll need to add `-u you@example.com` to every GPG signing command below, where `you@example.com` is the email address associated with the key you want to use.

- An install of some required Python packages:

```
$ pip install wheel twine
```

- Access to Django's record on PyPI. Create a file with your credentials:

```
/.pypirc
```

```
[pypi]
username:YourUsername
password:YourPassword
```

- Access to the `djangoproject.com` server to upload files.
- Access to the admin on `djangoproject.com` as a "Site maintainer".
- Access to post to `django-announce`.
- If this is a security release, access to the pre-notification distribution list.

If this is your first release, you'll need to coordinate with James and/or Jacob to get all these things lined up.

10.10.3 Pre-release tasks

A few items need to be taken care of before even beginning the release process. This stuff starts about a week before the release; most of it can be done any time leading up to the actual release:

1. If this is a security release, send out pre-notification **one week** before the release. We maintain a list of who gets these pre-notification emails in the private `django-core` repository. Send the mail to `security@djangoproject.com` and BCC the pre-notification recipients. This email should be signed by the key you'll use for the release, and should include patches for each issue being fixed.
2. As the release approaches, watch Trac to make sure no release blockers are left for the upcoming release.
3. Check with the other committers to make sure they don't have any uncommitted changes for the release.
4. Proofread the release notes, including looking at the online version to catch any broken links or reST errors, and make sure the release notes contain the correct date.
5. Double-check that the release notes mention deprecation timelines for any APIs noted as deprecated, and that they mention any changes in Python version support.
6. Double-check that the release notes index has a link to the notes for the new release; this will be in `docs/releases/index.txt`.
7. If this is a feature release, ensure translations from Transifex have been integrated. This is typically done by a separate translation's manager rather than the releaser, but here are the steps. Provided you have an account on Transifex:

```
$ python scripts/manage_translations.py fetch
```

   and then commit the changed/added files (both .po and .mo). Sometimes there are validation errors which need to be debugged, so avoid doing this task immediately before a release is needed.

8. *Update the django-admin manual page*:

```
$ cd docs
$ make man
$ man _build/man/django-admin.1  # do a quick sanity check
$ cp _build/man/django-admin.1 man/django-admin.1
```

   and then commit the changed man page.

10.10.4 Preparing for release

Write the announcement blog post for the release. You can enter it into the admin at any time and mark it as inactive. Here are a few examples: example security release announcement, example regular release announcement, example pre-release announcement.

10.10.5 Actually rolling the release

OK, this is the fun part, where we actually push out a release!

1. Check Jenkins is green for the version(s) you're putting out. You probably shouldn't issue a release until it's green.
2. A release always begins from a release branch, so you should make sure you're on a stable branch and up-to-date. For example:

```
$ git checkout stable/1.5.x
$ git pull
```

3. If this is a security release, merge the appropriate patches from `django-private`. Rebase these patches as necessary to make each one a simple commit on the release branch rather than a merge commit. To ensure this, merge them with the `--ff-only` flag; for example:

```
$ git checkout stable/1.5.x
$ git merge --ff-only security/1.5.x
```

   (This assumes `security/1.5.x` is a branch in the `django-private` repo containing the necessary security patches for the next release in the 1.5 series.)

   If git refuses to merge with `--ff-only`, switch to the security-patch branch and rebase it on the branch you are about to merge it into (`git checkout security/1.5.x; git rebase stable/1.5.x`) and then switch back and do the merge. Make sure the commit message for each security fix explains that the commit is a security fix and that an announcement will follow (example security commit).

4. For a feature release, remove the `UNDER DEVELOPMENT` header at the top of the release notes and add the release date on the next line. For a patch release, replace `*Under Development*` with the release date. Make this change on all branches where the release notes for a particular version are located.
5. Update the version number in `django/__init__.py` for the release. Please see *notes on setting the VERSION tuple* below for details on `VERSION`.
6. If this is a pre-release package, update the "Development Status" trove classifier in `setup.py` to reflect this. Otherwise, make sure the classifier is set to `Development Status :: 5 - Production/Stable`.
7. Tag the release using `git tag`. For example:

```
$ git tag --sign --message="Tag 1.5.1" 1.5.1
```

   You can check your work by running `git tag --verify <tag>`.

8. Push your work, including the tag: `git push --tags`.
9. Make sure you have an absolutely clean tree by running `git clean -dfx`.
10. Run `make -f extras/Makefile` to generate the release packages. This will create the release packages in a `dist/` directory.
11. Generate the hashes of the release packages:

```
$ cd dist
$ md5sum *
$ sha1sum *
$ sha256sum *
```

12. Create a "checksums" file, `Django-<<VERSION>>.checksum.txt` containing the hashes and release information. Start with this template and insert the correct version, date, GPG key ID (from `gpg --list-keys --keyid-format LONG`), release URL, and checksums:

```
This file contains MD5, SHA1, and SHA256 checksums for the source-code
tarball and wheel files of Django <<VERSION>>, released <<DATE>>.

To use this file, you will need a working install of PGP or other
compatible public-key encryption software. You will also need to have
the Django release manager's public key in your keyring; this key has
the ID ``XXXXXXXXXXXXXXXX`` and can be imported from the MIT
keyserver. For example, if using the open-source GNU Privacy Guard
implementation of PGP:

    gpg --keyserver pgp.mit.edu --recv-key XXXXXXXXXXXXXXXX

Once the key is imported, verify this file::

    gpg --verify <<THIS FILENAME>>

Once you have verified this file, you can use normal MD5, SHA1, or SHA256
checksumming applications to generate the checksums of the Django
package and compare them to the checksums listed below.

Release packages:
=================

https://www.djangoproject.com/m/releases/<<RELEASE TAR.GZ FILENAME>>
https://www.djangoproject.com/m/releases/<<RELEASE WHL FILENAME>>

MD5 checksums:
==============

<<MD5SUM>>  <<RELEASE TAR.GZ FILENAME>>
<<MD5SUM>>  <<RELEASE WHL FILENAME>>

SHA1 checksums:
===============

<<SHA1SUM>>  <<RELEASE TAR.GZ FILENAME>>
<<SHA1SUM>>  <<RELEASE WHL FILENAME>>

SHA256 checksums:
=================

<<SHA256SUM>>  <<RELEASE TAR.GZ FILENAME>>
<<SHA256SUM>>  <<RELEASE WHL FILENAME>>
```

13. Sign the checksum file (`gpg --clearsign --digest-algo SHA256 Django-<version>.checksum.txt`). This generates a signed document, `Django-<version>.checksum.txt.asc` which you can then verify using `gpg --verify Django-<version>.checksum.txt.asc`.

If you're issuing multiple releases, repeat these steps for each release.

10.10.6 Making the release(s) available to the public

Now you're ready to actually put the release out there. To do this:

1. Upload the release package(s) to the djangoproject server, replacing A.B. with the appropriate version number, e.g. 1.5 for a 1.5.x release:

```
$ scp Django-* djangoproject.com:/home/www/www/media/releases/A.B
```

2. Upload the checksum file(s):

```
$ scp Django-A.B.C.checksum.txt.asc djangoproject.com:/home/www/www/media/pgp/Django-A.B.C.check
```

3. Test that the release packages install correctly using `easy_install` and `pip`. Here's one method (which requires virtualenvwrapper):

```
$ RELEASE_VERSION='1.7.2'
$ MAJOR_VERSION=`echo $RELEASE_VERSION| cut -c 1-3`

$ mktmpenv
$ easy_install https://www.djangoproject.com/m/releases/$MAJOR_VERSION/Django-$RELEASE_VERSION.t
$ deactivate
$ mktmpenv
$ pip install https://www.djangoproject.com/m/releases/$MAJOR_VERSION/Django-$RELEASE_VERSION.ta
$ deactivate
$ mktmpenv
$ pip install https://www.djangoproject.com/m/releases/$MAJOR_VERSION/Django-$RELEASE_VERSION-py
$ deactivate
```

   This just tests that the tarballs are available (i.e. redirects are up) and that they install correctly, but it'll catch silly mistakes.

4. Ask a few people on IRC to verify the checksums by visiting the checksums file (e.g. https://www.djangoproject.com/m/pgp/Django-1.5b1.checksum.txt) and following the instructions in it. For bonus points, they can also unpack the downloaded release tarball and verify that its contents appear to be correct (proper version numbers, no stray `.pyc` or other undesirable files).

5. Upload the release packages to PyPI (for pre-releases, only upload the wheel file):

```
$ twine upload -s dist/*
```

6. Go to the Add release page in the admin, enter the new release number exactly as it appears in the name of the tarball (Django-<version>.tar.gz). So for example enter "1.5.1" or "1.4c2", etc. If the release is part of an LTS branch, mark it so.

7. Make the blog post announcing the release live.

8. For a new version release (e.g. 1.5, 1.6), update the default stable version of the docs by flipping the `is_default` flag to `True` on the appropriate `DocumentRelease` object in the `docs.djangoproject.com` database (this will automatically flip it to `False` for all others); you can do this using the site's admin.

9. Post the release announcement to the *django-announce*, *django-developers*, and *django-users* mailing lists. This should include a link to the announcement blog post. If this is a security release, also include oss-security@lists.openwall.com.

10. Add a link to the blog post in the topic of the #django IRC channel: `/msg chanserv TOPIC #django new topic goes here`.

10.10.7 Post-release

You're almost done! All that's left to do now is:

1. Update the `VERSION` tuple in `django/__init__.py` again, incrementing to whatever the next expected release will be. For example, after releasing 1.5.1, update `VERSION` to `VERSION = (1, 5, 2, 'alpha', 0)`.
2. Add the release in Trac's versions list if necessary (and make it the default if it's a final release). Not all versions are declared; take example on previous releases.
3. If this was a security release, update Archive of security issues with details of the issues addressed.

10.10.8 New stable branch tasks

There are several items to do in the time following the creation of a new stable branch (often following an alpha release). Some of these tasks don't need to be done by the releaser.

1. Create a new `DocumentRelease` object in the `docs.djangoproject.com` database for the new version's docs, and update the `docs/fixtures/doc_releases.json` JSON fixture, so people without access to the production DB can still run an up-to-date copy of the docs site.
2. Create a stub release note for the new feature version. Use the stub from the previous feature release version or copy the contents from the previous feature version and delete most of the contents leaving only the headings.
3. Increase the default PBKDF2 iterations in `django.contrib.auth.hashers.PBKDF2PasswordHasher` by about 20% (pick a round number). Run the tests, and update the 3 failing hasher tests with the new values. Make sure this gets noted in the release notes (see the 1.8 release notes for an example).
4. Remove features that have reached the end of their deprecation cycle. Each removal should be done in a separate commit for clarity. In the commit message, add a "refs #XXXX" to the original ticket where the deprecation began if possible.
5. Remove `.. versionadded::`, `.. versionadded::`, and `.. deprecated::` annotations in the documentation from two releases ago. For example, in Django 1.9, notes for 1.7 will be removed.
6. Add the new branch to Read the Docs. Since the automatically generated version names ("stable-A.B.x") differ from the version numbers we've used historically in Read the Docs ("A.B.x"), we currently ask Eric Holscher to add the version for us. Someday the alias functionality may be built-in to the Read the Docs UI.

10.10.9 Notes on setting the VERSION tuple

Django's version reporting is controlled by the `VERSION` tuple in `django/__init__.py`. This is a five-element tuple, whose elements are:

1. Major version.
2. Minor version.
3. Micro version.
4. Status – can be one of "alpha", "beta", "rc" or "final".
5. Series number, for alpha/beta/RC packages which run in sequence (allowing, for example, "beta 1", "beta 2", etc.).

For a final release, the status is always "final" and the series number is always 0. A series number of 0 with an "alpha" status will be reported as "pre-alpha".

Some examples:

- `(1, 2, 1, 'final', 0)` → "1.2.1"
- `(1, 3, 0, 'alpha', 0)` → "1.3 pre-alpha"
- `(1, 3, 0, 'beta', 2)` → "1.3 beta 2"

CHAPTER 11

Indices, glossary and tables

- genindex
- modindex
- Glossary

a

django.apps, 611

c

django.conf.urls, 1312
django.conf.urls.i18n, 445
django.contrib.admin, 678
django.contrib.admindocs, 684
django.contrib.auth, 392
django.contrib.auth.backends, 734
django.contrib.auth.forms, 365
django.contrib.auth.hashers, 374
django.contrib.auth.middleware, 1041
django.contrib.auth.password_validation, 374
django.contrib.auth.signals, 733
django.contrib.auth.views, 357
django.contrib.contenttypes, 736
django.contrib.contenttypes.admin, 741
django.contrib.contenttypes.fields, 738
django.contrib.contenttypes.forms, 741
django.contrib.flatpages, 742
django.contrib.gis, 747
django.contrib.gis.admin, 850
django.contrib.gis.db.backends, 773
django.contrib.gis.db.models, 769
django.contrib.gis.db.models.functions, 796
django.contrib.gis.feeds, 851
django.contrib.gis.forms, 778
django.contrib.gis.gdal, 820
django.contrib.gis.geoip, 841
django.contrib.gis.geoip2, 844
django.contrib.gis.geos, 805
django.contrib.gis.measure, 803
django.contrib.gis.serializers.geojson, 848
django.contrib.gis.utils, 846
django.contrib.gis.utils.layermapping, 846
django.contrib.gis.utils.ogrinspect, 848
django.contrib.gis.widgets, 779
django.contrib.humanize, 855
django.contrib.messages, 857
django.contrib.messages.middleware, 1038
django.contrib.postgres, 863
django.contrib.postgres.aggregates, 864
django.contrib.postgres.validators, 885
django.contrib.redirects, 885
django.contrib.sessions, 205
django.contrib.sessions.middleware, 1041
django.contrib.sitemaps, 887
django.contrib.sites, 894
django.contrib.sites.middleware, 1041
django.contrib.staticfiles, 901
django.contrib.syndication, 907
django.core.checks, 515
django.core.exceptions, 970
django.core.files, 974
django.core.files.storage, 976
django.core.files.uploadedfile, 978
django.core.files.uploadhandler, 980
django.core.mail, 416
django.core.management, 523
django.core.paginator, 480
django.core.signals, 1235
django.core.signing, 414
django.core.validators, 1328

d

django.db, 84
django.db.backends, 1236
django.db.backends.base.schema, 1182
django.db.migrations, 294
django.db.migrations.operations, 1042
django.db.models, 84
django.db.models.fields, 1050
django.db.models.fields.related, 1065
django.db.models.functions, 1157
django.db.models.functions.datetime, 1162

django.db.models.lookups, 1142
django.db.models.options, 1075
django.db.models.signals, 1229
django.db.transaction, 141
django.dispatch, 511

f

django.forms, 981
django.forms.fields, 998
django.forms.formsets, 227
django.forms.models, 237
django.forms.widgets, 1019

h

django.http, 1168

m

django.middleware, 1036
django.middleware.cache, 1036
django.middleware.clickjacking, 1041
django.middleware.common, 1036
django.middleware.csrf, 1041
django.middleware.exception, 1037
django.middleware.gzip, 1037
django.middleware.http, 1038
django.middleware.locale, 1038
django.middleware.security, 1039

s

django.shortcuts, 196

t

django.template, 259
django.template.backends, 262
django.template.backends.django, 262
django.template.backends.jinja2, 263
django.template.loader, 260
django.template.response, 1299
django.test, 309
django.test.signals, 1235
django.test.utils, 345

u

django.urls, 1309
django.utils, 1314
django.utils.cache, 1314
django.utils.dateparse, 1315
django.utils.decorators, 1316
django.utils.encoding, 1316
django.utils.feedgenerator, 1318
django.utils.functional, 1319
django.utils.html, 1322
django.utils.http, 1323
django.utils.log, 470
django.utils.module_loading, 1324
django.utils.safestring, 1324
django.utils.six, 489
django.utils.text, 1325
django.utils.timezone, 1325
django.utils.translation, 426

v

django.views, 1332
django.views.decorators.cache, 192
django.views.decorators.csrf, 927
django.views.decorators.gzip, 192
django.views.decorators.http, 191
django.views.decorators.vary, 192
django.views.generic.dates, 638
django.views.i18n, 439

