

Mastering Django: Core

The New Django Book Updated for Django 1.8LTS

Chapters 1 – 13: Django Fundamentals

For remaining chapters and references, please go to the Mastering Django website:

<http://www.masteringdjango.com>

Copyright©2016 Nigel George

Released under the GNU Free Documentation License Version 1.3

For full text of the GFDL, see the end of this document

TABLE OF CONTENTS

Front Matter	15
Acknowledgements.....	15
About the Author.....	15
Introduction	15
Why should you care about Django?.....	16
About This Book.....	17
How to Read This Book.....	18
Required Programming Knowledge.....	18
Required Python Knowledge	18
Required Django Version	19
Getting Help.....	19
Introduction to Django.....	20
Introducing Django	20
By Adrian Holovaty and Jacob Kaplan-Moss – December 2009	20
Django’s History	21
Chapter 1: Getting Started.....	22
Installing Django.....	23
Installing Python	23
Installing Django.....	25
Setting Up a Database.....	29
Starting a Project.....	29
Django settings	31
The development server.....	33
The Model-View-Controller (MVC) design pattern	35
What’s Next?.....	36
Chapter 2: Views and URLconfs	36
Your First Django-Powered Page: Hello World.....	36
Your First View	37

Your First URLconf.....	38
Regular Expressions	44
A Quick Note About 404 Errors.....	44
A Quick Note About The Site Root.....	46
How Django Processes a Request.....	46
Your Second View: Dynamic Content	49
URLconfs and Loose Coupling.....	51
Your Third View: Dynamic URLs.....	52
Django's Pretty Error Pages	58
What's next?.....	61
Chapter 3: Templates.....	62
Template System Basics.....	63
Ordering notice	63
Using the Template System	66
Creating Template Objects	66
Rendering a Template.....	68
Dictionaries and Contexts	69
Multiple Contexts, Same Template	71
Context Variable Lookup.....	72
Method Call Behavior	75
How Invalid Variables Are Handled	77
Basic Template Tags and Filters.....	77
Tags	78
Comments	86
Filters.....	87
Philosophies and Limitations	88
Separate logic from presentation	88
Discourage redundancy	88
Be decoupled from HTML	88

XML should not be used for template languages	88
Assume designer competence	89
Treat whitespace obviously	89
Don't invent a programming language	89
Safety and security	89
Extensibility	89
Using Templates in Views	90
Template Loading	92
Template directories	93
render()	96
Subdirectories in get_template()	97
The <code>include</code> Template Tag	98
Template Inheritance	99
What's next?	104
Chapter 4: Models.....	104
The "Dumb" Way to Do Database Queries in Views	105
Configuring the Database	106
Your First App	107
Defining Models in Python	108
Your First Model	109
Installing the Model	112
Basic Data Access	118
Adding Model String Representations	119
Inserting and Updating Data	121
Selecting Objects	123
Filtering Data	124
Retrieving Single Objects	125
Ordering Data	126
Chaining Lookups	128

Slicing Data.....	128
Updating Multiple Objects in One Statement	129
Deleting Objects.....	131
What's Next?.....	132
Chapter 5: The Django Admin Site.....	132
Using the Admin Site.....	133
Start the development server.....	133
Enter the admin site.....	135
Adding Your Models to the Admin Site	139
How the Admin Site Works.....	142
Making Fields Optional	142
Making Date and Numeric Fields Optional.....	143
Customizing Field Labels.....	145
Custom ModelAdmin classes.....	146
Customizing change lists.....	146
Customizing edit forms	154
Users, Groups, and Permissions	158
When and Why to Use the Admin Interface – And When Not to	160
What's Next?.....	161
Chapter 6: Forms.....	161
Getting Data From the Request Object	162
Information About the URL.....	162
Other Information About the Request	163
Information About Submitted Data.....	165
A Simple Form-Handling Example.....	165
Improving Our Simple Form-Handling Example	170
Simple validation.....	173
Making a Contact Form.....	176
Your First Form Class.....	177

Tying Form Objects Into Views	180
Changing How Fields Are Rendered.....	183
Setting a Maximum Length	184
Setting Initial Values	184
Custom Validation Rules.....	185
Specifying labels.....	186
Customizing Form Design	187
What's Next?.....	189
Chapter 7: Advanced Views and URLconfs	190
URLconf Tricks.....	190
Streamlining Function Imports	190
Special-Casing URLs in Debug Mode.....	191
Named groups.....	192
The matching/grouping algorithm.....	195
What the URLconf searches against	195
Captured arguments are always strings	195
Specifying defaults for view arguments	197
Performance	197
Error handling	197
Including other URLconfs.....	198
Captured parameters.....	200
Passing extra options to view functions	201
Passing extra options to <code>include()</code>	202
Reverse resolution of URLs	203
Examples	204
Naming URL patterns	205
URL namespaces	206
Introduction	206
Reversing namespaced URLs	208

URL namespaces and included URLconfs	208
What's Next?.....	209
Chapter 8: Advanced Templates.....	210
Template Language Review	210
RequestContext and Context Processors	211
auth	215
debug	216
i18n	216
media	216
static.....	216
csrf.....	218
request.....	218
messages.....	218
Guidelines for Writing Your Own Context Processors.....	218
Automatic HTML Escaping	219
How to Turn it Off	220
For Individual Variables	220
For Template Blocks.....	221
Automatic Escaping of String Literals in Filter Arguments	222
Inside Template Loading.....	223
The <code>DIRS</code> option	223
Loader types.....	224
Extending the Template System	226
Code layout	226
Creating a Template Library.....	227
Custom template tags and filters	230
Writing Custom Template Filters	230
Registering custom filters	231
Template filters that expect strings.....	232

Filters and auto-escaping.....	232
Filters and time zones	236
Writing custom template tags	237
Simple tags.....	237
Inclusion tags	239
Assignment tags.....	242
Advanced custom template tags	242
A quick overview.....	243
Writing the compilation function	243
Writing the renderer.....	245
Auto-escaping considerations.....	246
Thread-safety considerations	247
Registering the tag	249
Passing template variables to the tag.....	250
Setting a variable in the context.....	251
Parsing until another block tag	254
Parsing until another block tag, and saving contents.....	255
What's Next	257
Chapter 9: Advanced Models.....	257
Related Objects.....	257
Accessing Foreign Key Values	258
Accessing Many-to-Many Values.....	259
Managers	260
Adding Extra Manager Methods.....	260
Modifying Initial Manager QuerySets	262
Model methods.....	265
Overriding predefined model methods.....	266
Executing Raw SQL Queries	268
Performing raw queries	268

Mapping query fields to model fields	270
Index lookups	271
Deferring model fields	271
Adding annotations.....	272
Passing parameters into <code>raw()</code>	272
Executing custom SQL directly.....	273
Connections and cursors.....	275
Adding extra Manager methods	277
What's Next?.....	278
Chapter 10: Generic Views	278
Generic views of objects	279
Making “friendly” template contexts	282
Adding extra context.....	282
Viewing subsets of objects.....	283
Dynamic filtering.....	285
Performing extra work.....	287
What's Next?.....	288
Chapter 11: User Authentication in Django.....	290
Overview	290
Using the Django authentication system.....	291
User objects	291
Creating users	291
Creating superusers	292
Changing passwords	292
Authenticating Users	293
Permissions and Authorization	294
Default permissions	294
Groups.....	295
Programmatically creating permissions	295

Permission caching	297
Authentication in Web requests.....	298
How to log a user in	298
How to log a user out.....	299
Limiting access to logged-in users	300
Authentication Views.....	305
login.....	305
logout	308
logout_then_login.....	309
password_change.....	309
password_change_done.....	310
password_reset	310
password_reset_done	312
password_reset_confirm.....	313
password_reset_complete	314
The redirect_to_login helper function.....	316
Built-in forms	316
AdminPasswordChangeForm.....	316
AuthenticationForm.....	316
PasswordChangeForm	318
PasswordResetForm	318
SetPasswordForm	318
UserChangeForm	318
UserCreationForm.....	319
Authentication data in templates	319
Managing users in the admin	320
Creating Users.....	320
Changing Passwords	322
Password management in Django	324

How Django stores passwords.....	324
Using bcrypt with Django.....	325
Increasing the work factor	326
Password upgrading.....	327
Manually managing a user's password.....	328
Customizing authentication in Django.....	328
Other authentication sources	329
Specifying authentication backends	329
Writing an authentication backend	330
Handling authorization in custom backends	332
Custom permissions.....	334
Extending the existing User model	335
Substituting a custom User model.....	336
Referencing the User model	337
Specifying a custom User model.....	338
Extending Django's default User.....	343
Custom users and the built-in auth forms	343
Custom users and <code>django.contrib.admin</code>	344
Custom users and permissions	345
Custom users and Proxy models.....	346
Custom users and signals.....	346
Custom users and testing/fixtures.....	346
A full example	348
Chapter 12 - testing in Django	348
Introducing automated testing.....	348
What are automated tests?.....	348
Why you need to create tests.....	348
Basic testing strategies	351
Writing tests.....	351

Running tests	352
The test database.....	354
Order in which tests are executed.....	355
Rollback emulation	355
Other test conditions.....	356
Understanding the test output.....	356
Speeding up the tests	357
Testing tools.....	359
The test client	359
Provided test case classes.....	371
Test cases features.....	378
Email services.....	392
Management Commands	394
Skipping tests	394
Using the Django test runner to test reusable applications.....	395
Using different testing frameworks	397
Defining a test runner	398
Testing utilities.....	401
Chapter 13: Deploying Django	403
Preparing Your Codebase for Production	403
Deployment checklist.....	403
Critical settings.....	404
SECRET_KEY.....	404
DEBUG	404
Environment-specific settings.....	405
ALLOWED_HOSTS	405
CACHES.....	405
DATABASES	405
EMAIL_BACKEND and related settings	406

STATIC_ROOT and STATIC_URL	406
MEDIA_ROOT and MEDIA_URL	406
HTTPS	407
CSRF_COOKIE_SECURE	407
SESSION_COOKIE_SECURE	407
Performance optimizations	407
CONN_MAX_AGE.....	407
TEMPLATES	407
Error reporting	408
LOGGING.....	408
ADMINS and MANAGERS	408
Customize the default error views.....	408
Using a virtualenv	408
Using Different Settings for Production.....	409
Deploying Django to a production server.....	412
Deploying Django with Apache and mod_wsgi	412
Basic configuration.....	412
Using mod_wsgi daemon mode	413
Serving files	414
Serving the admin files.....	415
If you get a UnicodeEncodeError	415
Serving static files in production.....	416
Serving the site and your static files from the same server	416
Serving static files from a dedicated server.....	417
Serving static files from a cloud service or CDN	418
Scaling	419
Running on a Single Server	419
Separating Out the Database Server	420
Running a Separate Media Server	421

Implementing Load Balancing and Redundancy.....	422
Going Big	423
Performance Tuning	425
There's No Such Thing As Too Much RAM.....	425
Turn Off Keep-Alive.....	425
Use memcached.....	426
Use memcached Often	426
Join the Conversation	426
License & Copyright	427
GNU Free Documentation License.....	427

FRONT MATTER

ACKNOWLEDGEMENTS

First and foremost, I would like to thank the original authors of the Django Book – Adrian Holovaty and Jacob Kaplan-Moss. They provided such a strong foundation that it has really been a delight writing this new edition.

Equal first in the shout out has to be the Django community. Vibrant and collaborative, the Django community is what really stood out to this cynical old businessman 6 years ago when I first discovered the “new kid on the web-framework block”. It’s your support that makes Django so great. Thank you.

ABOUT THE AUTHOR

Nigel George is a business systems developer specializing in the application of Open Source technologies to solve common business problems. He has a broad range of experience in software development – from writing database apps for small business, to developing the backend and UI for a distributed sensor network at the University of Newcastle, Australia.

Nigel also has over 15 years experience in technical writing for business. He has written several training manuals and hundreds of technical procedures for corporations and Australian government departments. He has been using Django since version 0.96.

Nigel lives in Newcastle, NSW, Australia.

INTRODUCTION

This year it will be 30 years since I plugged the 5.25” DOS 3.3 disk into my school’s very first Apple IIe computer and discovered BASIC.

In the intervening years I have written more lines of code than I could guess in about a dozen languages. I still write code every week – although the list of languages, and number of lines are somewhat diminished these days. Over the years I have seen plenty of horrible code and some really good stuff too. In my own work, I have written my fair share of good and bad.

Interestingly, not once in my entire career have I been employed as a programmer. I had my own IT business for five years, and have been in businesses large and small – mostly in R&D, technical and operations management – but never working solely as a programmer. What I have been is the guy that gets called up to **Get Stuff Done**.

Emphasized for good reason – business is all about Getting Stuff Done.

When everything has to work yesterday, religious wars over curly braces and pontification over which language is best for what application become trivialities.

Having read dozens and dozens of textbooks on all the various programming languages I have used, I know why you are here reading the introduction, so let's get right to the point.

WHY SHOULD YOU CARE ABOUT DJANGO?

While it is a given that Django is not the only web framework that will allow you to Get Stuff Done, I can confidently say one thing – if you want to write clean, intelligible code and build high performance, good looking modern websites quickly, then you will definitely benefit from working through this book.

I have deliberately not rattled off comparisons with other languages and frameworks because that is not the point – all languages and the frameworks and tools built on them have strengths and weaknesses. However, having worked with many of them over the years, I am totally convinced that Django stands way out in front for ease of use and ability to allow a programmer to produce robust, secure, and bug free code quickly.

Django is spectacularly good at getting out of your way when you just need to Get Something Done, but still exposes all the good stuff just under the surface when you want to dig down further. It is also built with Python, arguably the most intelligible and easy to learn programming language.

Of course these strengths do bring one challenge. Because both Python and Django hide an enormous amount of power and functionality just below the surface, it can be a bit confusing for beginners. This is where this book comes in. It's designed to quickly get you moving on your own Django projects, and then ultimately teach you everything you need to know to successfully design, develop, and deploy a site that you'll be proud of.

Adrian and Jacob wrote the original Django Book because they firmly believed that Django makes Web development better. I think Django's longevity and exponential growth in the years since the publication of the original Django Book is testament to this belief.

As per the original, this book is open source and all are welcome to improve it by either submitting comments and suggestions at the [website](#) or sending me an email to nigel at masteringdjango dot com.

I, like many, get a great deal of pleasure out of working with Django – it truly is as exciting, fun and useful as Adrian and Jacob had hoped it would be!

ABOUT THIS BOOK

This book is about Django, a Web development framework that saves you time and makes Web development a joy. Using Django, you can build and maintain high-quality Web applications with minimal fuss.