Symbols

–addrport ADDRPORT
    testserver , 963
–admins
    sendtestemail , 957
–all
    diffsettings , 948
–all, -a
    dumpdata , 949
    makemessages , 953
–app APP_LABEL
    loaddata , 951
–backwards
    sqlmigrate , 959
–blank BLANK
    ogrinspect , 850
–check
    makemigrations , 955
–clear, -c
    collectstatic , 902
–command COMMAND, -c COMMAND
    shell , 958
–database DATABASE
    changepassword , 964
    createcachetable , 948
    createsuperuser , 965
    dbshell , 948
    dumpdata , 949
    flush , 950
    inspectdb , 950
    loaddata , 950
    migrate , 955
    showmigrations , 958
    sqlflush , 959
    sqlmigrate , 959
    sqlsequencereset , 959
–debug-sql, -d
    test , 962
–decimal DECIMAL
    ogrinspect , 850
–deploy
    check , 947
–domain DOMAIN, -d DOMAIN
    makemessages , 953
–dry-run
    createcachetable , 948
    makemigrations , 954
–dry-run, -n
    collectstatic , 902
–email EMAIL
    createsuperuser , 964
–empty
    makemigrations , 954
–exclude EXCLUDE, -e EXCLUDE
    dumpdata , 949
–exclude EXCLUDE, -x EXCLUDE
    compilemessages , 947
    makemessages , 953
–exclude-tag EXCLUDE_TAGS
    test , 963
–exit, -e
    makemigrations , 954
–extension EXTENSIONS, -e EXTENSIONS
    makemessages , 953
    startapp , 960
    startproject , 961
–fail-level {CRITICAL,ERROR,WARNING,INFO,DEBUG}
    check , 947
–failfast
    test , 962
–fake
    migrate , 955
–fake-initial
    migrate , 955
–format FORMAT
    dumpdata , 949
–geom-name GEOM_NAME
    ogrinspect , 850
–ignore PATTERN, -i PATTERN
    collectstatic , 902
    makemessages , 953

–ignorenonexistent, -i
loaddata , 951
–indent INDENT
dumpdata , 949
–insecure
runserver , 903
–interface {ipython,bpython,python}, {ipython,bpython,python}
shell , 958
–ipv6, -6
runserver , 956
–keep-pot
makemessages , 954
–keepdb, -k
test , 962
–layer LAYER_KEY
ogrinspect , 850
–link, -l
collectstatic , 902
–list, -l
showmigrations , 958
–list-tags
check , 947
–liveserver LIVESERVER
test , 962
–locale LOCALE, -l LOCALE
compilemessages , 947
makemessages , 953
–managers
sendtestemail , 957
–mapping
ogrinspect , 850
–merge
makemigrations , 954
–multi-geom
ogrinspect , 850
–name FILES, -n FILES
startapp , 960
startproject , 961
–name NAME, -n NAME
makemigrations , 954
–name-field NAME_FIELD
ogrinspect , 850
–natural-foreign
dumpdata , 949
–natural-primary
dumpdata , 949
–no-color
, 966
–no-default-ignore
collectstatic , 902
makemessages , 954
–no-imports
ogrinspect , 850
–no-location
makemessages , 954
–no-optimize
squashmigrations , 959
–no-post-process
collectstatic , 902
-i –no-wrap
makemessages , 954
–noinput, –no-input
collectstatic , 902
flush , 949
makemigrations , 954
migrate , 955
squashmigrations , 960
test , 962
testserver , 964
–noreload
runserver , 956
–nostartup
shell , 958
–nostatic
runserver , 903
–nothreading
runserver , 956
–null NULL
ogrinspect , 850
–output OUTPUT, -o OUTPUT
dumpdata , 949
–parallel [N]
test , 962
–pks PRIMARY_KEYS
dumpdata , 949
–plan, -p
showmigrations , 958
–pythonpath PYTHONPATH
, 966
–reverse, -r
test , 962
–run-syncdb
migrate , 955
–settings SETTINGS
, 966
–srid SRID
ogrinspect , 850
–symlinks, -s
makemessages , 953
–tag TAGS
test , 963
–tag TAGS, -t TAGS
check , 946
–template TEMPLATE
startapp , 960
startproject , 961
–testrunner TESTRUNNER

test , 962
–traceback
, 966
–use-fuzzy, -f
compilemessages , 947
–username USERNAME
createsuperuser , 964
–verbosity {0,1,2,3}, –v {0,1,2,3}
, 966
–no-color, 966
–pythonpath PYTHONPATH, 966
–settings SETTINGS, 966
–traceback, 966
–verbosity {0,1,2,3}, –v {0,1,2,3}, 966
_open() (django.core.files.storage), 560
_save() (django.core.files.storage), 560

A

A (django.contrib.gis.measure), 805
ABSOLUTE_URL_OVERRIDES
setting, 1185
abstract (Options), 1082
AccessMixin (django.contrib.auth.mixins), 356
activate() (django.utils.timezone), 1325
activate() (django.utils.translation), 1327
add
template filter, 1262
add_arguments() (django.test.runner.DiscoverRunner class method), 344
add_error() (Form), 984
add_message() (django.contrib.messages), 859
add_never_cache_headers() (django.utils.cache), 1315
AddField (django.db.migrations.operations), 1044
addslashes
template filter, 1263
AdminEmailHandler (django.utils.log), 478
AdminPasswordChangeForm (django.contrib.auth.forms), 365
ADMINS
setting, 1185
AdminSite (django.contrib.admin), 722
Aggregate (django.db.models), 1149
aggregate() (django.db.models.query.QuerySet), 1128
all() (django.db.models.query.QuerySet), 1111
allow_lazy() (django.utils.functional), 1320
allow_migrate(), 152
allow_relation(), 152
AllowAllUsersModelBackend (django.contrib.auth.backends), 735
AllowAllUsersRemoteUserBackend (django.contrib.auth.backends), 735
ALLOWED_HOSTS
setting, 1186
AlterField (django.db.migrations.operations), 1045
AlterIndexTogether (django.db.migrations.operations), 1044
AlterModelManagers (django.db.migrations.operations), 1044
AlterModelOptions (django.db.migrations.operations), 1044
AlterModelTable (django.db.migrations.operations), 1044
AlterOrderWithRespectTo (django.db.migrations.operations), 1044
AlterUniqueTogether (django.db.migrations.operations), 1044
annotate() (django.db.models.query.QuerySet), 1103
apnumber
template filter, 855
app_directories.Loader (django.template.loaders), 1293
AppCommand (django.core.management), 527
AppConfig (django.apps), 613
APPEND_SLASH
setting, 1186
application namespace, **186**
AppRegistryNotReady, 970
apps (django.apps), 615
apps.AdminConfig (django.contrib.admin), 688
apps.SimpleAdminConfig (django.contrib.admin), 688
ArchiveIndexView (django.views.generic.dates), 638
Area (django.contrib.gis.db.models.functions), 797
Area (django.contrib.gis.measure), 805
area (GEOSGeometry), 812
area (OGRGeometry), 827
arg_joiner (Func), 1148
arity (Func), 1148
ArrayAgg (django.contrib.postgres.aggregates), 864
ArrayField (django.contrib.postgres.fields), 866
arrayfield.contained_by
field lookup type, 868
arrayfield.contains
field lookup type, 868
arrayfield.index
field lookup type, 868
arrayfield.len
field lookup type, 868
arrayfield.overlap
field lookup type, 868
arrayfield.slice
field lookup type, 869
as_int() (Field), 825
as_manager() (django.db.models.query.QuerySet), 1130
as_p() (Form), 988
as_sql() (django.db.models), 1143
as_sql() (Func), 1148
as_table() (Form), 989
as_ul() (Form), 988
as_vendorname() (django.db.models), 1143

asc() (Expression), 1152
AsGeoJSON (django.contrib.gis.db.models.functions), 797
AsGML (django.contrib.gis.db.models.functions), 797
AsKML (django.contrib.gis.db.models.functions), 798
AsSVG (django.contrib.gis.db.models.functions), 798
Atom1Feed (django.utils.feedgenerator), 1319
atomic() (django.db.transaction), 142
attrs (Widget), 1022
auth() (django.contrib.auth.context_processors), 1290
AUTH_PASSWORD_VALIDATORS
    setting, 1219
AUTH_USER_MODEL
    setting, 1218
authenticate() (django.contrib.auth), 348
AUTHENTICATION_BACKENDS
    setting, 1218
AuthenticationForm (django.contrib.auth.forms), 365
AuthenticationMiddleware (django.contrib.auth.middleware), 1041
auto_id (Form), 990
autodiscover() (django.contrib.admin), 688
autoescape
    template tag, 1246
AutoField (django.db.models), 1055
Avg (django.db.models), 1139

B

backends.base.SessionBase (django.contrib.sessions), 207
backends.cached_db.SessionStore (django.contrib.sessions), 215
backends.db.SessionStore (django.contrib.sessions), 215
backends.smtp.EmailBackend (django.core.mail), 422
bands (GDALRaster), 837
base36_to_int() (django.utils.http), 1323
base_session.AbstractBaseSession (django.contrib.sessions), 214
base_session.BaseSessionManager (django.contrib.sessions), 215
BaseArchiveIndexView (django.views.generic.dates), 647
BaseCommand (django.core.management), 525
BaseDatabaseSchemaEditor (django.db.backends.base.schema), 1182
BaseDateDetailView (django.views.generic.dates), 647
BaseDateListView (django.views.generic.dates), 660
BaseDayArchiveView (django.views.generic.dates), 647
BaseFormSet (django.forms.formsets), 227
BaseGenericInlineFormSet (django.contrib.contenttypes.forms), 741
BaseGeometryWidget (django.contrib.gis.widgets), 780
BaseMonthArchiveView (django.views.generic.dates), 647
BaseTodayArchiveView (django.views.generic.dates), 647
BaseWeekArchiveView (django.views.generic.dates), 647
BaseYearArchiveView (django.views.generic.dates), 647
bbcontains
    field lookup type, 781
bboverlaps
    field lookup type, 782
BigAutoField (django.db.models), 1055
BigIntegerField (django.db.models), 1055
BigIntegerRangeField (django.contrib.postgres.fields), 874
BinaryField (django.db.models), 1055
BitAnd (django.contrib.postgres.aggregates), 864
BitOr (django.contrib.postgres.aggregates), 864
blank (Field), 1050
block
    template tag, 1246
blocktrans
    template tag, 435
body (HttpRequest), 1169
BoolAnd (django.contrib.postgres.aggregates), 864
BooleanField (django.db.models), 1056
BooleanField (django.forms), 1004
BoolOr (django.contrib.postgres.aggregates), 864
BoundField (django.forms), 993
BoundingCircle (django.contrib.gis.db.models.functions), 798
BrokenLinkEmailsMiddleware (django.middleware.common), 1037
bulk_create() (django.db.models.query.QuerySet), 1125