Mastering Django: Core is a completely revised and updated version of the Django Book – first published by Apress in 2007 as “*The Definitive Guide to Django: Web Development Done Right*” and then republished as ‘*The Django Book*’ by the original authors in 2009. The latter publication was released as an Open Source project under the Gnu Free Documentation License (GFDL).

Mastering Django: Core could be considered an unofficial 3rd edition of the Django Book, although I will leave it up to Jacob and the Django community to decide whether it deserves that honor. Personally, I just wanted to see it back out there because, like many Django programmers, the Django Book is where I got started.

To retain Adrian and Jacob’s original desire for the Django Book to be accessible as possible, the source code for *Mastering Django: Core* is freely available online at <https://github.com/big-nige/djangobook.com>.

The main goal of this book is to make you a Django expert. The focus is twofold. First, I explain in depth what Django does and how to build Web applications with it. Second, I discuss higher-level concepts where appropriate, answering the question “How can I apply these tools effectively in my own projects?” By reading this book, you’ll learn the skills needed to develop powerful Web sites quickly, with code that is clean and easy to maintain.

The secondary, but no less important, goal of this book is to provide a programmer’s manual that covers the current LTS version of Django. Django has matured to the point where it is seeing many commercial and business critical deployments. As such, this book is intended to provide the definitive up-to-date resource for commercial deployment of Django 1.8LTS. The electronic version of this book will be kept in sync with Django 1.8 right up until the end of extended support (2018).

How TO READ THIS Book

In writing *Mastering Django: Core*, I have tried to maintain a similar balance between readability and reference as the first book, however Django has grown considerably since 2007 and with increased power and flexibility, comes some additional complexity.

Django still has one of the shortest learning curves of all the web application frameworks, but there is still some solid work ahead of you if you want to become a Django expert.

This book retains the same “learn by example” philosophy as the original book, however some of the more complex sections (database configuration for example) have been moved to later chapters, so that you can first learn how Django works with a simple, out-of-the-box configuration and then build on your knowledge with more advanced topics later.

With that in mind, I recommend that you read Chapters 1 through 13 in order. They form the foundation of how to use Django; once you’ve read them, you’ll be able to build and deploy Django-powered Web sites. Specifically, Chapters 1 through 6 are the “core curriculum,” Chapters 7 through 12 cover more advanced Django usage, and Chapter 13 covers deployment. The remaining chapters, 14 through 23, focus on specific Django features and can be read in any order.

The appendices are for reference. They, along with the free documentation at <http://www.djangoproject.com/>, are probably what you’ll flip back to occasionally to recall syntax or find quick synopses of what certain parts of Django do.

REQUIRED PROGRAMMING KNOWLEDGE

Readers of this book should understand the basics of procedural and object-oriented programming: control structures (e.g., `if`, `while`, `for`), data structures (lists, hashes/dictionaries), variables, classes and objects.

Experience in Web development is, as you may expect, very helpful, but it’s not required to understand this book. Throughout the book, we try to promote best practices in Web development for readers who lack this experience.

REQUIRED PYTHON KNOWLEDGE

At its core, Django is simply a collection of libraries written in the Python programming language. To develop a site using Django, you write Python code that uses these libraries.

Learning Django, then, is a matter of learning how to program in Python and understanding how the Django libraries work.

If you have experience programming in Python, you should have no trouble diving in. By and large, the Django code doesn't perform a lot of "magic" (i.e., programming trickery whose implementation is difficult to explain or understand). For you, learning Django will be a matter of learning Django's conventions and APIs.

If you don't have experience programming in Python, you're in for a treat. It's easy to learn and a joy to use! Although this book doesn't include a full Python tutorial, it highlights Python features and functionality where appropriate, particularly when code doesn't immediately make sense. Still, we recommend you read the official Python tutorial, available online at <http://docs.python.org/tut/>. We also recommend Mark Pilgrim's free book *Dive Into Python*, available at <http://www.diveintopython.net/> and published in print by Apress.

REQUIRED DJANGO VERSION

This book covers Django 1.8 LTS.

This is the long term support version of Django, with full support from the Django developers until at least April 2018.

If you have an early version of Django, it is recommended that you upgrade to the latest version of Django 1.8LTS.

At the time of printing, the most current production version of Django 1.8LTS is 1.8.2.

If you have installed a later version of Django, please note that while Django's developers maintain backwards compatibility as much as possible, some backwards incompatible changes do get introduced occasionally. The changes in each release are always covered in the release notes, which you can find here:
<https://docs.djangoproject.com/en/dev/releases/>

GETTING HELP

One of the greatest benefits of Django is its kind and helpful user community. For help with any aspect of Django – from installation, to application design, to database design, to deployment – feel free to ask questions online.

- The django-users mailing list is where thousands of Django users hang out to ask and answer questions. Sign up for free at <http://www.djangoproject.com/r/django-users>.

- The Django IRC channel is where Django users hang out to chat and help each other in real time. Join the fun by logging on to #django on the Freenode IRC network.
-

INTRODUCTION TO DJANGO

Great open source software almost always comes about because one or more clever developers had a problem to solve and no viable or cost effective solution available. Django is no exception.

Adrian and Jacob have long since “retired” from the project, but the fundamentals of what drove them to create Django live on. It is this solid base of real-world experience that has made Django as successful as it is.

In recognition of their contribution, I think it best we let them introduce Django in their own words (edited and reformatted from the original book).

INTRODUCING DJANGO

BY ADRIAN HOLOVATY AND JACOB KAPLAN-MOSS – DECEMBER 2009

In the early days, Web developers wrote every page by hand. Updating a Web site meant editing HTML; a “redesign” involved redoing every single page, one at a time.

As Web sites grew and became more ambitious, it quickly became obvious that that approach was tedious, time-consuming, and ultimately untenable. A group of enterprising hackers at NCSA (the National Center for Supercomputing Applications, where Mosaic, the first graphical Web browser, was developed) solved this problem by letting the Web server spawn external programs that could dynamically generate HTML. They called this protocol the Common Gateway Interface, or CGI, and it changed the Web forever.

It’s hard now to imagine what a revelation CGI must have been: instead of treating HTML pages as simple files on disk, CGI allows you to think of your pages as resources generated dynamically on demand. The development of CGI ushered in the first generation of dynamic Web sites.

However, CGI has its problems: CGI scripts need to contain a lot of repetitive “boilerplate” code, they make code reuse difficult, and they can be difficult for first-time developers to write and understand.

PHP fixed many of these problems, and it took the world by storm – it's now by far the most popular tool used to create dynamic Web sites, and dozens of similar languages and environments (ASP, JSP, etc.) followed PHP's design closely. PHP's major innovation is its ease of use: PHP code is simply embedded into plain HTML; the learning curve for someone who already knows HTML is extremely shallow.

But PHP has its own problems; its very ease of use encourages sloppy, repetitive, ill-conceived code. Worse, PHP does little to protect programmers from security vulnerabilities, and thus many PHP developers found themselves learning about security only once it was too late.

These and similar frustrations led directly to the development of the current crop of “third-generation” Web development frameworks. With this new explosion of Web development comes yet another increase in ambition; Web developers are expected to do more and more every day.

Django was invented to meet these new ambitions.

DJANGO'S HISTORY

Django grew organically from real-world applications written by a Web development team in Lawrence, Kansas, USA. It was born in the fall of 2003, when the Web programmers at the *Lawrence Journal-World* newspaper, Adrian Holovaty and Simon Willison, began using Python to build applications.

The World Online team, responsible for the production and maintenance of several local news sites, thrived in a development environment dictated by journalism deadlines. For the sites – including LJWorld.com, Lawrence.com and KUsports.com – journalists (and management) demanded that features be added and entire applications be built on an intensely fast schedule, often with only days' or hours' notice. Thus, Simon and Adrian developed a time-saving Web development framework out of necessity – it was the only way they could build maintainable applications under the extreme deadlines.

In summer 2005, after having developed this framework to a point where it was efficiently powering most of World Online's sites, the team, which now included Jacob Kaplan-Moss, decided to release the framework as open source software. They released it in July 2005 and named it Django, after the jazz guitarist Django Reinhardt.

This history is relevant because it helps explain two key things. The first is Django's “sweet spot.” Because Django was born in a news environment, it offers several features (such as its admin site, covered in Chapter 6) that are particularly well suited for “content” sites – sites

like Amazon.com, craigslist.org, and washingtonpost.com that offer dynamic, database-driven information. Don't let that turn you off, though – although Django is particularly good for developing those sorts of sites, that doesn't preclude it from being an effective tool for building any sort of dynamic Web site. (There's a difference between being *particularly effective* at something and being *ineffective* at other things.)

The second matter to note is how Django's origins have shaped the culture of its open source community. Because Django was extracted from real-world code, rather than being an academic exercise or commercial product, it is acutely focused on solving Web development problems that Django's developers themselves have faced – and continue to face. As a result, Django itself is actively improved on an almost daily basis. The framework's maintainers have a vested interest in making sure Django saves developers time, produces applications that are easy to maintain and performs well under load.

Django lets you build deep, dynamic, interesting sites in an extremely short time. Django is designed to let you focus on the fun, interesting parts of your job while easing the pain of the repetitive bits. In doing so, it provides high-level abstractions of common Web development patterns, shortcuts for frequent programming tasks, and clear conventions on how to solve problems. At the same time, Django tries to stay out of your way, letting you work outside the scope of the framework as needed. We wrote this book because we firmly believe that Django makes Web development better. It's designed to quickly get you moving on your own Django projects, and then ultimately teach you everything you need to know to successfully design, develop, and deploy a site that you'll be proud of.

CHAPTER 1: GETTING STARTED

There are two very important things you need to do to get started with Django:

1. Install Django (obviously); and
2. Get a good understanding of the Model-View-Controller (MVC) design pattern.

The first, installing Django, is really simple and detailed in the first part of this chapter.

The second is just as important, especially if you are a new programmer or coming from using a programming language that does not clearly separate the data and logic behind your website from the way it is displayed.

Django's philosophy is based on *loose coupling*, which is the underlying philosophy of MVC. We will be discussing loose coupling and MVC in much more detail as we go along, but if you don't know much about MVC, then you best not skip the second half of this chapter, because understanding MVC will make understanding Django *so* much easier.

INSTALLING DJANGO

There are a few steps to installing Django, but they are all straight forward. In this chapter, we'll walk you through how to install the framework and its few dependencies.

This chapter assumes you're installing Django on a desktop/laptop machine and will be using the development server and SQLite to run all the example code in this book.

This is by far the easiest, and best way to setup Django when you are first starting out. If you do want to go to a more advanced installation of Django, your options are covered in Chapter 13 – [Deploying Django](#), Chapter 22 – [Complete Installation Guide](#) and Chapter 23 – [Advanced Database Management](#).

INSTALLING PYTHON

Django itself is written purely in Python, so the first step in installing the framework is to make sure you have Python installed.

PYTHON VERSIONS

Django version 1.8 LTS works with Python version 2.7, 3.3 and 3.4. For each version of Python, only the latest micro release (A.B.C) is supported.

Which Python version should I use?

Django 1.8 LTS works with the latest releases of both Python 2 and Python 3.

If you are just trialing Django, it doesn't really matter – either will work as well as the other.

[NOTE: All of the code samples in this book are written in Python 3](#)

If, however, you are planning on eventually deploying code to a live website, Python 3 should be your first choice. The [Python wiki](#) puts the reason behind this very succinctly:

Short version: Python 2.x is legacy, Python 3.x is the present and future of the language

Unless you have a very good reason to use Python 2 (e.g. legacy libraries), Python 3 is the way to go.

INSTALLATION

If you're on Linux or Mac OS X, you probably have Python already installed. Type `python` at a command prompt (or in Applications/Utilities/Terminal, in OS X). If you see something like this, then Python is installed:

```
Python 2.7.5 (default, June 27 2015, 13:20:20)
[GCC x.x.x] on xxx
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Otherwise, you'll need to download and install Python. It's fast and easy, and detailed instructions are available at <http://www.python.org/download/>

WARNING: You can see that, in the above example, Python interactive mode is running Python 2.7. This is a trap for inexperienced users. On Linux and Mac OS X machines, it is common for both Python 2 and Python 3 to be installed. If your system is like this, you need to type `python3` in front of all your commands, rather than `python` to run Django with Python 3.

INSTALLING DJANGO

Before you Start! Using `virtualenv`

Before you install Django, it is highly recommended you consider working within a Python virtual environment while you learn Django.

`virtualenv` is a Python tool that is used to create isolated Python environments. It is easy to set up and will ensure that any other applications on your computer that depend on Python don't get messed up if you accidentally overwrite something important. Setup and use of `virtualenv` is detailed in [Chapter 22](#).

At any given time, two distinct versions of Django are available to you: the latest official release and the bleeding-edge development version. The version you decide to install depends on your priorities. Do you want a stable and tested version of Django, or do you want a version containing the latest features, perhaps so you can contribute to Django itself, at the expense of stability?

This book only deals with installing an official release of Django (in this case Django 1.8 LTS). Instructions on installing the development version can be found in [Chapter 22](#). It is recommended that you stick with the latest official release, however it's important to know the development version exists as it is often mentioned in the Django documentation and by members of the community.

You've got two easy options to install Django:

1. Install a version of Django provided by your operating system distribution.
2. Install an official release from the Django Project website.

INSTALLING OS DISTRIBUTION VERSION

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, but not always. If your distro uses Django version 1.8 or later, you are OK – all the code in this book should work (check the release notes if you are using a later version though, some functions are deprecated over time.). If your distro uses a version of Django older than 1.8, you will need to install an official release from the Django Project website.

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.

INSTALLING AN OFFICIAL RELEASE

The recommended way to install Django is with *pip*. Using *pip* to install Django is an easy two step process:

1. Install [pip](#). The easiest way 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. If you're using Linux, Mac OS X or some other flavor of Unix, enter the command `sudo pip install Django` at the shell prompt. If you're using Windows, start a command shell with administrator privileges and run the command `pip install Django`. This will install Django in your Python installation's `site-packages` directory.

There are other ways to install Django that are not covered here. If you have previously experimented with Django without using *pip* you will also need to uninstall any old versions of Django. For more information, see the [Complete Installation Guide](#) in Chapter 22.

TESTING THE DJANGO INSTALLATION

For some post-installation positive feedback, take a moment to test whether the installation worked. In a command shell, change into your home directory and start the Python interactive interpreter by typing `python` (or `python3` if your system has two versions of Python installed). If the installation was successful, you should be able to import the module `django`:

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

1.8.2

NOTE: You may have another version of Django installed.

Interactive Interpreter Examples

The Python interactive interpreter is a command-line program that lets you write a Python program interactively. To start it, run the command `python` or `python3` at the command line.

Throughout this book, we feature example Python interactive interpreter sessions. You can recognize these examples by the triple greater-than signs (`>>>`), which designate the interpreter's prompt. If you're copying examples from this book, don't copy those greater-than signs.

Multiline statements in the interactive interpreter are padded with three dots (`...`). For example:

```
>>> print ("""This is a
... string that spans
... three lines.""")
This is a
string that spans
three lines.

>>> def my_function(value):
...     print (value)
>>> my_function('hello')
hello
```

Those three dots at the start of the additional lines are inserted by the Python shell – don't type them in. They are included to be faithful to the actual output of the interpreter. If you copy any examples from this book while following along, don't copy those dots.

SETTING UP A DATABASE

This step is not necessary in order to complete any of the examples in this book. Django comes with SQLite installed by default. SQLite requires no configuration on your part.

If you would like to work with a “large” database engine like PostgreSQL, MySQL, or Oracle, see [Chapter 23](#).

STARTING A PROJECT

Once you’ve installed Python, Django and (optionally) your database server/library, you can take the first step in developing a Django application by creating a *project*.

A project is a collection of settings for an instance of Django, including database configuration, Django-specific options and application-specific settings.

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, change 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.

Warning!

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`.

If you are following along and using the development server, this does not matter right now, but it is important that you remember this when you go to deploy your Django project to a production server.

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 Appendix F.
- 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. (Read [more about packages](#) in the official Python docs if you're a Python beginner.)
- `mysite/settings.py`: Settings/configuration for this Django project. [Appendix D](#) 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 Chapters [2](#) and [7](#).
- `mysite/wsgi.py`: An entry-point for WSGI-compatible web servers to serve your project. See [Chapter 13](#) for more details.

DJANGO SETTINGS

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

First step while you’re editing `mysite/settings.py`, is to set `TIME_ZONE` to your time zone.

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.
- `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 makes 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.

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...

0 errors found

June 27, 2015 - 15:50:53

Django version 1.8.2, using settings 'mysite.settings'

Starting development server at http://127.0.0.1:8000/

Quit the server with CONTROL-C.
```

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.**

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!

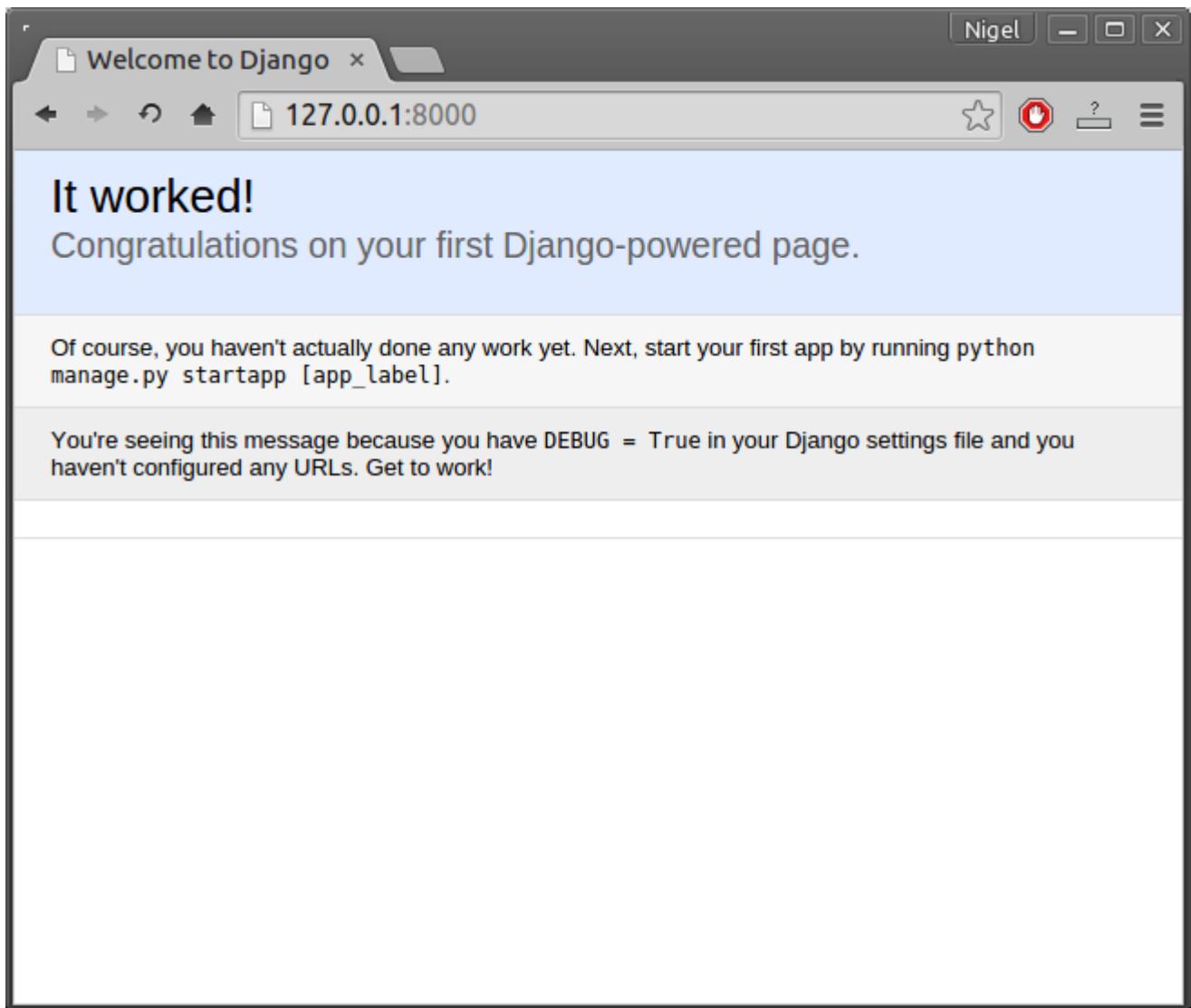


Figure 1-1. Django's welcome page

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.

THE MODEL-VIEW-CONTROLLER (MVC) DESIGN PATTERN

MVC has been around as a concept for a long time, but has seen exponential growth since the advent of the Internet because it is the best way to design client-server applications. All of the best web frameworks are built around the MVC concept. At the risk of starting a flame war, I contest that if you are not using MVC to design web apps, you are doing it wrong.

As concept, the MVC design pattern is really simple to understand:

- The **model(M)** is a model or representation of your data. It is not the actual data, but an interface to the data. The model allows you to pull data from your database without having to know the intricacies of the underlying database. The model usually also provides an *abstraction* layer with your database, so that you can use the same model with multiple databases.
- The **view(V)** is what you see. It is the presentation layer for your model. On your computer, the view is what you see in the browser for a Web app, or the UI for a desktop app. The view also provides an interface to collect user input.
- The **controller(C)** controls the flow of information between the model and the view. It uses programmed logic to decide what information is pulled from the database via the model and what information is passed to the view. It also gets information from the user via the view and implements business logic: either by changing the view, or modifying data through the model, or both.

Where it gets difficult is the vastly different interpretation of what actually happens at each layer – different frameworks implement the same functionality in different ways. One framework “guru” might say a certain function belongs in a view, while an other might vehemently defend the need for it to be in the controller.

You, as a budding programmer who Gets Stuff Done, do not have to care about this because in the end, it *doesn't matter*. As long as you understand how Django implements the MVC pattern, you are free to move on and get some real work done. Although, watching a flame war in a comment thread can be a highly amusing distraction...

Django follows the MVC pattern closely, however it does implement its own logic in the implementation. Because the “C” is handled by the framework itself and most of the

excitement in Django happens in models, templates and views, Django is often referred to as an *MTV framework*.

In the MTV development pattern:

- **M stands for “Model,”** the data access layer. This layer contains anything and everything about the data: how to access it, how to validate it, which behaviors it has, and the relationships between the data. We will be looking closely at [Django’s models](#) in Chapter 4.
- **T stands for “Template,”** the presentation layer. This layer contains presentation-related decisions: how something should be displayed on a Web page or other type of document. We will explore [Django’s templates](#) in Chapter 3.
- **V stands for “View,”** the business logic layer. This layer contains the logic that access the model and defers to the appropriate template(s). You can think of it as the bridge between models and templates. We will be checking out [Django’s views](#) in the next chapter.

This is probably the only unfortunate bit of naming in Django, because Django’s view is more like the controller in MVC, and MVC’s view is actually a Template in Django. It is a little confusing at first, but as a programmer getting a job done, you really won’t care for long. It is only a problem for those of us who have to teach it.

Oh, and to the flamers of course.

WHAT’S NEXT?

Now that you have everything installed and the development server running, you’re ready to move on to [Django views](#) and learning the basics of serving Web pages with Django.

CHAPTER 2: VIEWS AND URLCONFS

In the previous chapter, we explained how to set up a Django project and run the Django development server. In this chapter, you’ll learn the basics of creating dynamic Web pages with Django.

YOUR FIRST DJANGO-POWERED PAGE: HELLO WORLD

As our first goal, let's create a Web page that outputs that famous example message: "Hello world."

If you were publishing a simple "Hello world" Web page without a Web framework, you'd simply type "Hello world" into a text file, call it `hello.html`, and upload it to a directory on a Web server somewhere. Notice, in that process, you've specified two key pieces of information about that Web page: its contents (the string "Hello world") and its URL (`http://www.example.com/hello.html`, or maybe `http://www.example.com/files/hello.html` if you put it in a subdirectory).

With Django, you specify those same two things, but in a different way. The contents of the page are produced by a *view function*, and the URL is specified in a *URLconf*. First, let's write our "Hello world" view function.

YOUR FIRST VIEW

Within the `mysite` directory that `django-admin startproject` made in the last chapter, create an empty file called `views.py`. This Python module will contain our views for this chapter.

Our "Hello world" view is simple. Here's the entire function, plus import statements, which you should type into the `views.py` file:

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse("Hello world")
```

Let's step through this code one line at a time:

- First, we import the class `HttpResponse`, which lives in the `django.http` module. We need to import this class because it's used later in our code.
- Next, we define a function called `hello` – the view function.

Each view function takes at least one parameter, called `request` by convention. This is an object that contains information about the current Web request that has triggered this view, and it's an instance of the class `django.http.HttpRequest`. In this example, we don't do anything with `request`, but it must be the first parameter of the view nonetheless.

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 `hello` here, because that name clearly indicates the *gist* of the view, but it could just as well be named `hello_wonderful_beautiful_world`, or something equally revolting. The next section, "Your First URLconf", will shed light on how Django finds this function.

- The function is a simple one-liner: it merely returns an `HttpResponse` object that has been instantiated with the text "Hello world".

The main lesson here is this: a view is just a Python function that takes an `HttpRequest` as its first parameter and returns an instance of `HttpResponse`. In order for a Python function to be a Django view, it must do these two things. (There are exceptions, but we'll get to those later.)

YOUR FIRST URLCONF

If, at this point, you ran `python manage.py runserver` again, you'd still see the "Welcome to Django" message, with no trace of our "Hello world" view anywhere. That's because our `mysite` project doesn't yet know about the `hello` view; we need to tell Django explicitly that we're activating this view at a particular URL. (Continuing our previous analogy of publishing static HTML files, at this point we've created the HTML file but haven't uploaded it to a directory on the server yet.) To hook a view function to a particular URL with Django, we use a *URLconf*.

A URLconf is like a table of contents for your Django-powered Web site. Basically, it's a mapping between URLs and the view functions that should be called for those URLs. It's how you tell Django, "For this URL, call this code, and for that URL, call that code." For example, "When somebody visits the URL `/foo/`, call the view function `foo_view()`, which lives in the Python module `views.py`."

When you executed `django-admin startproject` in the previous chapter, the script created a URLconf for you automatically: the file `urls.py`. By default, it looks something like this:

```
"""mysite URL Configuration
```

The `'urlpatterns'` list routes URLs to views. For more information please see:

<https://docs.djangoproject.com/en/1.8/topics/http/urls/>

Examples:

Function views

1. Add an import: from my_app import views
2. Add a URL to urlpatterns: url(r'^\$', views.home, name='home')

Class-based views

1. Add an import: from other_app.views import Home
2. Add a URL to urlpatterns: url(r'^\$', Home.as_view(), name='home')

Including another URLconf

1. Add an import: from blog import urls as blog_urls
2. Add a URL to urlpatterns: url(r'^blog/', include(blog_urls))

"""

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

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

If we ignore the documentation comments at the top of the file, here's the essence of a URLconf:

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

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

Let's step through this code one line at a time:

- The first line imports two functions from the `django.conf.urls` module: `include` which allows you to include a full Python import path to another URLconf module,

and `url` which uses a regular expression to pattern match the URL in your browser to a module in your Django project.

- The second line calls the function `admin` from the `django.contrib` module. This function is called by the `include` function to load the URLs for the Django admin site.
- The third line is `urlpatterns` – a simple list of `url()` instances.

The main thing to note here is the variable `urlpatterns`, which Django expects to find in your URLconf module. This variable defines the mapping between URLs and the code that handles those URLs.

To add a URL and view to the URLconf, just add a mapping between a URL pattern and the view function. Here's how to hook in our `hello` view:

```
from django.conf.urls import include, url  
  
from django.contrib import admin  
  
from mysite.views import hello  
  
  
urlpatterns = [  
    url(r'^admin/', include(admin.site.urls)),  
    url(r'^hello/$', hello),  
]
```

We made two changes here:

- First, we imported the `hello` view from its module – `mysite/views.py`, which translates into `mysite.views` in Python import syntax. (This assumes `mysite/views.py` is on your Python path.)
- Next, we added the line `url(r'^hello/$', hello)`, to `urlpatterns`. This line is referred to as a *URLpattern*. The `url()` function tells Django how to handle the url that you are configuring. The first argument is a pattern-matching string (a regular expression; more on this in a bit) and the second argument is the view function to use for that pattern. `url()` can take other optional arguments as well, which we'll cover in more depth in [Chapter 7](#).

Note

One more important detail we've introduced here is that 'r' character in front of the regular expression string. This tells Python that the string is a "raw string" – its contents should not interpret backslashes. In normal Python strings, backslashes are used for escaping special characters – such as in the string '`\n`', which is a one-character string containing a newline. When you add the `r` to make it a raw string, Python does not apply its backslash escaping – so, `r'\n'` is a two-character string containing a literal backslash and a lowercase "n". There's a natural collision between Python's usage of backslashes and the backslashes that are found in regular expressions, so it's best practice to use raw strings any time you're defining a regular expression in Django.

In a nutshell, we just told Django that any request to the URL `/hello/` should be handled by the `hello` view function.

It's worth discussing the syntax of this URLpattern, as it may not be immediately obvious. Although we want to match the URL `/hello/`, the pattern looks a bit different than that. Here's why:

- Django removes the slash from the front of every incoming URL before it checks the URLpatterns. This means that our URLpattern doesn't include the leading slash in `/hello/`. (At first, this may seem unintuitive, but this requirement simplifies things – such as the inclusion of URLconfs within other URLconfs, which we'll cover in [Chapter 7](#).)
- The pattern includes a caret (^) and a dollar sign (\$). These are regular expression characters that have a special meaning: the caret means "require that the pattern matches the start of the string," and the dollar sign means "require that the pattern matches the end of the string."

This concept is best explained by example. If we had instead used the pattern `'^hello/'` (without a dollar sign at the end), then *any* URL starting with `/hello/` would match, such as `/hello/foo` and `/hello/bar`, not just `/hello/`. Similarly, if we had left off the initial caret character (i.e., `'hello/$'`), Django would match *any* URL that ends with `hello/`, such as `/foo/bar/hello/`. If we had simply used `hello/`, without a caret *or* dollar sign, then *any* URL containing `hello/` would match, such as

`/foo/hello/bar`. Thus, we use both the caret and dollar sign to ensure that only the URL `/hello/` matches – nothing more, nothing less.

Most of your URLpatterns will start with carets and end with dollar signs, but it's nice to have the flexibility to perform more sophisticated matches.

You may be wondering what happens if someone requests the URL `/hello` (that is, *without* a trailing slash). Because our URLpattern requires a trailing slash, that URL would *not* match. However, by default, any request to a URL that *doesn't* match a URLpattern and *doesn't* end with a slash will be redirected to the same URL with a trailing slash. (This is regulated by the `APPEND_SLASH` Django setting, which is covered in [Appendix D](#).)

The other thing to note about this URLconf is that we've passed the `hello` view function as an object without calling the function. This is a key feature of Python (and other dynamic languages): functions are first-class objects, which means you can pass them around just like any other variables. Cool stuff, eh?

To test our changes to the URLconf, start the Django development server, as you did in [Chapter 1](#), by running the command `python manage.py runserver`. (If you left it running, that's fine, too. The development server automatically detects changes to your Python code and reloads as necessary, so you don't have to restart the server between changes.) The server is running at the address `http://127.0.0.1:8000/`, so open up a Web browser and go to `http://127.0.0.1:8000/hello/`. You should see the text "Hello world" – the output of your Django view.

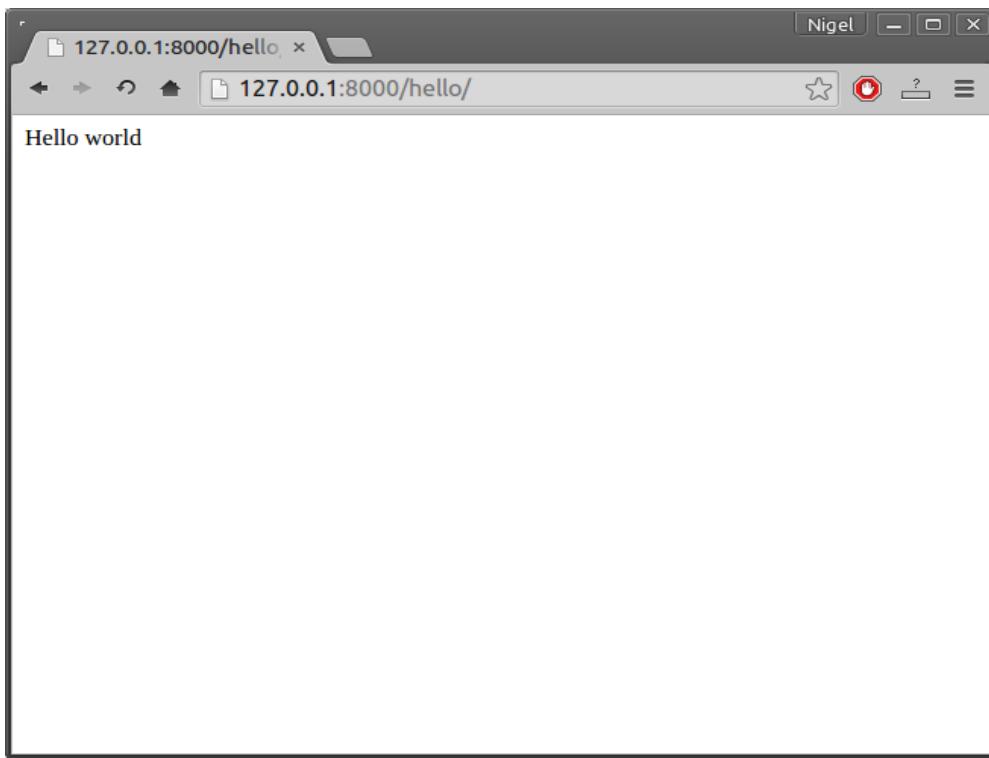


Figure 2-1. Hooray! Your first Django-powered Web page.

REGULAR EXPRESSIONS

Regular expressions (or *regexes*) are a compact way of specifying patterns in text. While Django URLconfs allow arbitrary regexes for powerful URL matching, you'll probably only use a few regex symbols in practice. Here's a selection of common symbols:

Symbol	Matches
. (dot)	Any single character
\d	Any single digit
[A-Z]	Any character between A and Z (uppercase)
[a-z]	Any character between a and z (lowercase)
[A-Za-z]	Any character between a and z (case-insensitive)
+	One or more of the previous expression (e.g., \d+ matches one or more digits)
[^/]+	One or more characters until (and not including) a forward slash
?	Zero or one of the previous expression (e.g., \d? matches zero or one digits)
*	Zero or more of the previous expression (e.g., \d* matches zero, one or more than one digit)
{1,3}	Between one and three (inclusive) of the previous expression (e.g., \d{1,3} matches one, two or three digits)

For more on regular expressions, see <https://docs.python.org/3.4/library/re.html>.

A QUICK NOTE ABOUT 404 ERRORS

At this point, our URLconf defines only a single URLpattern: the one that handles requests to the URL `/hello/`. What happens when you request a different URL?

To find out, try running the Django development server and visiting a page such as `http://127.0.0.1:8000/goodbye/` or `http://127.0.0.1:8000/hello/subdirectory/`, or even `http://127.0.0.1:8000/` (the site “root”). You should see a “Page not found” message (see Figure 2-2). Django displays this message because you requested a URL that’s not defined in your URLconf.

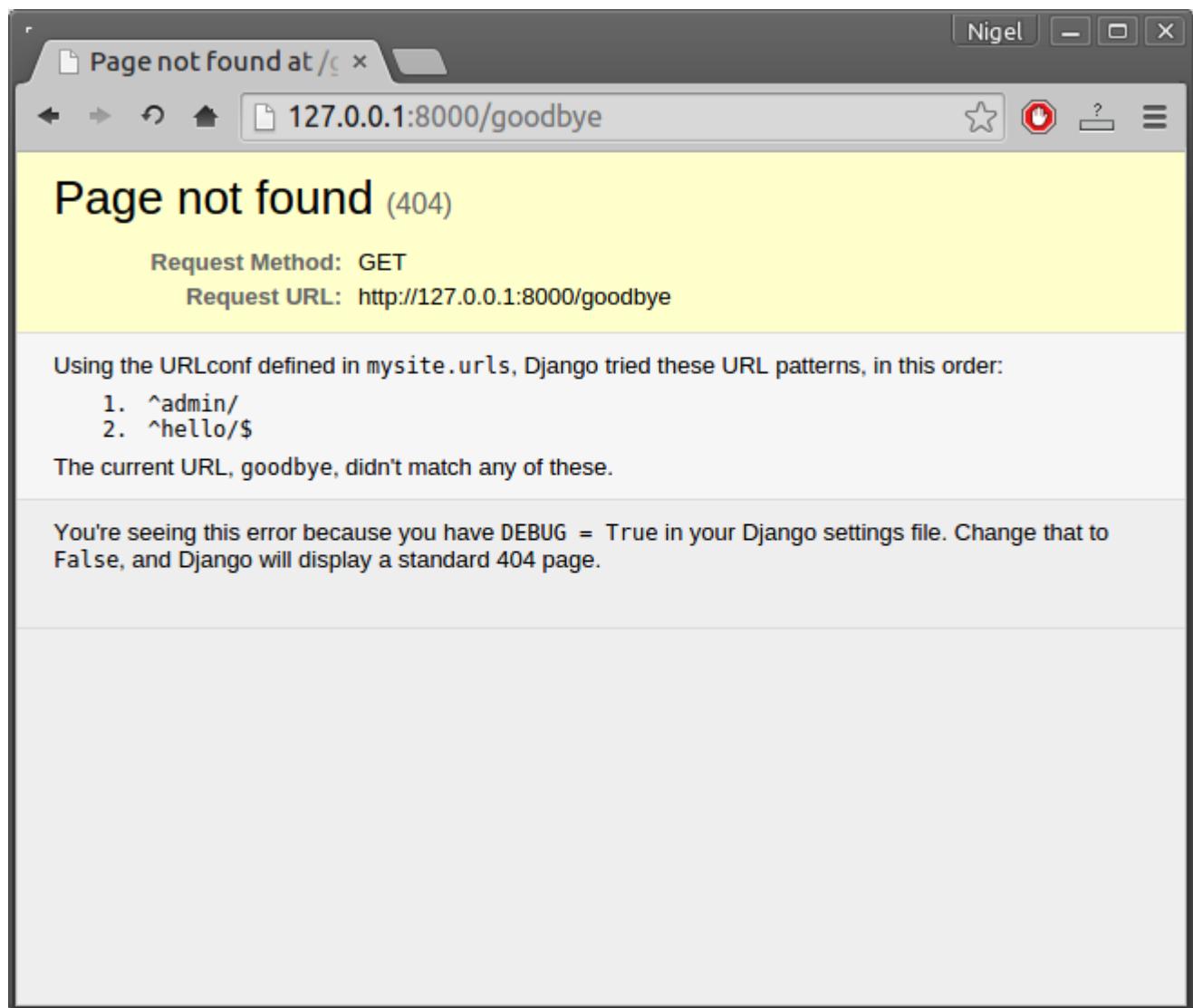


Figure 2-2. Django’s 404 page

The utility of this page goes beyond the basic 404 error message. It also tells you precisely which URLconf Django used and every pattern in that URLconf. From that information, you should be able to tell why the requested URL threw a 404.

Naturally, this is sensitive information intended only for you, the Web developer. If this were a production site deployed live on the Internet, you wouldn't want to expose that information to the public. For that reason, this "Page not found" page is only displayed if your Django project is in *debug mode*. We'll explain how to deactivate debug mode later. For now, just know that every Django project is in debug mode when you first create it, and if the project is not in debug mode, Django outputs a different 404 response.

A QUICK NOTE ABOUT THE SITE ROOT

As explained in the last section, you'll see a 404 error message if you view the site root – `http://127.0.0.1:8000/`. Django doesn't magically add anything to the site root; that URL is not special-cased in any way. It's up to you to assign it to a URLpattern, just like every other entry in your URLconf.

The URLpattern to match the site root is a bit unintuitive, though, so it's worth mentioning. When you're ready to implement a view for the site root, use the URLpattern '`'^$'`', which matches an empty string. For example:

```
from mysite.views import hello, my_homepage_view
```

```
urlpatterns = [
    url(r'^$', my_homepage_view),
    # ...
]
```

HOW DJANGO PROCESSES A REQUEST

Before continuing to our second view function, let's pause to learn a little more about how Django works. Specifically, when you view your "Hello world" message by visiting `http://127.0.0.1:8000/hello/` in your Web browser, what does Django do behind the scenes?

It all starts with the *settings file*. When you run `python manage.py runserver`, the script looks for a file called `settings.py` in the inner `mysite` directory. This file contains all sorts of

configuration for this particular Django project, all in uppercase: `TEMPLATE_DIRS`, `DATABASES`, etc. The most important setting is called `ROOT_URLCONF`. `ROOT_URLCONF` tells Django which Python module should be used as the URLconf for this Web site.

Remember when `django-admin startproject` created the files `settings.py` and `urls.py`? The autogenerated `settings.py` contains a `ROOT_URLCONF` setting that points to the autogenerated `urls.py`. Open the `settings.py` file and see for yourself; it should look like this:

```
ROOT_URLCONF = 'mysite.urls'
```

This corresponds to the file `mysite/urls.py`.

When a request comes in for a particular URL – say, a request for `/hello/` – Django loads the URLconf pointed to by the `ROOT_URLCONF` setting. Then it checks each of the URLpatterns in that URLconf, in order, comparing the requested URL with the patterns one at a time, until it finds one that matches. When it finds one that matches, it calls the view function associated with that pattern, passing it an `HttpRequest` object as the first parameter. (We'll cover the specifics of `HttpRequest` later.)

As we saw in our first view example, a view function must return an `HttpResponse`. Once it does this, Django does the rest, converting the Python object to a proper Web response with the appropriate HTTP headers and body (i.e., the content of the Web page).

In summary:

1. A request comes in to `/hello/`.
2. Django determines the root URLconf by looking at the `ROOT_URLCONF` setting.
3. Django looks at all of the URLpatterns in the URLconf for the first one that matches `/hello/`.
4. If it finds a match, it calls the associated view function.
5. The view function returns an `HttpResponse`.
6. Django converts the `HttpResponse` to the proper HTTP response, which results in a Web page.

You now know the basics of how to make Django-powered pages. It's quite simple, really – just write view functions and map them to URLs via URLconfs.

YOUR SECOND VIEW: DYNAMIC CONTENT

Our “Hello world” view was instructive in demonstrating the basics of how Django works, but it wasn’t an example of a *dynamic* Web page, because the content of the page are always the same. Every time you view `/hello/`, you’ll see the same thing; it might as well be a static HTML file.

For our second view, let’s create something more dynamic – a Web page that displays the current date and time. This is a nice, simple next step, because it doesn’t involve a database or any user input – just the output of your server’s internal clock. It’s only marginally more exciting than “Hello world,” but it’ll demonstrate a few new concepts.

This view needs to do two things: calculate the current date and time, and return an `HttpResponse` containing that value. If you have experience with Python, you know that Python includes a `datetime` module for calculating dates. Here’s how to use it:

```
>> import datetime  
  
>>> now = datetime.datetime.now()  
  
>>> now  
  
datetime.datetime(2015, 7, 15, 18, 12, 39, 2731)  
  
>>> print (now)  
  
2015-07-15 18:12:39.002731
```

That’s simple enough, and it has nothing to do with Django. It’s just Python code. (We want to emphasize that you should be aware of what code is “just Python” vs. code that is Django-specific. As you learn Django, we want you to be able to apply your knowledge to other Python projects that don’t necessarily use Django.)

To make a Django view that displays the current date and time, then, we just need to hook this `datetime.datetime.now()` statement into a view and return an `HttpResponse`. Here’s what the updated `views.py` looks like:

```
from django.http import HttpResponse  
  
import datetime  
  
  
def hello(request):  
    return HttpResponse("Hello world")
```

```

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponseRedirect(html)

```

Let's step through the changes we've made to `views.py` to accommodate the `current_datetime` view.

- We've added an `import datetime` to the top of the module, so we can calculate dates.
- The new `current_datetime` function calculates the current date and time, as a `datetime.datetime` object, and stores that as the local variable `now`.
- The second line of code within the view constructs an HTML response using Python's "format-string" capability. The `%s` within the string is a placeholder, and the percent sign after the string means "Replace the `%s` in the preceding string with the value of the variable `now`." The `now` variable is technically a `datetime.datetime` object, not a string, but the `%s` format character converts it to its string representation, which is something like "2008-12-13 14:09:39.002731". This will result in an HTML string such as "<html><body>It is now 2008-12-13 14:09:39.002731.</body></html>".
- Finally, the view returns an `HttpResponse` object that contains the generated response – just as we did in `hello`.

After adding that to `views.py`, add the URLpattern to `urls.py` to tell Django which URL should handle this view. Something like `/time/` would make sense:

```

from django.conf.urls import include, url
from django.contrib import admin
from mysite.views import hello, current_datetime

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

```

```
        url(r'^time/$', current_datetime),  
    ]
```

We've made two changes here. First, we imported the `current_datetime` function at the top. Second, and more importantly, we added a URLpattern mapping the URL `/time/` to that new view. Getting the hang of this?

With the view written and URLconf updated, fire up the `runserver` and visit `http://127.0.0.1:8000/time/` in your browser. You should see the current date and time. If you don't see your local time, it is because you did not change the default timezone in your `settings.py` (see [Chapter 1](#)).

URLCONFS AND LOOSE COUPLING

Now's a good time to highlight a key philosophy behind URLconfs and behind Django in general: the principle of *loose coupling*. Simply put, loose coupling is a software-development approach that values the importance of making pieces interchangeable. If two pieces of code are loosely coupled, then changes made to one of the pieces will have little or no effect on the other.

Django's URLconfs are a good example of this principle in practice. In a Django web application, the URL definitions and the view functions they call are loosely coupled; that is, the decision of what the URL should be for a given function, and the implementation of the function itself, reside in two separate places. This lets you switch out one piece without affecting the other.

For example, consider our `current_datetime` view. If we wanted to change the URL for the application – say, to move it from `/time/` to `/current-time/` – we could make a quick change to the URLconf, without having to worry about the view itself. Similarly, if we wanted to change the view function – altering its logic somehow – we could do that without affecting the URL to which the function is bound.

Furthermore, if we wanted to expose the current-date functionality at *several* URLs, we could easily take care of that by editing the URLconf, without having to touch the view code. In this example, our `current_datetime` is available at two URLs. It's a contrived example, but this technique can come in handy:

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

```
        url(r'^hello/$', hello),  
  
        url(r'^time/$', current_datetime),  
  
        url(r'^another-time-page/$', current_datetime),  
    ]
```

URLconfs and views are loose coupling in action. I'll continue to point out examples of this important philosophy throughout this book.

YOUR THIRD VIEW: DYNAMIC URLs

In our `current_datetime` view, the contents of the page – the current date/time – were dynamic, but the URL (`/time/`) was static. In most dynamic Web applications, though, a URL contains parameters that influence the output of the page. For example, an online bookstore might give each book its own URL, like `/books/243/` and `/books/81196/`.

Let's create a third view that displays the current date and time offset by a certain number of hours. The goal is to craft a site in such a way that the page `/time/plus/1/` displays the date/time one hour into the future, the page `/time/plus/2/` displays the date/time two hours into the future, the page `/time/plus/3/` displays the date/time three hours into the future, and so on.

A novice might think to code a separate view function for each hour offset, which might result in a URLconf like this:

```
urlpatterns = [  
  
    url(r'^time/$', current_datetime),  
  
    url(r'^time/plus/1/$', one_hour_ahead),  
  
    url(r'^time/plus/2/$', two_hours_ahead),  
  
    url(r'^time/plus/3/$', three_hours_ahead),  
  
    url(r'^time/plus/4/$', four_hours_ahead),  
  
]
```

Clearly, this line of thought is flawed. Not only would this result in redundant view functions, but also the application is fundamentally limited to supporting only the predefined hour ranges – one, two, three or four hours. If we decided to create a page that displayed the time *five* hours into the future, we'd have to create a separate view and URLconf line for that, furthering the duplication. We need to do some abstraction here.

A Word About Pretty URLs

If you’re experienced in another Web development platform, you may be thinking, “Hey, let’s use a query string parameter!” – something like `/time/plus?hours=3`, in which the hours would be designated by the `hours` parameter in the URL’s query string (the part after the ‘?’).

You *can* do that with Django (and we’ll tell you how in [Chapter 7](#)), but one of Django’s core philosophies is that URLs should be beautiful. The URL `/time/plus/3/` is far cleaner, simpler, more readable, easier to recite to somebody aloud and . . . just plain prettier than its query string counterpart. Pretty URLs are a characteristic of a quality Web application.

Django’s URLconf system encourages pretty URLs by making it easier to use pretty URLs than *not* to.

How, then do we design our application to handle arbitrary hour offsets? The key is to use *wildcard URLpatterns*. As we mentioned previously, a URLpattern is a regular expression; hence, we can use the regular expression pattern `\d+` to match one or more digits:

```
urlpatterns = [
    # ...
    url(r'^time/plus/\d+/$', hours_ahead),
    # ...
]
```

(We’re using the `# ...` to imply there might be other URLpatterns that we trimmed from this example.)

This new URLpattern will match any URL such as `/time/plus/2/`, `/time/plus/25/`, or even `/time/plus/1000000000000000000/`. Come to think of it, let’s limit it so that the maximum allowed offset is something reasonable. In this example, we will set a maximum 99 hours by only allowing either one- or two-digit numbers – and in regular expression syntax, that translates into `\d{1,2}`:

```
url(r'^time/plus/\d{1,2}/$', hours_ahead),
```

Now that we've designated a wildcard for the URL, we need a way of passing that wildcard data to the view function, so that we can use a single view function for any arbitrary hour offset. We do this by placing parentheses around the data in the URLpattern that we want to save. In the case of our example, we want to save whatever number was entered in the URL, so let's put parentheses around the `\d{1,2}`, like this:

```
url(r'^time/plus/(\d{1,2})/$', hours_ahead),
```

If you're familiar with regular expressions, you'll be right at home here; we're using parentheses to *capture* data from the matched text.

The final URLconf, including our previous two views, looks like this:

```
from django.conf.urls import include, url
from django.contrib import admin
from mysite.views import hello, current_datetime, hours_ahead

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^hello/$', hello),
    url(r'^time/$', current_datetime),
    url(r'^time/plus/(\d{1,2})/$', hours_ahead),
]
```

With that taken care of, let's write the `hours_ahead` view.

`hours_ahead` is very similar to the `current_datetime` view we wrote earlier, with a key difference: it takes an extra argument, the number of hours of offset. Here's the view code:

```
from django.http import Http404, HttpResponseRedirect
import datetime

def hours_ahead(request, offset):
    try:
        offset = int(offset)
    
```

```

except ValueError:

    raise Http404()

dt = datetime.datetime.now() + datetime.timedelta(hours=offset)

html = "<html><body>In %s hour(s), it will be %s.</body></html>" %
(offset, dt)

return HttpResponse(html)

```

Let's step through this code one line at a time:

- The view function, `hours_ahead`, takes two parameters: `request` and `offset`.
 - `request` is an `HttpRequest` object, just as in `hello` and `current_datetime`. We'll say it again: each view always takes an `HttpRequest` object as its first parameter.
 - `offset` is the string captured by the parentheses in the URLpattern. For example, if the requested URL were `/time/plus/3/`, then `offset` would be the string '`3`'. If the requested URL were `/time/plus/21/`, then `offset` would be the string '`21`'. Note that captured values will always be *Unicode objects*, not integers, even if the string is composed of only digits, such as '`21`'.

We decided to call the variable `offset`, but you can call it whatever you'd like, as long as it's a valid Python identifier. The variable name doesn't matter; all that matters is that it's the second argument to the function, after `request`. (It's also possible to use keyword, rather than positional, arguments in an URLconf. We cover that in [Chapter 7](#).)

- The first thing we do within the function is call `int()` on `offset`. This converts the Unicode string value to an integer.

Note that Python will raise a `ValueError` exception if you call `int()` on a value that cannot be converted to an integer, such as the string '`foo`'. In this example, if we encounter the `ValueError`, we raise the exception `django.http.Http404`, which, as you can imagine, results in a 404 "Page not found" error.

Astute readers will wonder: how could we ever reach the `ValueError` case, anyway, given that the regular expression in our URLpattern – `(\d{1,2})` – captures only digits, and therefore `offset` will only ever be a string composed of digits? The answer is, we won't, because the URLpattern provides a modest but useful level of input

validation, but we still check for the `ValueError` in case this view function ever gets called in some other way. It's good practice to implement view functions such that they don't make any assumptions about their parameters. Loose coupling, remember?

- In the next line of the function, we calculate the current date/time and add the appropriate number of hours. We've already seen `datetime.datetime.now()` from the `current_datetime` view; the new concept here is that you can perform date/time arithmetic by creating a `datetime.timedelta` object and adding to a `datetime.datetime` object. Our result is stored in the variable `dt`.

This line also shows why we called `int()` on `offset` – the `datetime.timedelta` function requires the `hours` parameter to be an integer.

- Next, we construct the HTML output of this view function, just as we did in `current_datetime`. A small difference in this line from the previous line is that it uses Python's format-string capability with *two* values, not just one. Hence, there are two `%s` symbols in the string and a tuple of values to insert: `(offset, dt)`.
- Finally, we return an `HttpResponse` of the HTML.

With that view function and URLconf written, start the Django development server (if it's not already running), and visit `http://127.0.0.1:8000/time/plus/3/` to verify it works. Then try

`http://127.0.0.1:8000/time/plus/5/` Then

`http://127.0.0.1:8000/time/plus/24/` Finally, visit

`http://127.0.0.1:8000/time/plus/100/` to verify that the pattern in your URLconf only accepts one- or two-digit numbers; Django should display a "Page not found" error in this case, just as we saw in the section "A Quick Note About 404 Errors" earlier. The URL `http://127.0.0.1:8000/time/plus/` (with *no* hour designation) should also throw a 404.

Coding Order

In this example, we wrote the URLpattern first and the view second, but in the previous examples, we wrote the view first, then the URLpattern. Which technique is better?

Well, every developer is different.

If you're a big-picture type of person, it may make the most sense to you to write all of the URLpatterns for your application at the same time, at the start of your project, and

then code up the views. This has the advantage of giving you a clear to-do list, and it essentially defines the parameter requirements for the view functions you'll need to write.

If you're more of a bottom-up developer, you might prefer to write the views first, and then anchor them to URLs afterward. That's OK, too.

In the end, it comes down to which technique fits your brain the best. Both approaches are valid.

DJANGO'S PRETTY ERROR PAGES

Take a moment to admire the fine Web application we've made so far . . . now let's break it! Let's deliberately introduce a Python error into our `views.py` file by commenting out the `offset = int(offset)` lines in the `hours_ahead` view:

```
def hours_ahead(request, offset):
    # try:
    #     offset = int(offset)
    # except ValueError:
    #     raise Http404()

    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)

    html = "<html><body>In %s hour(s), it will be %s.</body></html>" % (offset, dt)