C

cache
    template tag, 402
cache_control() (django.views.decorators.cache), 192
CACHE_MIDDLEWARE_ALIAS
    setting, 1188
CACHE_MIDDLEWARE_KEY_PREFIX
    setting, 1188
CACHE_MIDDLEWARE_SECONDS
    setting, 1188
cached.Loader (django.template.loaders), 1293
cached_property (django.utils.functional), 1319
CACHES
    setting, 1186
CACHES-BACKEND
    setting, 1187
CACHES-KEY_FUNCTION
    setting, 1187
CACHES-KEY_PREFIX
    setting, 1187
CACHES-LOCATION

setting, 1187
CACHES-OPTIONS
setting, 1187
CACHES-TIMEOUT
setting, 1188
CACHES-VERSION
setting, 1188
CallbackFilter (django.utils.log), 478
capfirst
template filter, 1263
CASCADE (django.db.models), 1066
Case (django.db.models.expressions), 1155
Cast (django.db.models.functions), 1158
center
template filter, 1263
Centroid (django.contrib.gis.db.models.functions), 799
centroid (Polygon), 830
changepassword
django-admin command, 964
changepassword
–database DATABASE, 964
CharField (django.db.models), 1056
CharField (django.forms), 1004
check
django-admin command, 946
check
–deploy, 947
–fail-level {CRITICAL,ERROR,WARNING,INFO,DEBUG}, 947
–list-tags, 947
–tag TAGS, -t TAGS, 946
check_for_language() (django.utils.translation), 1327
check_password() (django.contrib.auth.hashers), 374
CheckboxInput (django.forms), 1027
CheckboxSelectMultiple (django.forms), 1029
CheckMessage (django.core.checks), 617
ChoiceField (django.forms), 1004
choices (Field), 1050
choices (Select), 1027
chunks() (File), 974
city() (GeoIP), 842
city() (GeoIP2), 845
city_info (GeoIP), 843
clean() (Field), 999
clean() (Form), 982
clean() (Model), 1093
clean_savepoints() (django.db.transaction), 148
ClearableFileInput (django.forms), 1029
clearsessions
django-admin command, 965
Client (django.test), 315
client (Response), 319
client.RedirectCycleError, 973
close() (FieldFile), 1060
close() (File), 975
closed (LineString), 814
Coalesce (django.db.models.functions), 1158
Collect (django.contrib.gis.db.models), 795
collectstatic
django-admin command, 901
collectstatic
–clear, -c, 902
–dry-run, -n, 902
–ignore PATTERN, -i PATTERN, 902
–link, -l, 902
–no-default-ignore, 902
–no-post-process, 902
–noinput, –no-input, 902
ComboField (django.forms), 1013
CommandError, 528
CommaSeparatedIntegerField (django.db.models), 1056
comment
template tag, 1247
commit() (django.db.transaction), 147
CommonMiddleware (django.middleware.common), 1036
CommonPasswordValidator (django.contrib.auth.password_validation), 376
compilemessages
django-admin command, 947
compilemessages
–exclude EXCLUDE, -x EXCLUDE, 947
–locale LOCALE, -l LOCALE, 947
–use-fuzzy, -f, 947
Concat (django.db.models.functions), 1159
concrete (Field), 1075
concrete model, **1343**
condition() (django.views.decorators.http), 191
conditional_escape() (django.utils.html), 1322
ConditionalGetMiddleware (django.middleware.http), 1038
CONN_MAX_AGE
setting, 1192
connect() (Signal), 512
contained
field lookup type, 782
contains
field lookup type, 1131
contains_properly
field lookup type, 782
content (Response), 319
ContentFile (django.core.files.base), 975
ContentType (django.contrib.contenttypes.models), 736
ContentTypeManager (django.contrib.contenttypes.models), 737
Context (django.template), 1284

context (Response), 319
ContextPopException, 1287
cookie_date() (django.utils.http), 1323
cookies (Client), 320
coords() (GeoIP), 843
coords() (GeoIP2), 845
CoordTransform (django.contrib.gis.gdal), 835
copy() (QueryDict), 1175
Corr (django.contrib.postgres.aggregates), 864
Count (django.db.models), 1139
count (Paginator), 483
count() (django.db.models.query.QuerySet), 1126
country() (GeoIP), 842
country() (GeoIP2), 845
coupling
    loose, 1336
CovarPop (django.contrib.postgres.aggregates), 865
coveredby
    field lookup type, 782
covers
    field lookup type, 783
create() (django.db.models.query.QuerySet), 1123
create_test_db() (django.db.connection.creation), 345
createcachetable
    django-admin command, 948
createcachetable
    –database DATABASE, 948
    –dry-run, 948
CreateExtension (django.contrib.postgres.operations), 881
CreateModel (django.db.migrations.operations), 1043
createsuperuser
    django-admin command, 964
createsuperuser
    –database DATABASE, 965
    –email EMAIL, 964
    –username USERNAME, 964
Critical (django.core.checks), 617
crosses
    field lookup type, 783
CSRF_COOKIE_AGE
    setting, 1188
CSRF_COOKIE_DOMAIN
    setting, 1189
CSRF_COOKIE_HTTPONLY
    setting, 1189
CSRF_COOKIE_NAME
    setting, 1189
CSRF_COOKIE_PATH
    setting, 1189
CSRF_COOKIE_SECURE
    setting, 1189
csrf_exempt() (django.views.decorators.csrf), 930
CSRF_FAILURE_VIEW
    setting, 1190
CSRF_HEADER_NAME
    setting, 1190
csrf_protect() (django.views.decorators.csrf), 928
csrf_token
    template tag, 1247
CSRF_TRUSTED_ORIGINS
    setting, 1190
CsrfViewMiddleware (django.middleware.csrf), 1041
CurrentSiteMiddleware (django.contrib.sites.middleware), 1041
cut
    template filter, 1263
cycle
    template tag, 1247

D

D (django.contrib.gis.measure), 805
data (BoundField), 993
data() (GDALBand), 840
DATA_UPLOAD_MAX_MEMORY_SIZE
    setting, 1195
DATA_UPLOAD_MAX_NUMBER_FIELDS
    setting, 1196
DATABASE-ATOMIC_REQUESTS
    setting, 1191
DATABASE-AUTOCOMMIT
    setting, 1191
DATABASE-ENGINE
    setting, 1191
DATABASE-TEST
    setting, 1193
DATABASE-TIME_ZONE
    setting, 1192
DATABASE_ROUTERS
    setting, 1196
DatabaseError, 972
DATABASES
    setting, 1190
DataError, 972
DATAFILE
    setting, 1195
DATAFILE_MAXSIZE
    setting, 1195
DATAFILE_TMP
    setting, 1195
DATAFILE_TMP_MAXSIZE
    setting, 1195
DataSource (django.contrib.gis.gdal), 820
date
    field lookup type, 1134
    template filter, 1263
DATE_FORMAT
    setting, 1196

DATE_INPUT_FORMATS
setting, 1196
DateDetailView (django.views.generic.dates), 646
DateField (django.db.models), 1056
DateField (django.forms), 1005
DateInput (django.forms), 1026
DateMixin (django.views.generic.dates), 659
DateRangeField (django.contrib.postgres.fields), 874
DateRangeField (django.contrib.postgres.forms), 880
dates() (django.db.models.query.QuerySet), 1109
DATETIME_FORMAT
setting, 1196
DATETIME_INPUT_FORMATS
setting, 1197
DateTimeField (django.db.models), 1057
DateTimeField (django.forms), 1006
DateTimeInput (django.forms), 1026
DateTimeRangeField (django.contrib.postgres.fields), 874
DateTimeRangeField (django.contrib.postgres.forms), 879
datetimes() (django.db.models.query.QuerySet), 1110
day
field lookup type, 1135
day (DayMixin), 658
DayArchiveView (django.views.generic.dates), 644
DayMixin (django.views.generic.dates), 658
db (QuerySet), 1102
db_column (Field), 1052
db_for_read(), 152
db_for_write(), 152
db_index (Field), 1052
db_table (Options), 1083
db_type() (Field), 1073
dbshell
django-admin command, 948
dbshell
–database DATABASE, 948
deactivate() (django.utils.timezone), 1325
deactivate() (django.utils.translation), 1327
deactivate_all() (django.utils.translation), 1327
DEBUG
setting, 1197
debug
template tag, 1248
Debug (django.core.checks), 617
debug() (django.template.context_processors), 1290
DEBUG_PROPAGATE_EXCEPTIONS
setting, 1198
DECIMAL_SEPARATOR
setting, 1198
DecimalField (django.db.models), 1057
DecimalField (django.forms), 1006
DecimalValidator (django.core.validators), 1332
decorator_from_middleware() (django.utils.decorators), 1316
decorator_from_middleware_with_args() (django.utils.decorators), 1316
default
template filter, 1265
default (Field), 1052
DEFAULT_CHARSET
setting, 1198
DEFAULT_CONTENT_TYPE
setting, 1198
DEFAULT_EXCEPTION_REPORTER_FILTER
setting, 1198
DEFAULT_FILE_STORAGE
setting, 1199
DEFAULT_FROM_EMAIL
setting, 1199
default_if_none
template filter, 1265
DEFAULT_INDEX_TABLESPACE
setting, 1199
DEFAULT_TABLESPACE
setting, 1199
defaults.bad_request() (django.views), 1334
defaults.page_not_found() (django.views), 1333
defaults.permission_denied() (django.views), 1333
defaults.server_error() (django.views), 1333
DefaultStorage (django.core.files.storage), 976
defer() (django.db.models.query.QuerySet), 1119
delete() (Client), 317
delete() (django.db.models.query.QuerySet), 1130
delete() (File), 975
delete() (Model), 1097
delete() (Storage), 977
DeleteModel (django.db.migrations.operations), 1043
desc() (Expression), 1152
description (Field), 1073
destroy_test_db() (django.db.connection.creation), 345
dict() (QueryDict), 1176
dictsort
template filter, 1265
dictsortreversed
template filter, 1267
Difference (django.contrib.gis.db.models.functions), 799
diffsettings
django-admin command, 948
diffsettings
–all, 948
dim (GeometryField), 771
dims (GEOSGeometry), 808
disabled (Field), 1003
DISALLOWED_USER_AGENTS
setting, 1199
DiscoverRunner (django.test.runner), 343

disjoint
field lookup type, 783
Distance (django.contrib.gis.db.models.functions), 799
Distance (django.contrib.gis.measure), 805
distance_gt
field lookup type, 787
distance_gte
field lookup type, 788
distance_lt
field lookup type, 788
distance_lte
field lookup type, 788
distinct (Count), 1139
distinct() (django.db.models.query.QuerySet), 1106
divisibleby
template filter, 1267
django-admin command
changepassword, 964
check, 946
clearsessions, 965
collectstatic, 901
compilemessages, 947
createcachetable, 948
createsuperuser, 964
dbshell, 948
diffsettings, 948
dumpdata, 948
findstatic, 903
flush, 949
help, 946
inspectdb, 950
loaddata, 950
makemessages, 953
makemigrations, 954
migrate, 955
ogrinspect, 850
ping_google, 894
runserver, 903, 955
sendtestemail, 957
shell, 958
showmigrations, 958
sqlflush, 959
sqlmigrate, 959
sqlsequencereset, 959
squashmigrations, 959
startapp, 960
startproject, 961
test, 961
testserver, 963
version, 946
django.apps (), 611
django.conf.urls (), 1312
django.conf.urls.i18n (), 445
django.contrib.admin (), 678
django.contrib.admindocs (), 684
django.contrib.auth (), 392
django.contrib.auth.backends (), 734
django.contrib.auth.forms (), 365
django.contrib.auth.hashers (), 374
django.contrib.auth.middleware (), 1041
django.contrib.auth.password_validation (), 374
django.contrib.auth.signals (), 733
django.contrib.auth.views (), 357
django.contrib.contenttypes (), 736
django.contrib.contenttypes.admin (), 741
django.contrib.contenttypes.fields (), 738
django.contrib.contenttypes.forms (), 741
django.contrib.flatpages (), 742
django.contrib.gis (), 747
django.contrib.gis.admin (), 850
django.contrib.gis.db.backends (), 773
django.contrib.gis.db.models (), 769, 773
django.contrib.gis.db.models.functions (), 777, 796
django.contrib.gis.feeds (), 851
django.contrib.gis.forms (), 778
django.contrib.gis.gdal (), 820
django.contrib.gis.geoip (), 841
django.contrib.gis.geoip2 (), 844
django.contrib.gis.geos (), 805
django.contrib.gis.measure (), 803
django.contrib.gis.serializers.geojson (), 848
django.contrib.gis.utils (), 846
django.contrib.gis.utils.layermapping (), 846
django.contrib.gis.utils.ogrinspect (), 848
django.contrib.gis.widgets (), 779
django.contrib.humanize (), 855
django.contrib.messages (), 857
django.contrib.messages.middleware (), 1038
django.contrib.postgres (), 863
django.contrib.postgres.aggregates (), 864
django.contrib.postgres.forms.BaseRangeField (django.contrib.postgres.fields), 876
django.contrib.postgres.validators (), 885
django.contrib.redirects (), 885
django.contrib.sessions (), 205
django.contrib.sessions.middleware (), 1041
django.contrib.sitemaps (), 887
django.contrib.sites (), 894
django.contrib.sites.middleware (), 1041
django.contrib.staticfiles (), 901
django.contrib.syndication (), 907
django.core.checks (), 515
django.core.exceptions (), 970
django.core.files (), 974
django.core.files.storage (), 976
django.core.files.uploadedfile (), 978
django.core.files.uploadhandler (), 980
django.core.mail (), 416

django.core.mail.outbox (django.core.mail), 336
django.core.management (), 523
django.core.paginator (), 480
django.core.signals (), 1235
django.core.signing (), 414
django.core.validators (), 1328
django.db (), 84
django.db.backends (), 1236
django.db.backends.base.schema (), 1182
django.db.migrations (), 294
django.db.migrations.operations (), 1042
django.db.models (), 84
django.db.models.fields (), 1050
django.db.models.fields.related (), 1065
django.db.models.functions (), 1157
django.db.models.functions.datetime (), 1162
django.db.models.lookups (), 1142
django.db.models.options (), 1075
django.db.models.signals (), 1229
django.db.transaction (), 141
django.dispatch (), 511
django.forms (), 981
django.forms.fields (), 998
django.forms.formsets (), 227, 1019
django.forms.models (), 237, 1017
django.forms.widgets (), 1019
django.http (), 1168
django.middleware (), 1036
django.middleware.cache (), 1036
django.middleware.clickjacking (), 675, 1041
django.middleware.common (), 1036
django.middleware.csrf (), 925, 1041
django.middleware.exception (), 1037
django.middleware.gzip (), 1037
django.middleware.http (), 1038
django.middleware.locale (), 1038
django.middleware.security (), 1039
django.shortcuts (), 196
django.template (), 259
django.template.backends (), 262
django.template.backends.django (), 262
django.template.backends.jinja2 (), 263
django.template.loader (), 260
django.template.response (), 1299
django.test (), 309
django.test.signals (), 1235
django.test.utils (), 345
django.urls (), 1309
django.utils (), 1314
django.utils.cache (), 1314
django.utils.dateparse (), 1315
django.utils.decorators (), 1316
django.utils.encoding (), 1316
django.utils.feedgenerator (), 1318
django.utils.functional (), 1319
django.utils.html (), 1322
django.utils.http (), 1323
django.utils.log (), 470
django.utils.module_loading (), 1324
django.utils.safestring (), 1324
django.utils.six (), 489
django.utils.text (), 1325
django.utils.timezone (), 1325
django.utils.translation (), 426, 1326
django.views (), 1332
django.views.decorators.cache (), 192
django.views.decorators.csrf (), 927
django.views.decorators.gzip (), 192
django.views.decorators.http (), 191
django.views.decorators.vary (), 192
django.views.generic.dates (), 638
django.views.i18n (), 439
DJANGO_SETTINGS_MODULE, 24, 511, 568, 898, 945, 1446
DjangoTemplates (django.template.backends.django), 262
DO_NOTHING (django.db.models), 1067
Don't repeat yourself, 1336
Driver (django.contrib.gis.gdal), 826
driver (GDALRaster), 835
DRY, 1336
dumpdata
    django-admin command, 948
dumpdata
    –all, -a, 949
    –database DATABASE, 949
    –exclude EXCLUDE, -e EXCLUDE, 949
    –format FORMAT, 949
    –indent INDENT, 949
    –natural-foreign, 949
    –natural-primary, 949
    –output OUTPUT, -o OUTPUT, 949
    –pks PRIMARY_KEYS, 949
dumps() (django.core.signing), 416
DurationField (django.db.models), 1058
DurationField (django.forms), 1007
dwithin
    field lookup type, 788