    return HttpResponse(html)
```

Load up the development server and navigate to `/time/plus/3/`. You'll see an error page with a significant amount of information, including a `TypeError` message displayed at the very top: "unsupported type for timedelta hours component: str".

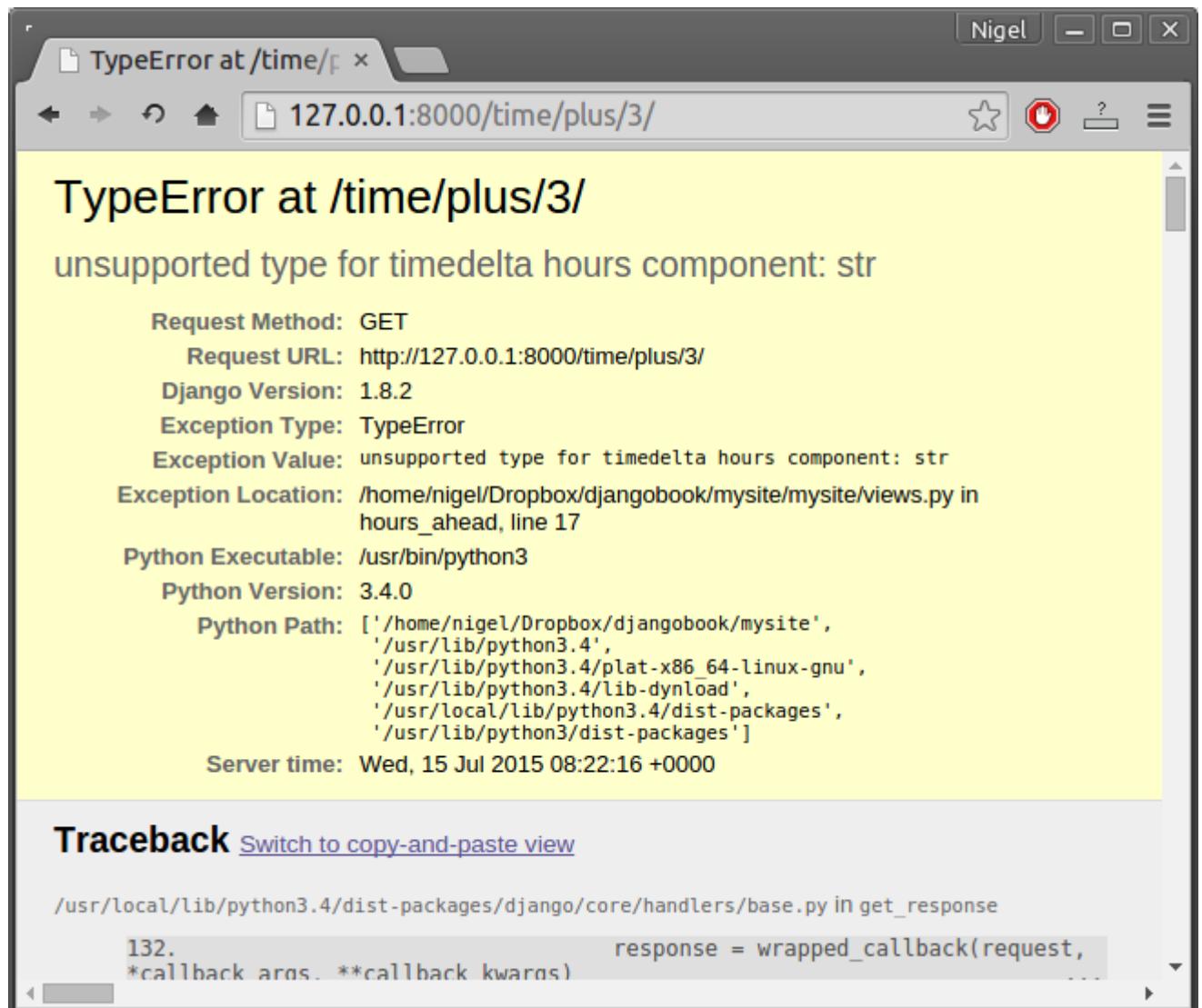


Figure 2-3. Django's error page

What happened? Well, the `datetime.timedelta` function expects the `hours` parameter to be an integer, and we commented out the bit of code that converted `offset` to an integer. That caused `datetime.timedelta` to raise the `TypeError`. It's the typical kind of small bug that every programmer runs into at some point.

The point of this example was to demonstrate Django's error pages. Take some time to explore the error page and get to know the various bits of information it gives you.

Here are some things to notice:

- At the top of the page, you get the key information about the exception: the type of exception, any parameters to the exception (the "unsupported type" message in this case), the file in which the exception was raised, and the offending line number.
- Under the key exception information, the page displays the full Python traceback for this exception. This is similar to the standard traceback you get in Python's command-line interpreter, except it's more interactive. For each level ("frame") in the stack, Django displays the name of the file, the function/method name, the line number, and the source code of that line.

Click the line of source code (in dark gray), and you'll see several lines from before and after the erroneous line, to give you context.

Click "Local vars" under any frame in the stack to view a table of all local variables and their values, in that frame, at the exact point in the code at which the exception was raised. This debugging information can be a great help.

- Note the "Switch to copy-and-paste view" text under the "Traceback" header. Click those words, and the traceback will switch to a alternate version that can be easily copied and pasted. Use this when you want to share your exception traceback with others to get technical support – such as the kind folks in the Django IRC chat room or on the Django users mailing list.

Underneath, the "Share this traceback on a public Web site" button will do this work for you in just one click. Click it to post the traceback to <http://www.dpaste.com/>, where you'll get a distinct URL that you can share with other people.

- Next, the "Request information" section includes a wealth of information about the incoming Web request that spawned the error: GET and POST information, cookie values, and meta information, such as CGI headers. [Appendix G](#) has a complete reference of all the information a request object contains.

Below the "Request information" section, the "Settings" section lists all of the settings for this particular Django installation. (We've already mentioned `ROOT_URLCONF`, and we'll show you various Django settings throughout the book. All the available settings are covered in detail in [Appendix D](#).)

The Django error page is capable of displaying more information in certain special cases, such as the case of template syntax errors. We'll get to those later, when we discuss the Django

template system. For now, uncomment the `offset = int(offset)` lines to get the view function working properly again.

Are you the type of programmer who likes to debug with the help of carefully placed `print` statements? You can use the Django error page to do so – just without the `print` statements. At any point in your view, temporarily insert an `assert False` to trigger the error page. Then, you can view the local variables and state of the program. Here's an example, using the `hours_ahead` view:

```
def hours_ahead(request, offset):
    try:
        offset = int(offset)
    except ValueError:
        raise Http404()
    dt = datetime.datetime.now() + datetime.timedelta(hours=offset)
    assert False
    html = "<html><body>In %s hour(s), it will be %s.</body></html>" % (offset, dt)
    return HttpResponse(html)
```

Finally, it's obvious that much of this information is sensitive – it exposes the innards of your Python code and Django configuration – and it would be foolish to show this information on the public Internet. A malicious person could use it to attempt to reverse-engineer your Web application and do nasty things. For that reason, the Django error page is only displayed when your Django project is in debug mode. We'll explain how to deactivate [debug mode in Chapter 13](#). For now, just know that every Django project is in debug mode automatically when you start it. (Sound familiar? The “Page not found” errors, described earlier in this chapter, work the same way.)

WHAT'S NEXT?

So far, we've been writing our view functions with HTML hard-coded directly in the Python code. We've done that to keep things simple while we demonstrated core concepts, but in the real world, this is nearly always a bad idea.

Django ships with a simple yet powerful [template engine](#) that allows you to separate the design of the page from the underlying code. We'll dive into Django's template engine in the next chapter.

CHAPTER 3: TEMPLATES

In the previous chapter, you may have noticed something peculiar in how we returned the text in our example views. Namely, the HTML was hard-coded directly in our Python code, like this:

```
def current_datetime(request):  
    now = datetime.datetime.now()  
  
    html = "It is now %s." % now  
  
    return HttpResponse(html)
```

Although this technique was convenient for the purpose of explaining how views work, it's not a good idea to hard-code HTML directly in your views. Here's why:

- Any change to the design of the page requires a change to the Python code. The design of a site tends to change far more frequently than the underlying Python code, so it would be convenient if the design could change without needing to modify the Python code.
- This is only a very simple example. A common webpage template has hundreds of lines of HTML and scripts. Untangling and troubleshooting program code from this mess is a nightmare (*cough-PHP-cough*).
- Writing Python code and designing HTML are two different disciplines, and most professional Web development environments split these responsibilities between separate people (or even separate departments). Designers and HTML/CSS coders shouldn't be required to edit Python code to get their job done.
- It's most efficient if programmers can work on Python code and designers can work on templates at the same time, rather than one person waiting for the other to finish editing a single file that contains both Python and HTML.

For these reasons, it's much cleaner and more maintainable to separate the design of the page from the Python code itself. We can do this with Django's *template system*, which we discuss in this chapter.

TEMPLATE SYSTEM BASICS

A Django template is a string of text that is intended to separate the presentation of a document from its data. A template defines placeholders and various bits of basic logic (template tags) that regulate how the document should be displayed. Usually, templates are used for producing HTML, but Django templates are equally capable of generating any text-based format.

Philosophy behind Django templates

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.

Let's start with a simple example template. This Django template describes an HTML page that thanks a person for placing an order with a company. Think of it as a form letter:

ORDERING NOTICE

Dear {{ person_name }},

Thanks for placing an order from {{ company }}. It's scheduled to ship on {{ ship_date|date:"F j, Y" }}.

Here are the items you've ordered:

```
{% for item in item_list %}  
• {{ item }}  
{% endfor %} {% if ordered_warranty %}
```

Your warranty information will be included in the packaging.

```
{% else %}
```

You didn't order a warranty, so you're on your own when the products inevitably stop working.

```
{% endif %}
```

Sincerely,

```
{{ company }}
```

This template is basic HTML with some variables and template tags thrown in. Let's step through it:

- Any text surrounded by a pair of braces (e.g., `{{ person_name }}`) is a *variable*. This means “insert the value of the variable with the given name.” (How do we specify the values of the variables? We’ll get to that in a moment.)
- Any text that’s surrounded by curly braces and percent signs (e.g., `{% if ordered_warranty %}`) is a *template tag*. The definition of a tag is quite broad: a tag just tells the template system to “do something.”

This example template contains a `for` tag (`{% for item in item_list %}`) and an `if` tag (`{% if ordered_warranty %}`).

A `for` tag works very much like a `for` statement in Python, letting you loop over each item in a sequence. An `if` tag, as you may expect, acts as a logical “if” statement. In this particular case, the tag checks whether the value of the `ordered_warranty` variable evaluates to `True`. If it does, the template system will display everything between the `{% if ordered_warranty %}` and `{% else %}`. If not, the template system will display everything between `{% else %}` and `{% endif %}`. Note that the `{% else %}` is optional.

- Finally, the second paragraph of this template contains an example of a *filter*, which is the most convenient way to alter the formatting of a variable. In this example, `{{ ship_date|date:"F j, Y" }}`, we’re passing the `ship_date` variable to the `date` filter, giving the `date` filter the argument `"F j, Y"`. The `date` filter formats dates in a given format, as specified by that argument. Filters are attached using a pipe character (`|`), as a reference to Unix pipes.

Each Django template has access to several built-in tags and filters, many of which are discussed in the sections that follow. Appendix E contains the full list of tags and filters, and it's a good idea to familiarize yourself with that list so you know what's possible. It's also possible to create your own filters and tags; we'll cover that in [Chapter 8](#).

USING THE TEMPLATE SYSTEM

A Django project can be configured with one or several template engines (or even zero if you don't use templates). Django ships with a built-in backend for its own template system – the *Django Template language (DTL)*. Django 1.8 also includes support for the popular alternative [Jinja2](#).

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.

All of the examples in this chapter will use the DTL. For more advanced template topics, including configuring third-party template engines see [Chapter 8](#).

Before we go about implementing Django templates in your view, let's first dig inside the DTL a little so you can see how it works.

Here is the most basic way you can use Django's template system in Python code:

1. Create a `Template` object by providing the raw template code as a string.
2. Call the `render()` method of the `Template` object with a given set of variables (the *context*). This returns a fully rendered template as a string, with all of the variables and template tags evaluated according to the context.

In code, here's what that looks like:

```
>>> from django import template  
  
>>> t = template.Template('My name is {{ name }}.')  
  
>>> c = template.Context({'name': 'Nige'})  
  
>>> print (t.render(c))  
  
My name is Nige.  
  
>>> c = template.Context({'name': 'Barry'})  
  
>>> print (t.render(c))  
  
My name is Barry.
```

The following sections describe each step in much more detail.

CREATING TEMPLATE OBJECTS

The easiest way to create a `Template` object is to instantiate it directly. The `Template` class lives in the `django.template` module, and the constructor takes one argument, the raw template code. Let's dip into the Python interactive interpreter to see how this works in code.

From the `mysite` project directory created by `django-admin startproject` (as covered in [Chapter 1](#)), type `python manage.py shell` to start the interactive interpreter.

A special Python prompt

If you've used Python before, you may be wondering why we're running `python manage.py shell` instead of just `python` (or `python3`). Both commands will start the interactive interpreter, but the `manage.py shell` command has one key difference: before starting the interpreter, it tells Django which settings file to use. Many parts of Django, including the template system, rely on your settings, and you won't be able to use them unless the framework knows which settings to use.

If you're curious, here's how it works behind the scenes. Django looks for an environment variable called `DJANGO_SETTINGS_MODULE`, which should be set to the import path of your `settings.py`. For example, `DJANGO_SETTINGS_MODULE` might be set to `'mysite.settings'`, assuming `mysite` is on your Python path.

When you run `python manage.py shell`, the command takes care of setting `DJANGO_SETTINGS_MODULE` for you. You will need to use `python manage.py shell` in these examples, or Django will throw an exception.

Let's go through some template system basics:

```
>>> from django.template import Template  
>>> t = Template('My name is {{ name }}.')  
>>> print (t)
```

If you're following along interactively, you'll see something like this:

That `0x7f12e132a860` will be different every time, and it isn't relevant; it's a Python thing (the Python "identity" of the `Template` object, if you must know).

When you create a `Template` object, the template system compiles the raw template code into an internal, optimized form, ready for rendering. But if your template code includes any syntax errors, the call to `Template()` will cause a `TemplateSyntaxError` exception:

```
>>> from django.template import Template  
  
>>> t = Template('{% notatag %}')  
  
Traceback (most recent call last):  
  
  File "", line 1, in ?  
  
    ...  
  
django.template.base.TemplateSyntaxError: Invalid block tag: 'notatag'
```

The term “block tag” here refers to `{% notatag %}`. “Block tag” and “template tag” are synonymous.

The system raises a `TemplateSyntaxError` exception for any of the following cases:

- Invalid tags
- Invalid arguments to valid tags
- Invalid filters
- Invalid arguments to valid filters
- Invalid template syntax
- Unclosed tags (for tags that require closing tags)

RENDERING A TEMPLATE

Once you have a `Template` object, you can pass it data by giving it a `context`. A context is simply a set of template variable names and their associated values. A template uses this to populate its variables and evaluate its tags.

A context is represented in Django by the `Context` class, which lives in the `django.template` module. Its constructor takes one optional argument: a dictionary mapping variable names to variable values. Call the `Template` object’s `render()` method with the context to “fill” the template:

```
>>> from django.template import Context, Template
```

```
>>> t = Template('My name is {{ name }}.')
>>> c = Context({'name': 'Stephane'})
>>> t.render(c)
'My name is Stephane.'
```

DICTIONARIES AND CONTEXTS

A Python dictionary is a mapping between known keys and variable values. A `Context` is similar to a dictionary, but a `Context` provides additional functionality, as covered in [Chapter 8](#).

Variable names must begin with a letter (A-Z or a-z) and may contain more letters, digits, underscores, and dots. (Dots are a special case we'll get to in a moment.) Variable names are case sensitive.

Here's an example of template compilation and rendering, using a template similar to the example in the beginning of this chapter:

```
>>> from django.template import Template, Context
>>> raw_template = """
Dear {{ person_name }},
...
...
Thanks for placing an order from {{ company }}. It's scheduled to
... ship on {{ ship_date|date:"F j, Y" }}.
...
...
... {%- if ordered_warranty %}
...
Your warranty information will be included in the packaging.

... {%- else %}
...
You didn't order a warranty, so you're on your own when
...
... the products inevitably stop working.
```

```
... { % endif %}

...
...

Sincerely,
{{ company }}

"""

>>> t = Template(raw_template)

>>> import datetime

>>> c = Context({'person_name': 'John Smith',
...     'company': 'Outdoor Equipment',
...     'ship_date': datetime.date(2015, 7, 2),
...     'ordered_warranty': False})

>>> t.render(c)

"

Dear John Smith,

\n\n

Thanks for placing an order from Outdoor
Equipment. It's scheduled to\ncnship on July 2, 2015.

\n\n\n

You

didn't order a warranty, so you're on your own when\nthe products
inevitably stop working.