E

earliest() (django.db.models.query.QuerySet), 1127
editable (Field), 1053
eggs.Loader (django.template.loaders), 1293
email (models.User), 729
EMAIL_BACKEND
    setting, 1199
EMAIL_FILE_PATH
    setting, 1199

EMAIL_HOST
setting, 1199
EMAIL_HOST_PASSWORD
setting, 1200
EMAIL_HOST_USER
setting, 1200
EMAIL_PORT
setting, 1200
EMAIL_SSL_CERTFILE
setting, 1200
EMAIL_SSL_KEYFILE
setting, 1201
EMAIL_SUBJECT_PREFIX
setting, 1200
EMAIL_TIMEOUT
setting, 1201
EMAIL_USE_SSL
setting, 1200
EMAIL_USE_TLS
setting, 1200
EmailField (django.db.models), 1058
EmailField (django.forms), 1007
EmailInput (django.forms), 1026
EmailMessage (django.core.mail), 420
EmailValidator (django.core.validators), 1330
EmptyPage, 483
Enclosure (django.utils.feedgenerator), 1319
end_index() (Page), 484
endswith
field lookup type, 1133
Engine (django.template), 1282
engines (django.template.loader), 262
ensure_csrf_cookie() (django.views.decorators.csrf), 931
Envelope (django.contrib.gis.db.models.functions), 799
Envelope (django.contrib.gis.gdal), 831
equals
field lookup type, 783
Error, 972
Error (django.core.checks), 617
errors (BoundField), 994
errors (Form), 983
escape
template filter, 1267
escape() (django.utils.html), 1322
escape_uri_path() (django.utils.encoding), 1317
escapejs
template filter, 1267
etag() (django.views.decorators.http), 192
ewkb (GEOSGeometry), 810
ewkt (GEOSGeometry), 809
ewkt (OGRGeometry), 828
exact
field lookup type, 783, 1131
ExceptionMiddleware (django.middleware.exception), 1037
exclude() (django.db.models.query.QuerySet), 1103
exists() (django.db.models.query.QuerySet), 1128
exists() (Storage), 977
Expression (django.db.models), 1151
ExpressionWrapper (django.db.models), 1150
extends
template tag, 1249
Extent (django.contrib.gis.db.models), 795
extent (GDALRaster), 837
extent (Layer), 822
Extent3D (django.contrib.gis.db.models), 795
extra() (django.db.models.query.QuerySet), 1116
Extract (django.db.models.functions.datetime), 1162
ExtractDay (django.db.models.functions.datetime), 1163
ExtractHour (django.db.models.functions.datetime), 1164
ExtractMinute (django.db.models.functions.datetime), 1164
ExtractMonth (django.db.models.functions.datetime), 1163
ExtractSecond (django.db.models.functions.datetime), 1164
ExtractWeekDay (django.db.models.functions.datetime), 1163
ExtractYear (django.db.models.functions.datetime), 1163

F

F (django.db.models), 1146
Feature (django.contrib.gis.gdal), 823
Feature release, **1655**
Feed (django.contrib.gis.feeds), 851
FetchFromCacheMiddleware (django.middleware.cache), 1036
fid (Feature), 824
field, **1343**
field (BoundField), 994
Field (django.contrib.gis.gdal), 824
Field (django.db.models), 1072
Field (django.forms), 998
field lookup type
arrayfield.contained_by, 868
arrayfield.contains, 868
arrayfield.index, 868
arrayfield.len, 868
arrayfield.overlap, 868
arrayfield.slice, 869
bbcontains, 781
bboverlaps, 782
contained, 782
contains, 1131
contains_properly, 782
coveredby, 782

covers, 783
crosses, 783
date, 1134
day, 1135
disjoint, 783
distance_gt, 787
distance_gte, 788
distance_lt, 788
distance_lte, 788
dwithin, 788
endswith, 1133
equals, 783
exact, 783, 1131
gis-contains, 782
gt, 1133
gte, 1133
hour, 1136
hstorefield.contained_by, 871
hstorefield.contains, 871
hstorefield.has_any_keys, 871
hstorefield.has_key, 871
hstorefield.has_keys, 871
hstorefield.key, 870
hstorefield.keys, 872
hstorefield.values, 872
icontains, 1132
iendswith, 1134
iexact, 1131
in, 1132
intersects, 783
iregex, 1138
isnull, 1137
istartswith, 1133
isvalid, 784
jsonfield.contained_by, 873
jsonfield.contains, 873
jsonfield.has_any_keys, 873
jsonfield.has_key, 873
jsonfield.has_keys, 873
jsonfield.key, 873
left, 785
lt, 1133
lte, 1133
minute, 1136
month, 1135
overlaps, 784
overlaps_above, 786
overlaps_below, 786
overlaps_left, 786
overlaps_right, 786
range, 1134
rangefield.adjacent_to, 876
rangefield.contained_by, 875
rangefield.contains, 875
rangefield.endswith, 876
rangefield.fully_gt, 875
rangefield.fully_lt, 875
rangefield.isempty, 876
rangefield.not_gt, 876
rangefield.not_lt, 875
rangefield.overlap, 875
rangefield.startswith, 876
regex, 1138
relate, 784
right, 786
same_as, 783
search, 882, 1137
second, 1137
startswith, 1133
strictly_above, 787
strictly_below, 787
touches, 785
trigram_similar, 880
unaccent, 881
week_day, 1136
within, 785
year, 1135
field_order (Form), 992
FieldDoesNotExist, 970
FieldError, 971
FieldFile (django.db.models.fields.files), 1060
fields (ComboField), 1013
fields (Feature), 824
fields (Form), 985
fields (Layer), 822
fields (ModelAdmin), 690
File (django.core.files), 974
file (File), 974
FILE_CHARSET
setting, 1201
FILE_UPLOAD_DIRECTORY_PERMISSIONS
setting, 1201
FILE_UPLOAD_HANDLERS
setting, 1201
FILE_UPLOAD_MAX_MEMORY_SIZE
setting, 1201
FILE_UPLOAD_PERMISSIONS
setting, 1202
FILE_UPLOAD_TEMP_DIR
setting, 1202
FileField (django.db.models), 1058
FileField (django.forms), 1008
FileInput (django.forms), 1029
filepath_to_uri() (django.utils.encoding), 1317
FilePathField (django.db.models), 1061
FilePathField (django.forms), 1008
FileResponse (django.http), 1182
FILES (HttpRequest), 1170

filesizeformat
template filter, 1268
filesystem.Loader (django.template.loaders), 1292
FileSystemStorage (django.core.files.storage), 976
FileUploadHandler (django.core.files.uploadhandler), 980
filter
template tag, 1249
filter() (django.db.models.query.QuerySet), 1103
findstatic
django-admin command, 903
findstatic
findstatic –first, 903
findstatic –first
findstatic , 903
first
template filter, 1268
first() (django.db.models.query.QuerySet), 1127
FIRST_DAY_OF_WEEK
setting, 1202
firstof
template tag, 1249
FixedOffset (django.utils.timezone), 1325
FIXTURE_DIRS
setting, 1202
FlatPage (django.contrib.flatpages.models), 745
FlatpageFallbackMiddleware (django.contrib.flatpages.middleware), 743
FlatPageSitemap (django.contrib.flatpages.sitemaps), 746
flatten() (Context), 1288
FloatField (django.db.models), 1062
FloatField (django.forms), 1009
floatformat
template filter, 1268
FloatRangeField (django.contrib.postgres.fields), 874
FloatRangeField (django.contrib.postgres.forms), 879
flush
django-admin command, 949
flush
–database DATABASE, 950
–noinput, –no-input, 949
for
template tag, 1250
force_bytes() (django.utils.encoding), 1317
force_escape
template filter, 1269
FORCE_SCRIPT_NAME
setting, 1203
force_str() (django.utils.encoding), 1317
force_text() (django.utils.encoding), 1317
force_unicode() (django.utils.encoding), 1317
ForceRHR (django.contrib.gis.db.models.functions), 800
ForeignKey (django.db.models), 1065
form (BoundField), 994
Form (django.forms), 982
form (ModelAdmin), 693
format (DateInput), 1026
format (TimeInput), 1026
format file, **470**
format_html() (django.utils.html), 1322
format_html_join() (django.utils.html), 1322
FORMAT_MODULE_PATH
setting, 1203
formfield() (Field), 1074
formset_factory() (django.forms.formsets), 1019
from_bbox() (django.contrib.gis.gdal.OGRGeometry class method), 826
from_bbox() (django.contrib.gis.geos.Polygon class method), 814
from_db() (django.db.models.Model class method), 1090
from_queryset() (django.db.models), 134
fromfile() (django.contrib.gis.geos), 816
fromstr() (django.contrib.gis.geos), 817
Func (django.db.models), 1148
function (Func), 1148

G

GDAL_LIBRARY_PATH
setting, 841
GDALBand (django.contrib.gis.gdal), 839
GDALRaster (django.contrib.gis.gdal), 835
generic view, **1343**
generic_inlineformset_factory() (django.contrib.contenttypes.forms), 741
GenericForeignKey (django.contrib.contenttypes.fields), 738
GenericInlineModelAdmin (django.contrib.contenttypes.admin), 741
GenericIPAddressField (django.db.models), 1063
GenericIPAddressField (django.forms), 1010
GenericRelation (django.contrib.contenttypes.fields), 740
GenericSitemap (django.contrib.sitemaps), 890
GenericStackedInline (django.contrib.contenttypes.admin), 742
GenericTabularInline (django.contrib.contenttypes.admin), 742
GeoAtom1Feed (django.contrib.gis.feeds), 852
GeoHash (django.contrib.gis.db.models.functions), 800
GeoIP (django.contrib.gis.geoip), 842
GeoIP2 (django.contrib.gis.geoip2), 844
GEOIP_CITY
setting, 842, 844
GEOIP_COUNTRY
setting, 842, 844
GEOIP_LIBRARY_PATH
setting, 842

GEOIP_PATH
    setting, 842, 844
geom (Feature), 823
geom_type (Field), 779
geom_type (Layer), 822
GeoManager (django.contrib.gis.db.models), 772
geometry() (Feed), 852
GeometryCollection (django.contrib.gis.gdal), 830
GeometryCollection (django.contrib.gis.geos), 816
GeometryCollectionField (django.contrib.gis.db.models), 770
GeometryCollectionField (django.contrib.gis.forms), 779
GeometryField (django.contrib.gis.db.models), 769
GeometryField (django.contrib.gis.forms), 779
GeoModelAdmin (django.contrib.gis.admin), 850
GeoQuerySet (django.contrib.gis.db.models), 781
GeoRSSFeed (django.contrib.gis.feeds), 852
geos (OGRGeometry), 827
geos() (GeoIP), 843
geos() (GeoIP2), 845
GEOS_LIBRARY_PATH
    setting, 820
GEOSException, 820
GEOSGeometry (django.contrib.gis.geos), 808
get (Feature), 824
GET (HttpRequest), 1169
get() (Client), 315
get() (Context), 1287
get() (django.db.models.query.QuerySet), 1122
get() (QueryDict), 1174
get_autocommit() (django.db.transaction), 147
get_available_languages
    template tag, 438
get_available_name() (django.core.files.storage), 561
get_cache_key() (django.utils.cache), 1315
get_connection() (django.core.mail), 422
get_current_language
    template tag, 438
get_current_language_bidi
    template tag, 438
get_current_timezone
    template tag, 463
get_current_timezone() (django.utils.timezone), 1325
get_current_timezone_name() (django.utils.timezone), 1325
get_default_timezone() (django.utils.timezone), 1325
get_default_timezone_name() (django.utils.timezone), 1325
get_digit
    template filter, 1269
get_fixed_timezone() (django.utils.timezone), 1325
get_flatpages
    template tag, 746
get_language() (django.utils.translation), 1327
get_language_bidi() (django.utils.translation), 1327
get_language_from_request() (django.utils.translation), 1328
get_language_info
    template tag, 438
get_language_info() (django.utils.translation), 434
get_language_info_list
    template tag, 439
get_list_or_404() (django.shortcuts), 199
get_lookup() (django.db.models), 1143
get_max_age() (django.utils.cache), 1314
get_media_prefix
    template tag, 1281
get_messages() (django.contrib.messages), 859
get_model() (apps), 615
get_model_class() (django.contrib.sessions.backends.db.SessionStore class method), 215
get_object_or_404() (django.shortcuts), 198
get_or_create() (django.db.models.query.QuerySet), 1123
get_password_validators() (django.contrib.auth.password_validation), 376
get_rollback() (django.db.transaction), 148
get_script_prefix() (django.urls), 1312
get_session_store_class() (django.contrib.sessions.base_session.AbstractBaseSession class method), 215
get_static_prefix
    template tag, 1280
get_storage_class() (django.core.files.storage), 976
get_tag_uri() (django.utils.feedgenerator), 1318
get_template() (django.template.loader), 260
get_transform() (django.db.models), 1143
get_user() (django.contrib.auth), 736
get_user_model() (django.contrib.auth), 383
get_valid_name() (django.core.files.storage), 561
gettext() (django.utils.translation), 1326
gettext_lazy() (django.utils.translation), 1326
gettext_noop() (django.utils.translation), 1327
gis-contains
    field lookup type, 782
gml (OGRGeometry), 827
gml() (GeoQuerySet), 793
Greatest (django.db.models.functions), 1159
gt
    field lookup type, 1133
gte
    field lookup type, 1133
gzip_page() (django.views.decorators.gzip), 192
GZipMiddleware (django.middleware.gzip), 1037

H

handler400 (django.conf.urls), 1313

handler403 (django.conf.urls), 1313
handler404 (django.conf.urls), 1314
handler500 (django.conf.urls), 1314
has_error() (Form), 984
has_next() (Page), 483
hasz (GEOSGeometry), 809
head() (Client), 317
height (GDALBand), 839
height (GDALRaster), 836
height (ImageFile), 975
help
    django-admin command, 946
help (BaseCommand), 525
help_text (Field), 1002, 1053
hex (GEOSGeometry), 809
hex (OGRGeometry), 828
hidden (Field), 1075
HiddenInput (django.forms), 1026
HOST
    setting, 1191
hour
    field lookup type, 1136
HStoreExtension (django.contrib.postgres.operations), 881
HStoreField (django.contrib.postgres.fields), 869
HStoreField (django.contrib.postgres.forms), 879
hstorefield.contained_by
    field lookup type, 871
hstorefield.contains
    field lookup type, 871
hstorefield.has_any_keys
    field lookup type, 871
hstorefield.has_key
    field lookup type, 871
hstorefield.has_keys
    field lookup type, 871
hstorefield.key
    field lookup type, 870
hstorefield.keys
    field lookup type, 872
hstorefield.values
    field lookup type, 872
html_safe() (django.utils.html), 1323
http_date() (django.utils.http), 1323
HttpRequest (django.http), 1169
HttpResponse (django.http), 1176
HttpResponseBadRequest (django.http), 1180
HttpResponseForbidden (django.http), 1180
HttpResponseGone (django.http), 1180
HttpResponseNotAllowed (django.http), 1180
HttpResponseNotFound (django.http), 1180
HttpResponseNotModified (django.http), 1180
HttpResponsePermanentRedirect (django.http), 1180
HttpResponseRedirect (django.http), 1179
HttpResponseServerError (django.http), 1180

I

i18n (Sitemap), 890
i18n_patterns() (django.conf.urls.i18n), 446
icontains
    field lookup type, 1132
iendswith
    field lookup type, 1134
iexact
    field lookup type, 1131
if
    template tag, 1251
ifchanged
    template tag, 1254
IGNORABLE_404_URLS
    setting, 1203
ImageField (django.db.models), 1062
ImageField (django.forms), 1009
ImageFile (django.core.files.images), 975
import_string() (django.utils.module_loading), 1324
ImproperlyConfigured, 971
in
    field lookup type, 1132
in_bulk() (django.db.models.query.QuerySet), 1126
include
    template tag, 1255
include() (django.conf.urls), 1312
index (Feature), 824
Info (django.core.checks), 617
info (GeoIP), 843
initial (Field), 1001
initial (Form), 984
initial (Migration), 298
inlineformset_factory() (django.forms.models), 1019
InlineModelAdmin (django.contrib.admin), 714
InMemoryUploadedFile (django.core.files.uploadedfile), 979
inspectdb
    django-admin command, 950
inspectdb
    –database DATABASE, 950
INSTALLED_APPS
    setting, 1204
instance namespace, **186**
int_list_validator() (django.core.validators), 1331
int_to_base36() (django.utils.http), 1323
intcomma
    template filter, 855
IntegerField (django.db.models), 1063
IntegerField (django.forms), 1009
IntegerRangeField (django.contrib.postgres.fields), 874
IntegerRangeField (django.contrib.postgres.forms), 879
IntegrityError, 972

InterfaceError, 972
INTERNAL_IPS
    setting, 1204
InternalError, 972
internationalization, **469**
Intersection (django.contrib.gis.db.models.functions), 800
intersects
    field lookup type, 783
intword
    template filter, 855
InvalidPage, 483
iregex
    field lookup type, 1138
iri_to_uri() (django.utils.encoding), 1317
iriencode
    template filter, 1269
is_active (django.contrib.auth), 388
is_aware() (django.utils.timezone), 1326
is_bound (Form), 982
is_naive() (django.utils.timezone), 1326
is_password_usable() (django.contrib.auth.hashers), 374
is_protected_type() (django.utils.encoding), 1317
is_staff (django.contrib.auth), 388
is_valid() (Form), 982
isnull
    field lookup type, 1137
istartswith
    field lookup type, 1133
isvalid
    field lookup type, 784
IsValid (django.contrib.gis.db.models.functions), 800
items (Sitemap), 888
items() (QueryDict), 1174
iterator() (django.db.models.query.QuerySet), 1126

J

Java, 580
javascript_catalog() (django.views.i18n), 440
JavaScriptCatalog (django.views.i18n), 439
Jinja2 (django.template.backends.jinja2), 263
join
    template filter, 1270
json (GEOSGeometry), 810
json (OGRGeometry), 828
json() (Response), 319
json_catalog() (django.views.i18n), 444
JSONCatalog (django.views.i18n), 444
JSONField (django.contrib.postgres.fields), 872
JSONField (django.contrib.postgres.forms), 879
jsonfield.contained_by
    field lookup type, 873
jsonfield.contains
    field lookup type, 873
jsonfield.has_any_keys
    field lookup type, 873
jsonfield.has_key
    field lookup type, 873
jsonfield.has_keys
    field lookup type, 873
jsonfield.key
    field lookup type, 873
JsonResponse (django.http), 1180
JVM, 580
Jython, 580
JYTHONPATH, 580

K

keep_lazy() (django.utils.functional), 1321
keep_lazy_text() (django.utils.functional), 1321
KeysValidator (django.contrib.postgres.validators), 885
kml (GEOSGeometry), 810
kml (OGRGeometry), 828
kml() (GeoQuerySet), 794

L

label (AppConfig), 613
label (BoundField), 994
label (Field), 1000
label (Options), 1089
LabelCommand (django.core.management), 527
language
    template tag, 438
language code, **469**
language_bidi
    template filter, 439
LANGUAGE_CODE
    setting, 1204
LANGUAGE_COOKIE_AGE
    setting, 1205
LANGUAGE_COOKIE_DOMAIN
    setting, 1205
LANGUAGE_COOKIE_NAME
    setting, 1205
LANGUAGE_COOKIE_PATH
    setting, 1205
language_name
    template filter, 439
language_name_local
    template filter, 439
language_name_translated
    template filter, 439
LANGUAGE_SESSION_KEY (django.utils.translation), 1328
LANGUAGES
    setting, 1205
last
    template filter, 1270

last() (django.db.models.query.QuerySet), 1127
last_modified() (django.views.decorators.http), 192
lastmod (Sitemap), 889
lat_lon() (GeoIP), 843
lat_lon() (GeoIP2), 845
latest() (django.db.models.query.QuerySet), 1127
Layer (django.contrib.gis.gdal), 821
LayerMapping (django.contrib.gis.utils), 847
learn_cache_key() (django.utils.cache), 1315
Least (django.db.models.functions), 1160
left
    field lookup type, 785
length
    template filter, 1270
Length (django.contrib.gis.db.models.functions), 800
Length (django.db.models.functions), 1160
length_is
    template filter, 1270
lhs (Lookup), 1144
lhs (Transform), 1144
limit (Sitemap), 890
LinearRing (django.contrib.gis.geos), 814
linebreaks
    template filter, 1270
linebreaksbr
    template filter, 1271
linenumbers
    template filter, 1271
LineString (django.contrib.gis.gdal), 830
LineString (django.contrib.gis.geos), 814
LineStringField (django.contrib.gis.db.models), 769
LineStringField (django.contrib.gis.forms), 779
listdir() (Storage), 978
lists() (QueryDict), 1175
LiveServerTestCase (django.test), 325
ljust
    template filter, 1271
ll (Envelope), 831
load
    template tag, 1256
loaddata
    django-admin command, 950
loaddata
    –app APP_LABEL, 951
    –database DATABASE, 950
    –ignorenonexistent, -i, 951
Loader (django.template.loaders.base), 1295
loads() (django.core.signing), 416
locale name, **469**
LOCALE_PATHS
    setting, 1206
LocaleMiddleware (django.middleware.locale), 1038
localization, **469**
localize
    template filter, 458
    template tag, 458
localize (Field), 1003
localtime
    template filter, 464
    template tag, 463
localtime() (django.utils.timezone), 1326
location (Sitemap), 888
locmem.Loader (django.template.loaders), 1294
LOGGING
    setting, 1206
LOGGING_CONFIG
    setting, 1206
login() (Client), 318
login() (django.contrib.auth), 350
login() (django.contrib.auth.views), 358
LOGIN_REDIRECT_URL
    setting, 1218
login_required() (django.contrib.auth.decorators), 352
LOGIN_URL
    setting, 1218
LoginRequiredMixin (django.contrib.auth.mixins), 353
logout() (Client), 318
logout() (django.contrib.auth), 351
logout() (django.contrib.auth.views), 360
LOGOUT_REDIRECT_URL
    setting, 1218
logout_then_login() (django.contrib.auth.views), 361
lon_lat() (GeoIP), 843
lon_lat() (GeoIP2), 845
Long-term support release, **1655**
Lookup (django.db.models), 1144
lookups.RegisterLookupMixin (django.db.models), 1142
lorem
    template tag, 1256
lower
    template filter, 1271
Lower (django.db.models.functions), 1160
lt
    field lookup type, 1133
lte
    field lookup type, 1133

M

mail_admins() (django.core.mail), 418
mail_managers() (django.core.mail), 418
make_aware() (django.utils.timezone), 1326
make_list
    template filter, 1271
make_naive() (django.utils.timezone), 1326
make_password() (django.contrib.auth.hashers), 374
MakeLine (django.contrib.gis.db.models), 796
makemessages
    django-admin command, 953

makemessages
–all, -a, 953
–domain DOMAIN, -d DOMAIN, 953
–exclude EXCLUDE, -x EXCLUDE, 953
–extension EXTENSIONS, -e EXTENSIONS, 953
–ignore PATTERN, -i PATTERN, 953
–keep-pot, 954
–locale LOCALE, -l LOCALE, 953
–no-default-ignore, 954
–no-location, 954
–no-wrap, 954
–symlinks, -s, 953
makemigrations
django-admin command, 954
makemigrations
–check, 955
–dry-run, 954
–empty, 954
–exit, -e, 954
–merge, 954
–name NAME, -n NAME, 954
–noinput, –no-input, 954
MakeValid (django.contrib.gis.db.models.functions), 801
managed (Options), 1085
Manager (django.db.models), 130
MANAGERS
setting, 1207
managers.CurrentSiteManager (django.contrib.sites), 898
ManyToManyField (django.db.models), 1069
mapping() (django.contrib.gis.utils), 848
mark_for_escaping() (django.utils.safestring), 1324
mark_safe() (django.utils.safestring), 1324
Max (django.db.models), 1139
max (GDALBand), 839
max_x (Envelope), 831
max_y (Envelope), 831
MaxLengthValidator (django.core.validators), 1331
MaxValueValidator (django.core.validators), 1331
mean (GDALBand), 839
MEDIA_ROOT
setting, 1207
MEDIA_URL
setting, 1207
MemoryFileUploadHandler (django.core.files.uploadhandler), 980
MemSize (django.contrib.gis.db.models.functions), 801
message file, **470**
MESSAGE_LEVEL
setting, 1219
MESSAGE_STORAGE
setting, 1220
MESSAGE_TAGS
setting, 1220
MessageMiddleware (django.contrib.messages.middleware), 1038
META (HttpRequest), 1170
method_decorator() (django.utils.decorators), 1316
MIDDLEWARE
setting, 1207
middleware.RedirectFallbackMiddleware (django.contrib.redirects), 886
MIDDLEWARE_CLASSES
setting, 1207
MiddlewareNotUsed, 971
migrate
django-admin command, 955
migrate
–database DATABASE, 955
–fake, 955
–fake-initial, 955
–noinput, –no-input, 955
–run-syncdb, 955
MIGRATION_MODULES
setting, 1208
Min (django.db.models), 1140
min (GDALBand), 839
min_x (Envelope), 831
min_y (Envelope), 831
MinimumLengthValidator (django.contrib.auth.password_validation), 376
MinLengthValidator (django.core.validators), 1332
minute
field lookup type, 1136
MinValueValidator (django.core.validators), 1331
mode (File), 974
model, **1343**
model (ContentType), 736
Model (django.db.models), 1089
model (Field), 1075
Model.DoesNotExist, 1100
ModelAdmin (django.contrib.admin), 687
ModelBackend (django.contrib.auth.backends), 734
ModelChoiceField (django.forms), 1015
ModelForm (django.forms), 237
modelform_factory() (django.forms.models), 1018
modelformset_factory() (django.forms.models), 1018
ModelMultipleChoiceField (django.forms), 1017
models.AbstractBaseUser (django.contrib.auth), 385
models.AnonymousUser (django.contrib.auth), 732
models.BaseInlineFormSet (django.forms), 252
models.BaseModelFormSet (django.forms), 246
models.BaseUserManager (django.contrib.auth), 387
models.CustomUser (django.contrib.auth), 384, 388
models.CustomUserManager (django.contrib.auth), 386
models.Group (django.contrib.auth), 733
models.LogEntry (django.contrib.admin), 726

models.Permission (django.contrib.auth), 732
models.PermissionsMixin (django.contrib.auth), 388
models.ProtectedError, 973
models.Redirect (django.contrib.redirects), 886
models.Site (django.contrib.sites), 894
models.User (django.contrib.auth), 729, 730
models.UserManager (django.contrib.auth), 732
modify_settings() (django.test), 330
module (AppConfig), 613
month
    field lookup type, 1135
month (MonthMixin), 657
MONTH_DAY_FORMAT
    setting, 1208
MonthArchiveView (django.views.generic.dates), 641
MonthMixin (django.views.generic.dates), 657
MTV, **1343**
MultiLineString (django.contrib.gis.geos), 815
MultiLineStringField (django.contrib.gis.db.models), 770
MultiLineStringField (django.contrib.gis.forms), 779
MultipleChoiceField (django.forms), 1010
MultipleHiddenInput (django.forms), 1029
MultipleObjectsReturned, 970
MultiPoint (django.contrib.gis.geos), 815
MultiPointField (django.contrib.gis.db.models), 769
MultiPointField (django.contrib.gis.forms), 779
MultiPolygon (django.contrib.gis.geos), 815
MultiPolygonField (django.contrib.gis.db.models), 770
MultiPolygonField (django.contrib.gis.forms), 779
MultiValueField (django.forms), 1013
MultiWidget (django.forms), 1023
MVC, **1343**