\n\n\n

Sincerely,
Outdoor Equipment

"
```

Let's step through this code one statement at a time:

- First, we import the classes `Template` and `Context`, which both live in the module `django.template`.
- We save the raw text of our template into the variable `raw_template`. Note that we use triple quote marks to designate the string, because it wraps over multiple lines; in contrast, strings within single quote marks cannot be wrapped over multiple lines.
- Next, we create a template object, `t`, by passing `raw_template` to the `Template` class constructor.
- We import the `datetime` module from Python's standard library, because we'll need it in the following statement.
- Then, we create a `Context` object, `c`. The `Context` constructor takes a Python dictionary, which maps variable names to values. Here, for example, we specify that `the_person_name` is '`John Smith`', `company` is '`Outdoor Equipment`', and so forth.
- Finally, we call the `render()` method on our template object, passing it the context. This returns the rendered template – i.e., it replaces template variables with the actual values of the variables, and it executes any template tags.

Note that the “You didn't order a warranty” paragraph was displayed because the `ordered_warranty` variable evaluated to `False`. Also note the date, `July 2, 2015`, which is displayed according to the format string '`F j, Y`'. (We'll explain format strings for the `date` filter in a little while.)

If you're new to Python, you may wonder why this output includes newline characters ('`\n`') rather than displaying the line breaks. That's happening because of a subtlety in the Python interactive interpreter: the call to `t.render(c)` returns a string, and by default the interactive interpreter displays the *representation* of the string, rather than the printed value of the string. If you want to see the string with line breaks displayed as true line breaks rather than '`\n`' characters, use the `print` function:

```
print(t.render(c)).
```

Those are the fundamentals of using the Django template system: just write a template string, create a `Template` object, create a `Context`, and call the `render()` method.

MULTIPLE CONTEXTS, SAME TEMPLATE

Once you have a `Template` object, you can render multiple contexts through it. For example:

```
>>> from django.template import Template, Context  
>>> t = Template('Hello, {{ name }}')  
>>> print (t.render(Context({'name': 'John'})))  
Hello, John  
>>> print (t.render(Context({'name': 'Julie'})))  
Hello, Julie  
>>> print (t.render(Context({'name': 'Pat'})))  
Hello, Pat
```

Whenever you're using the same template source to render multiple contexts like this, it's more efficient to create the `Template` object *once*, and then call `render()` on it multiple times:

```
# Bad  
  
for name in ('John', 'Julie', 'Pat'):  
    t = Template('Hello, {{ name }}')  
    print (t.render(Context({'name': name})))  
  
# Good  
  
t = Template('Hello, {{ name }}')  
for name in ('John', 'Julie', 'Pat'):  
    print (t.render(Context({'name': name})))
```

Django's template parsing is quite fast. Behind the scenes, most of the parsing happens via a call to a single regular expression. This is in stark contrast to XML-based template engines, which incur the overhead of an XML parser and tend to be orders of magnitude slower than Django's template rendering engine.

CONTEXT VARIABLE LOOKUP

In the examples so far, we've passed simple values in the contexts – mostly strings, plus a `datetime.date` example. However, the template system elegantly handles more complex data structures, such as lists, dictionaries, and custom objects.

The key to traversing complex data structures in Django templates is the dot character ('.'). Use a dot to access dictionary keys, attributes, methods, or indices of an object.

This is best illustrated with a few examples. For instance, suppose you're passing a Python dictionary to a template. To access the values of that dictionary by dictionary key, use a dot:

```
>>> from django.template import Template, Context  
>>> person = {'name': 'Sally', 'age': '43'}  
>>> t = Template('{{ person.name }} is {{ person.age }} years old.')  
>>> c = Context({'person': person})  
>>> t.render(c)  
'Sally is 43 years old.'
```

Similarly, dots also allow access of object attributes. For example, a Python `datetime.date` object has `year`, `month`, and `day` attributes, and you can use a dot to access those attributes in a Django template:

```
>>> from django.template import Template, Context  
>>> import datetime  
>>> d = datetime.date(1993, 5, 2)  
>>> d.year  
1993  
>>> d.month  
5  
>>> d.day  
2  
>>> t = Template('The month is {{ date.month }} and the year is {{ date.year }}.')  
>>> c = Context({'date': d})  
>>> t.render(c)  
'The month is 5 and the year is 1993.'
```

This example uses a custom class, demonstrating that variable dots also allow attribute access on arbitrary objects:

```
>>> from django.template import Template, Context  
>>> class Person(object):
```

```
...     def __init__(self, first_name, last_name):
...
...         self.first_name, self.last_name = first_name, last_name

>>> t = Template('Hello, {{ person.first_name }} {{ person.last_name }}.')
>>> c = Context({'person': Person('John', 'Smith')})
>>> t.render(c)
'Hello, John Smith.'
```

Dots can also refer to *methods* on objects. For example, each Python string has the methods `upper()` and `isdigit()`, and you can call those in Django templates using the same dot syntax:

```
>>> from django.template import Template, Context
>>> t = Template('{{ var }} -- {{ var.upper }} -- {{ var.isdigit }}')
>>> t.render(Context({'var': 'hello'}))
'hello -- HELLO -- False'
>>> t.render(Context({'var': '123'}))
'123 -- 123 -- True'
```

Note that you do *not* include parentheses in the method calls. Also, it's not possible to pass arguments to the methods; you can only call methods that have no required arguments. (We explain this philosophy later in this chapter.)

Finally, dots are also used to access list indices, for example:

```
>>> from django.template import Template, Context
>>> t = Template('Item 2 is {{ items.2 }}.')
>>> c = Context({'items': ['apples', 'bananas', 'carrots']})
>>> t.render(c)
'Item 2 is carrots.'
```

Negative list indices are not allowed. For example, the template variable `{{ items.-1 }}` would cause a `TemplateSyntaxError`.

Python Lists

A reminder: Python lists have 0-based indices. The first item is at index 0, the second is at index 1, and so on

Dot lookups can be summarized like this: when the template system encounters a dot in a variable name, it tries the following lookups, in this order:

- Dictionary lookup (e.g., `foo["bar"]`)
- Attribute lookup (e.g., `foo.bar`)
- Method call (e.g., `foo.bar()`)
- List-index lookup (e.g., `foo[2]`)

The system uses the first lookup type that works. It's short-circuit logic.

Dot lookups can be nested multiple levels deep. For instance, the following example uses `{{ person.name.upper }}`, which translates into a dictionary lookup (`person['name']`) and then a method call (`upper()`):

```
>>> from django.template import Template, Context
>>> person = {'name': 'Sally', 'age': '43'}
>>> t = Template('{{ person.name.upper }} is {{ person.age }} years old.')
>>> c = Context({'person': person})
>>> t.render(c)
'SALLY is 43 years old.'
```

METHOD CALL BEHAVIOR

Method calls are slightly more complex than the other lookup types. Here are some things to keep in mind:

- If, during the method lookup, a method raises an exception, 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, the variable will render as the value of the engine's `string_if_invalid` configuration option (an empty string, by default). For 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 .'

```

- A method call will only work if the method has no required arguments. Otherwise, the system will move to the next lookup type (list-index lookup).

By design, Django intentionally limits the amount of logic processing available in the template, so it is not possible to pass arguments to method calls accessed from within templates. Data should be calculated in views and then pass to templates for display.

- Obviously, some methods have side effects, and it would be foolish at best, and possibly even a security hole, to allow the template system to access them.

Say, for instance, you have a `BankAccount` object that has a `delete()` method. If a template includes something like `{{ account.delete }}`, where `account` is a `BankAccount` object, the object would be deleted when the template is rendered!

To prevent this, set the function attribute `alters_data` on the method:

```
def delete(self):  
    # Delete the account  
    delete.alters_data = True
```

The template system won't execute any method marked in this way. Continuing the above example, if a template includes `{{ account.delete }}` and the `delete()` method has the `alters_data=True`, then the `delete()` method will not be executed when the template is rendered, the engine will instead replace the variable with `string_if_invalid`.

NOTE: The dynamically-generated `delete()` and `save()` methods on Django model objects get `alters_data=true` set automatically.

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 an empty string by default. For example:

```
>>> from django.template import Template, Context  
>>> t = Template('Your name is {{ name }}.')  
>>> t.render(Context())  
'Your name is .'  
>>> t.render(Context({'var': 'hello'}))  
'Your name is .'  
>>> t.render(Context({'NAME': 'hello'}))  
'Your name is .'  
>>> t.render(Context({'Name': 'hello'}))  
'Your name is .'
```

This behaviour is better than raising an exception because it's intended to be resilient to human error. In this case, all of the lookups failed because variable names have the wrong case or name. In the real world, it's unacceptable for a Web site to become inaccessible due to a small template syntax error.

BASIC TEMPLATE TAGS AND FILTERS

As we've mentioned already, the template system ships with built-in tags and filters. The sections that follow provide a rundown of the most common tags and filters.

TAGS

IF/ELSE

The `{% if %}` tag evaluates a variable, and if that variable is “True” (i.e., it exists, is not empty, and is not a false Boolean value), the system will display everything between `{% if %}` and `{% endif %}`, for example:

```
{% if today_is_weekend %}  
Welcome to the weekend!  
{% endif %}
```

An `{% else %}` tag is optional:

```
{% if today_is_weekend %}  
Welcome to the weekend!  
{% else %}  
Get back to work.  
{% endif %}
```

The `if` tag may also take one or several `{% elif %}` clauses as well:

```
{% if athlete_list %}  
    Number of athletes: {{ athlete_list|length }}  
{% elif athlete_in_locker_room_list %}  
    Athletes should be out of the locker room soon!  
{% elif ... %}  
    ...  
{% else %}  
    No athletes.  
{% endif %}
```

Python “Truthiness”

In Python and in the Django template system, these objects evaluate to `False` in a Boolean context:

- An empty list (`[]`)
- An empty tuple (`()`)
- An empty dictionary (`{}`)
- An empty string (`''`)
- Zero (`0`)
- The special object `None`
- The object `False` (obviously)
- Custom objects that define their own Boolean context behavior (this is advanced Python usage)

Everything else evaluates to `True`.

The `{% if %}` tag accepts `and`, `or`, `or not` for testing multiple variables, or to negate a given variable. For example:

```
{% 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.

The use of parentheses for controlling order of operations is not supported. If you find yourself needing parentheses, consider performing logic outside the template and passing the result of that as a dedicated template variable. Or, just use nested `{ % if %}` tags, like this:

```
{ % if athlete_list %}  
    { % if coach_list or cheerleader_list %}  
        We have athletes, and either coaches or cheerleaders!  
    { % endif %}  
{ % endif %}
```

Multiple uses of the same logical operator are fine, but you can't combine different operators. For example, this is valid:

```
{ % if athlete_list or coach_list or parent_list or teacher_list %}
```

Make sure to close each `{ % if %}` with an `{ % endif %}`. Otherwise, Django will throw a `TemplateSyntaxError`.

FOR

The `{% for %}` tag allows you to loop over each item in a sequence. As in Python's `for` statement, the syntax is `for X in Y`, where `Y` is the sequence to loop over and `X` is the name of the variable to use for a particular cycle of the loop. Each time through the loop, the template system will render everything between `{% for %}` and `{% endfor %}`.

For example, you could use the following to display a list of athletes given a variable `athlete_list`:

```
{% for athlete in athlete_list %}  
    {{ athlete.name }}  
{% endfor %}
```

Add `reversed` to the tag to loop over the list in reverse:

```
{% for athlete in athlete_list reversed %}  
    ...  
{% endfor %}
```

It's possible to nest `{% for %}` tags:

```
{% for athlete in athlete_list %}  
    {{ athlete.name }}  
        {% for sport in athlete.sports_played %}  
            {{ sport }}  
        {% endfor %}  
    {% endfor %}
```

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 %}
```

A common pattern is to check the size of the list before looping over it, and outputting some special text if the list is empty:

```
{% if athlete_list %}  
    {% for athlete in athlete_list %}  
        {{ athlete.name }}  
    {% endfor %}  
  
{% else %}
```

There are no athletes. Only computer programmers.

```
{% endif %}
```

Because this pattern is so common, the `for` tag supports an optional `{% empty %}` clause that lets you define what to output if the list is empty. This example is equivalent to the previous one:

```
{% for athlete in athlete_list %}  
    {{ athlete.name }}  
{% empty %}
```

There are no athletes. Only computer programmers.

```
{% endfor %}
```

There is no support for “breaking out” of a loop before the loop is finished. If you want to accomplish this, change the variable you’re looping over so that it includes only the values you want to loop over. Similarly, there is no support for a “continue” statement that would instruct the loop processor to return immediately to the front of the loop. (See the section “Philosophies and Limitations” later in this chapter for the reasoning behind this design decision.)

Within each `{% for %}` loop, you get access to a template variable called `forloop`. This variable has a few attributes that give you information about the progress of the loop:

- `forloop.counter` is always set to an integer representing the number of times the loop has been entered. This is one-indexed, so the first time through the loop, `forloop.counter` will be set to 1. Here's an example:

```
{% for item in todo_list %}  
{{ forloop.counter }}: {{ item }}  
{% endfor %}
```

- `forloop.counter0` is like `forloop.counter`, except it's zero-indexed. Its value will be set to 0 the first time through the loop.
- `forloop.revcounter` is always set to an integer representing the number of remaining items in the loop. The first time through the loop, `forloop.revcounter` will be set to the total number of items in the sequence you're traversing. The last time through the loop, `forloop.revcounter` will be set to 1.
- `forloop.revcounter0` is like `forloop.revcounter`, except it's zero-indexed. The first time through the loop, `forloop.revcounter0` will be set to the number of elements in the sequence minus 1. The last time through the loop, it will be set to 0.
- `forloop.first` is a Boolean value set to `True` if this is the first time through the loop. This is convenient for special-casing:

```
{% for object in objects %}  
  {% if forloop.first %}  
    • {%- else %}  
    • {%- endif %}  
      {{ object }}  
  {% endfor %}
```

- `forloop.last` is a Boolean value set to `True` if this is the last time through the loop. A common use for this is to put pipe characters between a list of links:

```
{% for link in links %}{{ link }}{% if not forloop.last %} | {% endif %}{% endfor %}
```

The above template code might output something like this:

```
Link1 | Link2 | Link3 | Link4
```

Another common use for this is to put a comma between words in a list:

```
Favorite places:
```

```
{% for p in places %}{{ p }}{% if not forloop.last %}, {% endif %}{% endfor %}
```

- `forloop.parentloop` is a reference to the `forloop` object for the *parent* loop, in case of nested loops. Here's an example:

```
{% for country in countries %}

<table>

    {% for city in country.city_list %}

        <tr>

            <td>Country #{{ forloop.parentloop.counter }}</td>

            <td>City #{{ forloop.counter }}</td>

            <td>{{ city }}</td>

        </tr>

    {% endfor %}

</table>

{% endfor
```

The `forloop` variable is only available within loops. After the template parser has reached `{% endfor %}`, `forloop` disappears.

Context and the `forloop` Variable

Inside the `{% for %}` block, the existing variables are moved out of the way to avoid overwriting the `forloop` variable. Django exposes this moved context in `forloop.parentloop`. You generally don't need to worry about this, but if you supply a template variable named `forloop` (though we advise against it), it will be named `forloop.parentloop` while inside the `{% for %}` block.