N

NAME
    setting, 1192
name (AppConfig), 613
name (BoundField), 994
name (ContentType), 737
name (DataSource), 821
name (Field), 824
name (FieldFile), 1060
name (File), 974
name (GDALRaster), 835
name (Layer), 821
name (models.Group), 733
name (models.Site), 894
name (OGRGeomType), 831
name (Origin), 1296
name (UploadedFile), 979
naturalday
    template filter, 856
naturaltime
    template filter, 856
never_cache() (django.views.decorators.cache), 192
ngettext() (django.utils.translation), 1327
ngettext_lazy() (django.utils.translation), 1327
non_atomic_requests() (django.db.transaction), 142
NON_FIELD_ERRORS (django.core.exceptions), 972
none() (django.db.models.query.QuerySet), 1110
noop (RunSQL), 1046
NoReverseMatch, 972
normalize_email() (django.contrib.auth.models.BaseUserManager class method), 387
normalize_username() (django.contrib.auth.models.AbstractBaseUser class method), 385
NotSupportedError, 972
now
    template tag, 1257
Now (django.db.models.functions), 1161
now() (django.utils.timezone), 1326
npgettext() (django.utils.translation), 1327
npgettext_lazy() (django.utils.translation), 1327
null (Field), 1050
NullBooleanField (django.db.models), 1063
NullBooleanField (django.forms), 1011
NullBooleanSelect (django.forms), 1027
num (OGRGeomType), 831
num_feat (Layer), 821
num_fields (Layer), 822
number (Page), 484
NUMBER_GROUPING
    setting, 1208
NumberInput (django.forms), 1025
NumericPasswordValidator (django.contrib.auth.password_validation), 376
NumGeometries (django.contrib.gis.db.models.functions), 801
NumPoints (django.contrib.gis.db.models.functions), 801

O

object_list (Page), 484
ObjectDoesNotExist, 970
objects (Model), 1082
ogr (GEOSGeometry), 810
OGRGeometry (django.contrib.gis.gdal), 826
OGRGeomType (django.contrib.gis.gdal), 831
ogrinspect
    django-admin command, 850
ogrinspect
    –blank BLANK, 850
    –decimal DECIMAL, 850
    –geom-name GEOM_NAME, 850
    –layer LAYER_KEY, 850
    –mapping, 850

–multi-geom, 850
–name-field NAME_FIELD, 850
–no-imports, 850
–null NULL, 850
–srid SRID, 850
on_commit() (django.db.transaction), 144
OneToOneField (django.db.models), 1071
only() (django.db.models.query.QuerySet), 1120
open() (django.contrib.gis.geoip.GeoIP class method), 843
open() (django.contrib.gis.geoip2.GeoIP2 class method), 845
open() (FieldFile), 1060
open() (File), 974
open() (Storage), 978
OpenLayersWidget (django.contrib.gis.widgets), 780
OperationalError, 972
OPTIONS
setting, 1192
Options (django.db.models.options), 1075
options() (Client), 317
order_by() (django.db.models.query.QuerySet), 1104
ordered (QuerySet), 1102
ordering (Options), 1086
ordinal
template filter, 857
Origin (django.template.base), 1296
origin (GDALRaster), 837
OSMGeoAdmin (django.contrib.gis.admin), 851
OSMWidget (django.contrib.gis.widgets), 780
outdim (WKBWriter), 818
outdim (WKTWriter), 819
output_field (django.db.models), 1143
overlaps
field lookup type, 784
overlaps_above
field lookup type, 786
overlaps_below
field lookup type, 786
overlaps_left
field lookup type, 786
overlaps_right
field lookup type, 786
override() (django.utils.timezone), 1325
override() (django.utils.translation), 1327
override_settings() (django.test), 330

P

Page (django.core.paginator), 483
page() (Paginator), 482
PageNotAnInteger, 483
Paginator (django.core.paginator), 482
paginator (Page), 484
parse_date() (django.utils.dateparse), 1315
parse_datetime() (django.utils.dateparse), 1315
parse_duration() (django.utils.dateparse), 1316
parse_time() (django.utils.dateparse), 1315
PASSWORD
setting, 1192
password_change() (django.contrib.auth.views), 361
password_change_done() (django.contrib.auth.views), 362
password_changed() (django.contrib.auth.password_validation), 376
PASSWORD_HASHERS
setting, 1219
password_reset() (django.contrib.auth.views), 362
password_reset_complete() (django.contrib.auth.views), 364
password_reset_confirm() (django.contrib.auth.views), 364
password_reset_done() (django.contrib.auth.views), 363
PASSWORD_RESET_TIMEOUT_DAYS
setting, 1218
password_validators_help_text_html() (django.contrib.auth.password_validation), 376
password_validators_help_texts() (django.contrib.auth.password_validation), 376
PasswordChangeForm (django.contrib.auth.forms), 366
PasswordInput (django.forms), 1026
PasswordResetForm (django.contrib.auth.forms), 366
Patch release, **1655**
patch() (Client), 317
patch_cache_control() (django.utils.cache), 1314
patch_response_headers() (django.utils.cache), 1315
patch_vary_headers() (django.utils.cache), 1315
path (AppConfig), 613
path (HttpRequest), 1169
path() (Storage), 978
Perimeter (django.contrib.gis.db.models.functions), 801
permission_required() (django.contrib.auth.decorators), 355
PermissionDenied, 971
PermissionRequiredMixin (django.contrib.auth.mixins), 355
PersistentRemoteUserMiddleware (django.contrib.auth.middleware), 1041
pgettext() (django.utils.translation), 1326
pgettext_lazy() (django.utils.translation), 1327
phone2numeric
template filter, 1272
ping_google
django-admin command, 894
ping_google() (django.contrib.sitemaps), 893
pk (Model), 1094
pluralize

template filter, 1272
Point (django.contrib.gis.gdal), 829
Point (django.contrib.gis.geos), 813
PointField (django.contrib.gis.db.models), 769
PointField (django.contrib.gis.forms), 779
PointOnSurface (django.contrib.gis.db.models.functions), 801
Polygon (django.contrib.gis.gdal), 830
Polygon (django.contrib.gis.geos), 814
PolygonField (django.contrib.gis.db.models), 769
PolygonField (django.contrib.gis.forms), 779
pop() (Context), 1287
pop() (QueryDict), 1175
PORT
setting, 1192
PositiveIntegerField (django.db.models), 1063
PositiveSmallIntegerField (django.db.models), 1063
POST (HttpRequest), 1169
post() (Client), 316
POSTGIS_VERSION
setting, 853
pprint
template filter, 1272
pre_save() (Field), 1074
precision (Field), 825
Prefetch (django.db.models), 1141
prefetch_related() (django.db.models.query.QuerySet), 1112
prefetch_related_objects() (django.db.models), 1142
prefix (Form), 998
PreparedGeometry (django.contrib.gis.geos), 816
PREPEND_WWW
setting, 1209
priority (Sitemap), 889
process_exception(), 203
process_template_response(), 203
process_view(), 202
ProgrammingError, 972
project, **1343**
property, **1343**
PROTECT (django.db.models), 1067
protocol (Sitemap), 889
proxy (Options), 1087
push() (Context), 1287
put() (Client), 317
Python Enhancement Proposals
PEP 20, 1337
PEP 234, 1126
PEP 249, 141, 144, 147, 936, 973
PEP 257, 1609
PEP 3134, 973
PEP 318, 330
PEP 3333, 82, 487, 602, 1171
PEP 343, 329
PEP 414, 485
PEP 420, 614
PEP 440, 946, 1357
PEP 8, 1599, 1609
python_2_unicode_compatible() (django.utils.encoding), 1316
PYTHONHASHSEED, 573
PYTHONPATH, 1446
PYTHONSTARTUP, 958
PYTHONWARNINGS, 574

Q

Q (django.db.models), 1141
QueryDict (django.http), 1173
queryset, **1343**
QuerySet (django.db.models.query), 1102

R

RadioSelect (django.forms), 1027
random
template filter, 1272
range
field lookup type, 1134
RangeField (django.contrib.postgres.fields), 876
rangefield.adjacent_to
field lookup type, 876
rangefield.contained_by
field lookup type, 875
rangefield.contains
field lookup type, 875
rangefield.endswith
field lookup type, 876
rangefield.fully_gt
field lookup type, 875
rangefield.fully_lt
field lookup type, 875
rangefield.isempty
field lookup type, 876
rangefield.not_gt
field lookup type, 876
rangefield.not_lt
field lookup type, 875
rangefield.overlap
field lookup type, 875
rangefield.startswith
field lookup type, 876
RangeMaxValueValidator (django.contrib.postgres.validators), 885
RangeMinValueValidator (django.contrib.postgres.validators), 885
RangeWidget (django.contrib.postgres.forms), 880
RasterField (django.contrib.gis.db.models), 770
raw() (django.db.models.query.QuerySet), 1122
raw() (Manager), 136

RawSQL (django.db.models.expressions), 1150
read() (File), 974
ready (apps), 615
ready() (AppConfig), 614
receiver() (django.dispatch), 513
redirect() (django.shortcuts), 197
redirect_to_login() (django.contrib.auth.views), 365
regex
  field lookup type, 1138
regex (RegexField), 1011
RegexField (django.forms), 1011
RegexValidator (django.core.validators), 1329
register() (django.contrib.admin), 688
register() (django.core.checks), 516
register_lookup() (django.db.models.lookups.RegisterLookupMixin class method), 1142
RegrAvgX (django.contrib.postgres.aggregates), 865
RegrAvgY (django.contrib.postgres.aggregates), 865
RegrCount (django.contrib.postgres.aggregates), 865
RegrIntercept (django.contrib.postgres.aggregates), 865
regroup
  template tag, 1257
RegrR2 (django.contrib.postgres.aggregates), 865
RegrSlope (django.contrib.postgres.aggregates), 865
RegrSXX (django.contrib.postgres.aggregates), 865
RegrSXY (django.contrib.postgres.aggregates), 865
RegrSYY (django.contrib.postgres.aggregates), 866
relate
  field lookup type, 784
RelatedManager (django.db.models.fields.related), 1079
RemoteUserBackend (django.contrib.auth.backends), 735
RemoteUserMiddleware (django.contrib.auth.middleware), 1041
RemoveField (django.db.migrations.operations), 1045
RenameField (django.db.migrations.operations), 1045
RenameModel (django.db.migrations.operations), 1043
render() (django.shortcuts), 196
render() (Template), 260, 1284
render() (Widget), 1022
render_to_response() (django.shortcuts), 197
render_to_string() (django.template.loader), 262
request (Response), 319
RequestContext (django.template), 1289
RequestFactory (django.test), 338
requests.RequestSite (django.contrib.sites), 900
require_GET() (django.views.decorators.http), 191
require_http_methods() (django.views.decorators.http), 191
require_POST() (django.views.decorators.http), 191
require_safe() (django.views.decorators.http), 191
required (Field), 999
RequireDebugFalse (django.utils.log), 479
RequireDebugTrue (django.utils.log), 479
requires_csrf_token() (django.views.decorators.csrf), 930
resolve() (django.urls), 1310
Resolver404, 972
ResolverMatch (django.urls), 1310
Response (django.test), 319
Reverse (django.contrib.gis.db.models.functions), 802
reverse() (django.db.models.query.QuerySet), 1106
reverse() (django.urls), 1309
reverse_lazy() (django.urls), 1310
RFC
  RFC 1034, 1354, 1380
  RFC 1123, 1323
  RFC 2046#section-5.2.1, 421
  RFC 2109, 1179, 1221
  RFC 2396, 1099
  RFC 2616, 1381, 1519
  RFC 2732, 1330
  RFC 3987#section-3.1, 1305, 1317
  RFC 3987#section-3.2, 1306, 1317
  RFC 4291#section-2.2, 1010, 1063
  RFC 5322, 1264
  RFC 6265, 1179, 1353, 1365, 1391
  RFC 7231, 929, 1381
  RFC 7231#section-4.2.1, 926, 1124
  RFC 7231#section-4.3.8, 317
  RFC 7231#section-6, 1177, 1178, 1182
  RFC 7231#section-6.1, 1177, 1182
  RFC 7231#section-6.5.3, 1334
  RFC 7231#section-7.1.1.1, 1323
  RFC 7231#section-7.1.4, 409, 1314
  RFC 7232#section-4.1, 412
  RFC 7234, 410
  RFC 7234#section-5.2.2.8, 410
  RFC 7239#page-7, 1217
rhs (Lookup), 1144
right
  field lookup type, 786
ring (GEOSGeometry), 809
rjust
  template filter, 1273
rollback() (django.db.transaction), 147
ROOT_URLCONF
  setting, 1209
Rss201rev2Feed (django.utils.feedgenerator), 1319
RssFeed (django.utils.feedgenerator), 1319
RssUserland091Feed (django.utils.feedgenerator), 1319
RunPython (django.db.migrations.operations), 1046
runserver
  django-admin command, 903, 955
runserver
  –insecure, 903
  –ipv6, -6, 956
  –noreload, 956

–nostatic, 903
–nothreading, 956
RunSQL (django.db.migrations.operations), 1045

S

safe
template filter, 1273
SafeBytes (django.utils.safestring), 1324
SafeExceptionReporterFilter (django.views.debug), 578
safeseq
template filter, 1273
SafeString (django.utils.safestring), 1324
SafeText (django.utils.safestring), 1324
SafeUnicode (django.utils.safestring), 1324
same_as
field lookup type, 783
sample (CovarPop), 865
sample (StdDev), 1140
sample (Variance), 1140
save() (FieldFile), 1060
save() (File), 975
save() (Model), 1094
save() (Storage), 978
savepoint() (django.db.transaction), 148
savepoint_commit() (django.db.transaction), 148
savepoint_rollback() (django.db.transaction), 148
Scale (django.contrib.gis.db.models.functions), 802
scale (GDALRaster), 837
search
field lookup type, 882, 1137
SearchQuery (django.contrib.postgres.search), 882
SearchRank (django.contrib.postgres.search), 883
SearchVector (django.contrib.postgres.search), 882
SearchVectorField (django.contrib.postgres.search), 884
second
field lookup type, 1137
SECRET_KEY
setting, 1209
SECURE_BROWSER_XSS_FILTER
setting, 1209
SECURE_CONTENT_TYPE_NOSNIFF
setting, 1210
SECURE_HSTS_INCLUDE_SUBDOMAINS
setting, 1210
SECURE_HSTS_SECONDS
setting, 1210
SECURE_PROXY_SSL_HEADER
setting, 1210
SECURE_REDIRECT_EXEMPT
setting, 1211
SECURE_SSL_HOST
setting, 1211
SECURE_SSL_REDIRECT
setting, 1211
SecurityMiddleware (django.middleware.security), 1039
Select (django.forms), 1027
select_for_update() (django.db.models.query.QuerySet), 1121
select_related() (django.db.models.query.QuerySet), 1111
select_template() (django.template.loader), 260
SelectDateWidget (django.forms), 1030
SelectMultiple (django.forms), 1027
send() (Signal), 514
send_mail() (django.core.mail), 417
send_mass_mail() (django.core.mail), 417
sendtestemail
django-admin command, 957
sendtestemail
–admins, 957
–managers, 957
sensitive_post_parameters() (django.views.decorators.debug), 577
sensitive_variables() (django.views.decorators.debug), 576
SeparateDatabaseAndState (django.db.migrations.operations), 1048
SERIALIZATION_MODULES
setting, 1211
serializers.JSONSerializer (django.contrib.sessions), 209
serializers.PickleSerializer (django.contrib.sessions), 209
SERVER_EMAIL
setting, 1212
session (Client), 320
SESSION_CACHE_ALIAS
setting, 1220
SESSION_COOKIE_AGE
setting, 1221
SESSION_COOKIE_DOMAIN
setting, 1221
SESSION_COOKIE_HTTPONLY
setting, 1221
SESSION_COOKIE_NAME
setting, 1221
SESSION_COOKIE_PATH
setting, 1221
SESSION_COOKIE_SECURE
setting, 1221
SESSION_ENGINE
setting, 1222
SESSION_EXPIRE_AT_BROWSER_CLOSE
setting, 1222
SESSION_FILE_PATH
setting, 1222
SESSION_SAVE_EVERY_REQUEST
setting, 1222
SESSION_SERIALIZER
setting, 1222

SessionMiddleware (django.contrib.sessions.middleware), 1041
SET() (django.db.models), 1067
set_autocommit() (django.db.transaction), 147
SET_DEFAULT (django.db.models), 1067
set_language() (django.views.i18n), 452
SET_NULL (django.db.models), 1067
set_rollback() (django.db.transaction), 148
SetPasswordForm (django.contrib.auth.forms), 366
setting
ABSOLUTE_URL_OVERRIDES, 1185
ADMINS, 1185
ALLOWED_HOSTS, 1186
APPEND_SLASH, 1186
AUTH_PASSWORD_VALIDATORS, 1219
AUTH_USER_MODEL, 1218
AUTHENTICATION_BACKENDS, 1218
CACHE_MIDDLEWARE_ALIAS, 1188
CACHE_MIDDLEWARE_KEY_PREFIX, 1188
CACHE_MIDDLEWARE_SECONDS, 1188
CACHES, 1186
CACHES-BACKEND, 1187
CACHES-KEY_FUNCTION, 1187
CACHES-KEY_PREFIX, 1187
CACHES-LOCATION, 1187
CACHES-OPTIONS, 1187
CACHES-TIMEOUT, 1188
CACHES-VERSION, 1188
CONN_MAX_AGE, 1192
CSRF_COOKIE_AGE, 1188
CSRF_COOKIE_DOMAIN, 1189
CSRF_COOKIE_HTTPONLY, 1189
CSRF_COOKIE_NAME, 1189
CSRF_COOKIE_PATH, 1189
CSRF_COOKIE_SECURE, 1189
CSRF_FAILURE_VIEW, 1190
CSRF_HEADER_NAME, 1190
CSRF_TRUSTED_ORIGINS, 1190
DATA_UPLOAD_MAX_MEMORY_SIZE, 1195
DATA_UPLOAD_MAX_NUMBER_FIELDS, 1196
DATABASE-ATOMIC_REQUESTS, 1191
DATABASE-AUTOCOMMIT, 1191
DATABASE-ENGINE, 1191
DATABASE-TEST, 1193
DATABASE-TIME_ZONE, 1192
DATABASE_ROUTERS, 1196
DATABASES, 1190
DATAFILE, 1195
DATAFILE_MAXSIZE, 1195
DATAFILE_TMP, 1195
DATAFILE_TMP_MAXSIZE, 1195
DATE_FORMAT, 1196
DATE_INPUT_FORMATS, 1196
DATETIME_FORMAT, 1196
DATETIME_INPUT_FORMATS, 1197
DEBUG, 1197
DEBUG_PROPAGATE_EXCEPTIONS, 1198
DECIMAL_SEPARATOR, 1198
DEFAULT_CHARSET, 1198
DEFAULT_CONTENT_TYPE, 1198
DEFAULT_EXCEPTION_REPORTER_FILTER, 1198
DEFAULT_FILE_STORAGE, 1199
DEFAULT_FROM_EMAIL, 1199
DEFAULT_INDEX_TABLESPACE, 1199
DEFAULT_TABLESPACE, 1199
DISALLOWED_USER_AGENTS, 1199
EMAIL_BACKEND, 1199
EMAIL_FILE_PATH, 1199
EMAIL_HOST, 1199
EMAIL_HOST_PASSWORD, 1200
EMAIL_HOST_USER, 1200
EMAIL_PORT, 1200
EMAIL_SSL_CERTFILE, 1200
EMAIL_SSL_KEYFILE, 1201
EMAIL_SUBJECT_PREFIX, 1200
EMAIL_TIMEOUT, 1201
EMAIL_USE_SSL, 1200
EMAIL_USE_TLS, 1200
FILE_CHARSET, 1201
FILE_UPLOAD_DIRECTORY_PERMISSIONS, 1201
FILE_UPLOAD_HANDLERS, 1201
FILE_UPLOAD_MAX_MEMORY_SIZE, 1201
FILE_UPLOAD_PERMISSIONS, 1202
FILE_UPLOAD_TEMP_DIR, 1202
FIRST_DAY_OF_WEEK, 1202
FIXTURE_DIRS, 1202
FORCE_SCRIPT_NAME, 1203
FORMAT_MODULE_PATH, 1203
GDAL_LIBRARY_PATH, 841
GEOIP_CITY, 842, 844
GEOIP_COUNTRY, 842, 844
GEOIP_LIBRARY_PATH, 842
GEOIP_PATH, 842, 844
GEOS_LIBRARY_PATH, 820
HOST, 1191
IGNORABLE_404_URLS, 1203
INSTALLED_APPS, 1204
INTERNAL_IPS, 1204
LANGUAGE_CODE, 1204
LANGUAGE_COOKIE_AGE, 1205
LANGUAGE_COOKIE_DOMAIN, 1205
LANGUAGE_COOKIE_NAME, 1205
LANGUAGE_COOKIE_PATH, 1205
LANGUAGES, 1205
LOCALE_PATHS, 1206

LOGGING, 1206
LOGGING_CONFIG, 1206
LOGIN_REDIRECT_URL, 1218
LOGIN_URL, 1218
LOGOUT_REDIRECT_URL, 1218
MANAGERS, 1207
MEDIA_ROOT, 1207
MEDIA_URL, 1207
MESSAGE_LEVEL, 1219
MESSAGE_STORAGE, 1220
MESSAGE_TAGS, 1220
MIDDLEWARE, 1207
MIDDLEWARE_CLASSES, 1207
MIGRATION_MODULES, 1208
MONTH_DAY_FORMAT, 1208
NAME, 1192
NUMBER_GROUPING, 1208
OPTIONS, 1192
PASSWORD, 1192
PASSWORD_HASHERS, 1219
PASSWORD_RESET_TIMEOUT_DAYS, 1218
PORT, 1192
POSTGIS_VERSION, 853
PREPEND_WWW, 1209
ROOT_URLCONF, 1209
SECRET_KEY, 1209
SECURE_BROWSER_XSS_FILTER, 1209
SECURE_CONTENT_TYPE_NOSNIFF, 1210
SECURE_HSTS_INCLUDE_SUBDOMAINS, 1210
SECURE_HSTS_SECONDS, 1210
SECURE_PROXY_SSL_HEADER, 1210
SECURE_REDIRECT_EXEMPT, 1211
SECURE_SSL_HOST, 1211
SECURE_SSL_REDIRECT, 1211
SERIALIZATION_MODULES, 1211
SERVER_EMAIL, 1212
SESSION_CACHE_ALIAS, 1220
SESSION_COOKIE_AGE, 1221
SESSION_COOKIE_DOMAIN, 1221
SESSION_COOKIE_HTTPONLY, 1221
SESSION_COOKIE_NAME, 1221
SESSION_COOKIE_PATH, 1221
SESSION_COOKIE_SECURE, 1221
SESSION_ENGINE, 1222
SESSION_EXPIRE_AT_BROWSER_CLOSE, 1222
SESSION_FILE_PATH, 1222
SESSION_SAVE_EVERY_REQUEST, 1222
SESSION_SERIALIZER, 1222
SHORT_DATE_FORMAT, 1212
SHORT_DATETIME_FORMAT, 1212
SIGNING_BACKEND, 1212
SILENCED_SYSTEM_CHECKS, 1212
SITE_ID, 1223
STATIC_ROOT, 1223
STATIC_URL, 1223
STATICFILES_DIRS, 1223
STATICFILES_FINDERS, 1224
STATICFILES_STORAGE, 1224
TEMPLATES, 1213
TEMPLATES-APP_DIRS, 1213
TEMPLATES-BACKEND, 1213
TEMPLATES-DIRS, 1213
TEMPLATES-NAME, 1213
TEMPLATES-OPTIONS, 1214
TEST_CHARSET, 1193
TEST_COLLATION, 1193
TEST_CREATE, 1194
TEST_DEPENDENCIES, 1194
TEST_MIRROR, 1194
TEST_NAME, 1194
TEST_NON_SERIALIZED_APPS, 1214
TEST_PASSWD, 1194
TEST_RUNNER, 1214
TEST_SERIALIZE, 1194
TEST_TBLSPACE, 1195
TEST_TBLSPACE_TMP, 1195
TEST_USER, 1194
TEST_USER_CREATE, 1194
THOUSAND_SEPARATOR, 1214
TIME_FORMAT, 1214
TIME_INPUT_FORMATS, 1214
TIME_ZONE, 1215
USE_ETAGS, 1215
USE_I18N, 1216
USE_L10N, 1216
USE_THOUSAND_SEPARATOR, 1216
USE_TZ, 1216
USE_X_FORWARDED_HOST, 1216
USE_X_FORWARDED_PORT, 1217
USER, 1193
WSGI_APPLICATION, 1217
X_FRAME_OPTIONS, 1217
YEAR_MONTH_FORMAT, 1217
setup() (django), 615
setup_test_environment() (django.test.utils), 345
setUpTestData() (django.test.TestCase class method), 324
shell
django-admin command, 958
shell
–command COMMAND, -c COMMAND, 958
–interface {ipython,bpython,python}, -i {ipython,bpython,python}, 958
–nostartup, 958
shell (Polygon), 830
SHORT_DATE_FORMAT
setting, 1212

SHORT_DATETIME_FORMAT
setting, 1212
shortcuts, 196
shortcuts.get_current_site() (django.contrib.sites), 900
showmigrations
django-admin command, 958
showmigrations
–database DATABASE, 958
–list, -l, 958
–plan, -p, 958
Signal (django.dispatch), 514
Signer (django.core.signing), 415
SIGNING_BACKEND
setting, 1212
SILENCED_SYSTEM_CHECKS
setting, 1212
SimpleArrayField (django.contrib.postgres.forms), 877
SimpleTemplateResponse (django.template.response), 1299
SimpleTestCase (django.test), 322
site (HttpRequest), 1171
SITE_ID
setting, 1223
Sitemap (django.contrib.sitemaps), 888
size (ArrayField), 867
size (FieldFile), 1060
size (File), 974
size (UploadedFile), 979
size() (Storage), 978
skew (GDALRaster), 837
skipIfDBFeature() (django.test), 337
skipUnlessDBFeature() (django.test), 338
slice
template filter, 1273
slug, **1343**
SlugField (django.db.models), 1063
SlugField (django.forms), 1012
slugify
template filter, 1274
slugify() (django.utils.text), 1325
SmallIntegerField (django.db.models), 1064
smart_bytes() (django.utils.encoding), 1317
smart_str() (django.utils.encoding), 1317
smart_text() (django.utils.encoding), 1316
smart_unicode() (django.utils.encoding), 1316
SnapToGrid (django.contrib.gis.db.models.functions), 802
spaceless
template tag, 1259
SpatialReference (django.contrib.gis.gdal), 832
SplitArrayField (django.contrib.postgres.forms), 878
SplitDateTimeField (django.forms), 1015
SplitDateTimeWidget (django.forms), 1029
SplitHiddenDateTimeWidget (django.forms), 1030
sqlflush
django-admin command, 959
sqlflush
–database DATABASE, 959
sqlmigrate
django-admin command, 959
sqlmigrate
–backwards, 959
–database DATABASE, 959
sqlsequencereset
django-admin command, 959
sqlsequencereset
–database DATABASE, 959
squashmigrations
django-admin command, 959
squashmigrations
–no-optimize, 959
–noinput, –no-input, 960
srid (Field), 779
srid (GDALRaster), 836
srid (GEOSGeometry), 809
srid (OGRGeometry), 827
srid (WKBWriter), 818
srs (GDALRaster), 836
srs (GEOSGeometry), 813
srs (Layer), 822
srs (OGRGeometry), 827
StackedInline (django.contrib.admin), 714
staff_member_required() (django.contrib.admin.views.decorators), 728
startapp
django-admin command, 960
startapp
–extension EXTENSIONS, -e EXTENSIONS, 960
–name FILES, -n FILES, 960
–template TEMPLATE, 960
startproject
django-admin command, 961
startproject
–extension EXTENSIONS, -e EXTENSIONS, 961
–name FILES, -n FILES, 961
–template TEMPLATE, 961
startswith
field lookup type, 1133
static
template tag, 1280
static() (django.template.context_processors), 1291
static.serve() (django.views), 1332
static.static() (django.conf.urls), 1312
STATIC_ROOT
setting, 1223
STATIC_URL
setting, 1223

STATICFILES_DIRS
    setting, 1223
STATICFILES_FINDERS
    setting, 1224
STATICFILES_STORAGE
    setting, 1224
std (GDALBand), 839
StdDev (django.db.models), 1140
Storage (django.core.files.storage), 976
storage (FileField), 1059
storage.base.BaseStorage (django.contrib.messages), 858
storage.base.Message (django.contrib.messages), 860
storage.CachedStaticFilesStorage
        (django.contrib.staticfiles), 905
storage.cookie.CookieStorage (django.contrib.messages
        ), 858
storage.fallback.FallbackStorage
        (django.contrib.messages), 858
storage.ManifestStaticFilesStorage
        (django.contrib.staticfiles), 904
storage.session.SessionStorage (django.contrib.messages
        ), 858
storage.StaticFilesStorage (django.contrib.staticfiles),
        904
StreamingHttpResponse (django.http), 1181
strictly_above
    field lookup type, 787
strictly_below
    field lookup type, 787
string_concat() (django.utils.translation), 1327
StringAgg (django.contrib.postgres.aggregates), 864
stringformat
    template filter, 1274
strip (CharField), 1004
strip (RegexField), 1011
strip_tags() (django.utils.html), 1322
striptags
    template filter, 1274
style (BaseCommand), 526
Substr (django.db.models.functions), 1161
Sum (django.db.models), 1140
SuspiciousOperation, 970
svg() (GeoQuerySet), 794
SymDifference (django.contrib.gis.db.models.functions),
        802
SyndicationFeed (django.utils.feedgenerator), 1318

T

TabularInline (django.contrib.admin), 714
teardown_test_environment() (django.test.utils), 345
template, **1343**
Template (django.template), 1283
template (Func), 1148
template filter
    add, 1262
    addslashes, 1263
    apnumber, 855
    capfirst, 1263
    center, 1263
    cut, 1263
    date, 1263
    default, 1265
    default_if_none, 1265
    dictsort, 1265
    dictsortreversed, 1267
    divisibleby, 1267
    escape, 1267
    escapejs, 1267
    filesizeformat, 1268
    first, 1268
    floatformat, 1268
    force_escape, 1269
    get_digit, 1269
    intcomma, 855
    intword, 855
    iriencode, 1269
    join, 1270
    language_bidi, 439
    language_name, 439
    language_name_local, 439
    language_name_translated, 439
    last, 1270
    length, 1270
    length_is, 1270
    linebreaks, 1270
    linebreaksbr, 1271
    linenumbers, 1271
    ljust, 1271
    localize, 458
    localtime, 464
    lower, 1271
    make_list, 1271
    naturalday, 856
    naturaltime, 856
    ordinal, 857
    phone2numeric, 1272
    pluralize, 1272
    pprint, 1272
    random, 1272
    rjust, 1273
    safe, 1273
    safeseq, 1273
    slice, 1273
    slugify, 1274
    stringformat, 1274
    striptags, 1274
    time, 1274
    timesince, 1275

timeuntil, 1275
timezone, 464
title, 1275
truncatechars, 1276
truncatechars_html, 1276
truncatewords, 1276
truncatewords_html, 1276
unlocalize, 458
unordered_list, 1277
upper, 1277
urlencode, 1277
urlize, 1278
urlizetrunc, 1278
utc, 464
wordcount, 1278
wordwrap, 1279
yesno, 1279
template tag
autoescape, 1246
block, 1246
blocktrans, 435
cache, 402
comment, 1247
csrf_token, 1247
cycle, 1247
debug, 1248
extends, 1249
filter, 1249
firstof, 1249
for, 1250
get_available_languages, 438
get_current_language, 438
get_current_language_bidi, 438
get_current_timezone, 463
get_flatpages, 746
get_language_info, 438
get_language_info_list, 439
get_media_prefix, 1281
get_static_prefix, 1280
if, 1251
ifchanged, 1254
include, 1255
language, 438
load, 1256
localize, 458
localtime, 463
lorem, 1256
now, 1257
regroup, 1257
spaceless, 1259
static, 1280
templatetag, 1260
timezone, 463
trans, 434
url, 1260
verbatim, 1261
widthratio, 1262
with, 1262
TemplateDoesNotExist, 260
TemplateResponse (django.template.response), 1301
TEMPLATES
setting, 1213
TEMPLATES-APP_DIRS
setting, 1213
TEMPLATES-BACKEND
setting, 1213
TEMPLATES-DIRS
setting, 1213
TEMPLATES-NAME
setting, 1213
TEMPLATES-OPTIONS
setting, 1214
TemplateSyntaxError, 260
templatetag
template tag, 1260
templatize() (django.utils.translation), 1328
TemporaryFileUploadHandler
(django.core.files.uploadhandler), 980
TemporaryUploadedFile (django.core.files.uploadedfile), 979
test
django-admin command, 961
test
–debug-sql, -d, 962
–exclude-tag EXCLUDE_TAGS, 963
–failfast, 962
–keepdb, -k, 962
–liveserver LIVESERVER, 962
–noinput, –no-input, 962
–parallel [N], 962
–reverse, -r, 962
–tag TAGS, 963
–testrunner TESTRUNNER, 962
TEST_CHARSET
setting, 1193
TEST_COLLATION
setting, 1193
TEST_CREATE
setting, 1194
TEST_DEPENDENCIES
setting, 1194
TEST_MIRROR
setting, 1194
TEST_NAME
setting, 1194
TEST_NON_SERIALIZED_APPS
setting, 1214
TEST_PASSWD

setting, 1194
TEST_RUNNER
setting, 1214
TEST_SERIALIZE
setting, 1194
TEST_TBLSPACE
setting, 1195
TEST_TBLSPACE_TMP
setting, 1195
TEST_USER
setting, 1194
TEST_USER_CREATE
setting, 1194
TestCase (django.test), 324
testing.StaticLiveServerTestCase (django.contrib.staticfiles), 907
testserver
django-admin command, 963
testserver
–addrport ADDRPORT, 963
–noinput, –no-input, 964
Textarea (django.forms), 1027
TextField (django.db.models), 1064
TextInput (django.forms), 1025
THOUSAND_SEPARATOR
setting, 1214
time
template filter, 1274
TIME_FORMAT
setting, 1214
TIME_INPUT_FORMATS
setting, 1214
TIME_ZONE
setting, 1215
TimeField (django.db.models), 1064
TimeField (django.forms), 1012
TimeInput (django.forms), 1026
timesince
template filter, 1275
TimestampSigner (django.core.signing), 415
timeuntil
template filter, 1275
timezone
template filter, 464
template tag, 463
title
template filter, 1275
to_locale() (django.utils.translation), 1328
TodayArchiveView (django.views.generic.dates), 645
touches
field lookup type, 785
trace() (Client), 317
trans
template tag, 434
TransactionManagementError, 973
TransactionNow (django.contrib.postgres.functions), 880
TransactionTestCase (django.test), 323
Transform (django.contrib.gis.db.models.functions), 802
Transform (django.db.models), 1143
Translate (django.contrib.gis.db.models.functions), 803
translation string, **470**
trigram_similar
field lookup type, 880
TrigramDistance (django.contrib.postgres.search), 885
TrigramExtension (django.contrib.postgres.operations), 881
TrigramSimilarity (django.contrib.postgres.search), 884
trim (WKTWriter), 819
Trunc (django.db.models.functions.datetime), 1165
truncatechars
template filter, 1276
truncatechars_html
template filter, 1276
truncatewords
template filter, 1276
truncatewords_html
template filter, 1276
TruncDate (django.db.models.functions.datetime), 1167
TruncDay (django.db.models.functions.datetime), 1167
TruncHour (django.db.models.functions.datetime), 1168
TruncMinute (django.db.models.functions.datetime), 1168
TruncMonth (django.db.models.functions.datetime), 1167
TruncSecond (django.db.models.functions.datetime), 1168
TruncYear (django.db.models.functions.datetime), 1166
tuple (Envelope), 832
tuple (OGRGeometry), 829
type (Field), 824
type_name (Field), 825
TypedChoiceField (django.forms), 1005
TypedMultipleChoiceField (django.forms), 1011
tz() (django.template.context_processors), 1291

U

ugettext() (django.utils.translation), 1326
ugettext_lazy() (django.utils.translation), 1326
ugettext_noop() (django.utils.translation), 1327
unaccent
field lookup type, 881
UnaccentExtension (django.contrib.postgres.operations), 881
ungettext() (django.utils.translation), 1327
ungettext_lazy() (django.utils.translation), 1327
Union (django.contrib.gis.db.models), 796
Union (django.contrib.gis.db.models.functions), 803
unique (Field), 1054

unit_attname() (django.contrib.gis.measure.Area class method), 805
unit_attname() (django.contrib.gis.measure.Distance class method), 805
unlocalize
  template filter, 458
unordered_list
  template filter, 1277
UnreadablePostError, 973
update() (Context), 1288
update() (django.db.models.query.QuerySet), 1129
update_or_create() (django.db.models.query.QuerySet), 1124
update_session_auth_hash() (django.contrib.auth), 357
UpdateCacheMiddleware (django.middleware.cache), 1036
UploadedFile (django.core.files.uploadedfile), 978
upper
  template filter, 1277
Upper (django.db.models.functions), 1161
ur (Envelope), 831
uri_to_iri() (django.utils.encoding), 1317
url
  template tag, 1260
url (FieldFile), 1060
url() (django.conf.urls), 1312
url() (Storage), 978
urlencode
  template filter, 1277
urlencode() (django.utils.http), 1323
URLField (django.db.models), 1064
URLField (django.forms), 1012
URLInput (django.forms), 1026
urlize
  template filter, 1278
urlizetrunc
  template filter, 1278
urlquote() (django.utils.http), 1323
urlquote_plus() (django.utils.http), 1323
urls
  definitive, 1338
urls.staticfiles_urlpatterns() (django.contrib.staticfiles), 906
urlsafe_base64_decode() (django.utils.http), 1323
urlsafe_base64_encode() (django.utils.http), 1323
URLValidator (django.core.validators), 1330
USE_ETAGS
  setting, 1215
USE_I18N
  setting, 1216
USE_L10N
  setting, 1216
USE_THOUSAND_SEPARATOR
  setting, 1216
USE_TZ
  setting, 1216
USE_X_FORWARDED_HOST
  setting, 1216
USE_X_FORWARDED_PORT
  setting, 1217
USER
  setting, 1193
user (HttpRequest), 1171
user (LogEntry), 726
user_logged_in() (django.contrib.auth.signals), 733
user_logged_out() (django.contrib.auth.signals), 734
user_login_failed() (django.contrib.auth.signals), 734
user_passes_test() (django.contrib.auth.decorators), 353
UserAttributeSimilarityValidator (django.contrib.auth.password_validation), 376
UserChangeForm (django.contrib.auth.forms), 366
UserCreationForm (django.contrib.auth.forms), 366
UserPassesTestMixin (django.contrib.auth.mixins), 354
using() (django.db.models.query.QuerySet), 1121
utc
  template filter, 464
utc (django.utils.timezone), 1325
UUIDField (django.db.models), 1065
UUIDField (django.forms), 1013

V

validate_comma_separated_integer_list (django.core.validators), 1331
validate_email (django.core.validators), 1330
validate_ipv46_address (django.core.validators), 1331
validate_ipv4_address (django.core.validators), 1331
validate_ipv6_address (django.core.validators), 1331
validate_password() (django.contrib.auth.password_validation), 376
validate_slug (django.core.validators), 1330
validate_unicode_slug (django.core.validators), 1330
ValidationError, 972
validators (Field), 1003, 1055
validators.ASCIIUsernameValidator (django.contrib.auth), 733
validators.UnicodeUsernameValidator (django.contrib.auth), 733
Value (django.db.models), 1150
value (Field), 825
values() (django.db.models.query.QuerySet), 1107
values_list() (django.db.models.query.QuerySet), 1109
Variance (django.db.models), 1140
vary_on_cookie() (django.views.decorators.vary), 192
vary_on_headers() (django.views.decorators.vary), 192
verbatim
  template tag, 1261
version

django-admin command, 946
view, **1343**
ViewDoesNotExist, 971
views.Feed (django.contrib.syndication), 912
views.index() (django.contrib.sitemaps), 891
views.serve() (django.contrib.staticfiles), 906
views.sitemap() (django.contrib.sitemaps), 887
views.SuccessMessageMixin (django.contrib.messages), 862

W

W3CGeoFeed (django.contrib.gis.feeds), 852
Warning (django.core.checks), 617
warp() (GDALRaster), 838
week (WeekMixin), 658
week_day
field lookup type, 1136
WeekArchiveView (django.views.generic.dates), 642
WeekMixin (django.views.generic.dates), 658
When (django.db.models.expressions), 1155
Widget (django.forms), 1022
widget (Field), 1002
width (Field), 825
width (GDALBand), 839
width (GDALRaster), 836
width (ImageFile), 975
widthratio
template tag, 1262
with
template tag, 1262
within
field lookup type, 785
wkb (GEOSGeometry), 810
wkb (OGRGeometry), 828
WKBReader (django.contrib.gis.geos), 817
WKBWriter (django.contrib.gis.geos), 817
wkt (Envelope), 832
wkt (GEOSGeometry), 810
wkt (OGRGeometry), 828
WKTReader (django.contrib.gis.geos), 817
WKTWriter (django.contrib.gis.geos), 819
wordcount
template filter, 1278
wordwrap
template filter, 1279
write() (File), 974
write() (WKBWriter), 817
write() (WKTWriter), 819
WSGI_APPLICATION
setting, 1217

X

x (LineString), 830
x (Point), 830
X_FRAME_OPTIONS
setting, 1217
XFrameOptionsMiddleware (django.middleware.clickjacking), 1041
xml
suckiness of, 1339

Y

y (LineString), 830
y (Point), 830
year
field lookup type, 1135
year (YearMixin), 657
YEAR_MONTH_FORMAT
setting, 1217
YearArchiveView (django.views.generic.dates), 639
YearMixin (django.views.generic.dates), 657
yesno
template filter, 1279

Z

z (LineString), 830
z (Point), 830