IFEQUAL/IFNOTEQUAL

The Django template system deliberately is not a full-fledged programming language and thus does not allow you to execute arbitrary Python statements. (More on this idea in the section “Philosophies and Limitations.”) However, it’s quite a common template requirement to compare two values and display something if they’re equal – and Django provides an `{% ifequal %}` tag for that purpose.

The `{% ifequal %}` tag compares two values and displays everything between `{% ifequal %}` and `{% endifequal %}` if the values are equal.

This example compares the template variables `user` and `currentuser`:

```
{% ifequal user currentuser %}  
Welcome!  
{% endifequal %}
```

The arguments can be hard-coded strings, with either single or double quotes, so the following is valid:

```
{% ifequal section 'sitenews' %}  
Site News  
{% endifequal %}  
{% ifequal section "community" %}  
Community  
{% endifequal %}
```

Just like `{% if %}`, the `{% ifequal %}` tag supports an optional `{% else %}`:

```
{% ifequal section 'sitenews' %}  
Site News  
{% else %}  
No News Here  
{% endifequal %}
```

Only template variables, strings, integers, and decimal numbers are allowed as arguments to `{% ifequal %}`. These are valid examples:

```
{% ifequal variable 1 %}  
{% ifequal variable 1.23 %}  
{% ifequal variable 'foo' %}  
{% ifequal variable "foo" %}
```

Any other types of variables, such as Python dictionaries, lists, or Booleans, can't be hard-coded in `{% ifequal %}`. These are invalid examples:

```
{% ifequal variable True %}  
{% ifequal variable [1, 2, 3] %}  
{% ifequal variable {'key': 'value'} %}
```

If you need to test whether something is true or false, use the `{% if %}` tags instead of `{% ifequal %}`.

An alternative to the `ifequal` tag is to use the `if` tag and the `==` operator.

The `ifnotequal` tag is identical to the `ifequal` tag, except that it tests whether the two arguments are not equal. An alternative to the `ifnotequal` tag is to use the `if` tag and the `!=` operator.

COMMENTS

Just as in HTML or Python, the Django template language allows for comments. To designate a comment, use `{# #}`:

```
{# This is a comment #}
```

The comment will not be output when the template is rendered.

Comments using this syntax cannot span multiple lines. This limitation improves template parsing performance. In the following template, the rendered output will look exactly the same as the template (i.e., the comment tag will not be parsed as a comment):

```
This is a {# this is not  
a comment #}  
  
test.
```

If you want to use multi-line comments, use the `{% comment %}` template tag, like this:

```
{% comment %}  
This is a  
multi-line comment.  
{% endcomment %}
```

Comment tags cannot be nested.

FILTERS

As explained earlier in this chapter, template filters are simple ways of altering the value of variables before they're displayed. Filters use a pipe character, like this:

```
{{ name|lower }}
```

This displays the value of the `{{ name }}` variable after being filtered through the `lower` filter, which converts text to lowercase.

Filters can be *chained* – that is, they can be used in tandem such that the output of one filter is applied to the next. Here's an example that takes the first element in a list and converts it to uppercase:

```
{{ my_list|first|upper }}
```

Some filters take arguments. A filter argument comes after a colon and is always in double quotes. For example:

```
{{ bio|truncatewords:"30" }}
```

This displays the first 30 words of the `bio` variable.

The following are a few of the most important filters. Appendix E covers the rest.

- `addslashes`: Adds a backslash before any backslash, single quote, or double quote. This is useful for escaping strings.
- `date`: Formats a `date` or `datetime` object according to a format string given in the parameter, for example:

```
{{ pub_date|date:"F j, Y" }}
```

Format strings are defined in Appendix E.

- `length`: Returns the length of the value. For a list, this returns the number of elements. For a string, this returns the number of characters. If the variable is `undefined`, `length` returns '0'.

PHILOSOPHIES AND LIMITATIONS

Now that you've gotten a feel for the Django Template Language(DTL), it is probably time to explain the basic design philosophy behind the DTL.

First and foremost, the **limitations to the DTL are intentional**.

Django was developed in the high volume, ever-changing environment of an online newsroom. The original creators of Django had a very definite set of philosophies in creating the DTL. These philosophies remain core to Django today. They are:

SEPARATE LOGIC FROM PRESENTATION

A template system is 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 Web sites use some sort of common site-wide 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 template system intentionally doesn't allow the following:

- Assignment to variables
- Advanced logic

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 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.

Having worked with many different templating systems myself over the years, I wholeheartedly endorse this approach – the DTL and the way it has been designed is one of the major pluses of the Django framework – when the pressure is on to Get Stuff Done, and you have both designers and programmers trying to communicate and get all the last minute tasks done, Django just gets out of the way and lets each team concentrate on what they are good at. Once you have found this out for yourself through real-life practice, you will find out very quickly why Django really is the ‘framework for perfectionists with deadlines’.

With all this in mind, Django is flexible – it does not require you to use the DTL. More than any other component of Web applications, template syntax is highly subjective, and programmers’ opinions vary wildly. The fact that Python alone has dozens, if not hundreds, of open source template-language implementations supports this point. Each was likely created because its developer deemed all existing template languages inadequate.

Because Django is intended to be a full-stack Web framework that provides all the pieces necessary for Web developers to be productive, most times it’s *more convenient* to use the DTL, but it’s not a strict requirement in any sense.

USING TEMPLATES IN VIEWS

You’ve learned the basics of using the template system; now let’s use this knowledge to create a view. Recall the `current_datetime` view in `mysite.views`, which we started in the previous chapter. Here’s what it looks like:

```
from django.http import HttpResponse

import datetime

def current_datetime(request):

    now = datetime.datetime.now()

    html = "It is now %s." % now

    return HttpResponse(html)
```

Let’s change this view to use Django’s template system. At first, you might think to do something like this:

```
from django.template import Template, Context

from django.http import HttpResponse

import datetime
```

```

def current_datetime(request):
    now = datetime.datetime.now()

    t = Template("It is now {{ current_date }}.")

    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)

```

Sure, that uses the template system, but it doesn't solve the problems we pointed out in the introduction of this chapter. Namely, the template is still embedded in the Python code, so true separation of data and presentation isn't achieved. Let's fix that by putting the template in a *separate file*, which this view will load.

You might first consider saving your template somewhere on your filesystem and using Python's built-in file-opening functionality to read the contents of the template. Here's what that might look like, assuming the template was saved as the file /home/djangouser/templates/mytemplate.html:

```

from django.template import Template, Context
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()

    # Simple way of using templates from the filesystem.

    # This is BAD because it doesn't account for missing files!
    fp = open('/home/djangouser/templates/mytemplate.html')

    t = Template(fp.read())
    fp.close()

    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)

```

This approach, however, is inelegant for these reasons:

- It doesn't handle the case of a missing file. If the file `mytemplate.html` doesn't exist or isn't readable, the `open()` call will raise an `IOError` exception.

- It hard-codes your template location. If you were to use this technique for every view function, you'd be duplicating the template locations. Not to mention it involves a lot of typing!
- It includes a lot of boring boilerplate code. You've got better things to do than to write calls to `open()`, `fp.read()`, and `fp.close()` each time you load a template.

To solve these issues, we'll use *template loading* and *template directories*.

TEMPLATE LOADING

Django provides a convenient and powerful API for loading templates from the filesystem, with the goal of removing redundancy both in your template-loading calls and in your templates themselves.

In order to use this template-loading API, first you'll need to tell the framework where you store your templates. The place to do this is in your settings file – the `settings.py` file that we mentioned last chapter, when we introduced the `ROOT_URLCONF` setting.

If you're following along, open your `settings.py` and find the `TEMPLATES` setting. It's a list of configurations, one for each engine:

```
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. By convention, when `APPS_DIRS` is set to `True`, `DjangoTemplates` looks for a “templates” subdirectory in each of the `INSTALLED_APPS`. This allows the template engine to find application templates even if `DIRS` is empty.
- `OPTIONS` contains backend-specific settings.

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.

TEMPLATE DIRECTORIES

`DIRS`, by default, is an empty list. To tell Django’s template-loading mechanism where to look for templates, pick a directory where you’d like to store your templates and add it to `DIRS`, like so:

```
'DIRS': [  
    '/home/html/example.com',  
    '/home/html/default',  
],
```

There are a few things to note:

- Unless you are building a very simple program with no apps, you are better off leaving `DIRS` empty. The default settings file configures `APP_DIRS` to `True`, so you are better off having a “templates” subdirectory in your Django app. More on why this is a good thing in [Chapter 14](#).
- If you want to have a set of master templates at project root, e.g. `mysite/templates`, you *do* need to set `DIRS`, like so:

```
'DIRS': [os.path.join(BASE_DIR, 'templates')],
```

Your templates directory does not have to be called 'templates', by the way – Django doesn't put any restrictions on the names you use – but it makes your project structure much easier to understand if you stick to convention.

- If you don't want to go with the default, or can't for some reason, you can specify any directory you want, as long as the directory and templates within that directory are readable by the user account under which your Web server runs.
- If you're on Windows, include your drive letter and use Unix-style forward slashes rather than backslashes, as follows:

```
'DIRS': [  
    'C:/www/django/templates',  
]
```

As we have not yet created a Django app, you will have to set DIRS to [os.path.join(BASE_DIR, 'templates')] as per the example above for the code below to work as expected.

With DIRS set, the next step is to change the view code to use Django's template-loading functionality rather than hard-coding the template paths. Returning to our current_datetime view, let's change it like so:

```
from django.template.loader import get_template  
  
from django.template import Context  
  
from django.http import HttpResponse  
  
import datetime  
  
def current_datetime(request):  
  
    now = datetime.datetime.now()  
  
    t = get_template('current_datetime.html')  
  
    html = t.render(Context({'current_date': now}))  
  
    return HttpResponse(html)
```

In this example, we're using the function django.template.loader.get_template() rather than loading the template from the filesystem manually. The get_template() function takes a template name as its argument, figures out where the template lives on the filesystem, opens that file, and returns a compiled Template object.

Our template in this example is `current_datetime.html`, but there's nothing special about that `.html` extension. You can give your templates whatever extension makes sense for your application, or you can leave off extensions entirely.

To determine the location of the template on your filesystem, `get_template()` will look in order:

- If `APP_DIRS` is set to `True`, and assuming you are using the DTL, it will look for a “templates” directory in the current app.
- If it does not find your template in the current app, `get_template()` combines your template directories from `DIRS` with the template name that you pass to `get_template()` and steps through each of them in order until it finds your template. For example, if the first entry in your `DIRS` is set to `'/home/django/mysite/templates'`, the above `get_template()` call would look for the template `/home/django/mysite/templates/current_datetime.html`.
- If `get_template()` cannot find the template with the given name, it raises a `TemplateDoesNotExist` exception.

To see what a template exception looks like, fire up the Django development server again by running `python manage.py runserver` within your Django project’s directory. Then, point your browser at the page that activates the `current_datetime` view (e.g., `http://127.0.0.1:8000/time/`). Assuming your `DEBUG` setting is set to `True` and you haven’t yet created a `current_datetime.html` template, you should see a Django error page highlighting the `TemplateDoesNotExist` error.

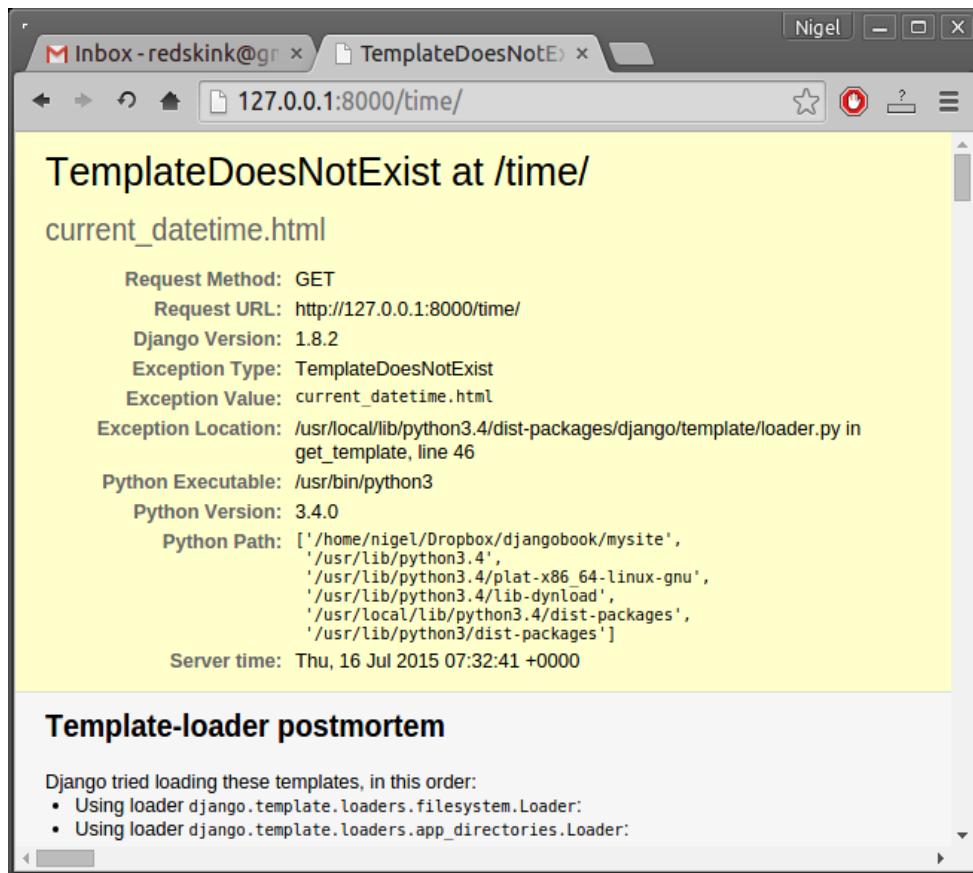


Figure 3-1: The error page shown when a template cannot be found.

This error page is similar to the one we explained in [Chapter 2](#), with one additional piece of debugging information: a “Template-loader postmortem” section. This section tells you which templates Django tried to load, along with the reason each attempt failed (e.g., “File does not exist”). This information is invaluable when you’re trying to debug template-loading errors.

Moving along, create the `current_datetime.html` file using the following template code:

```
It is now {{ current_date }}.
```

Save this file to `mysite/templates` (create the ‘templates’ directory if you have not done so already). Refresh the page in your Web browser, and you should see the fully rendered page.

RENDER()

So far, we’ve shown you how to load a template, fill a `Context` and return an `HttpResponse` object with the result of the rendered template. Next step was to optimize it to use `get_template()` instead of hard-coding templates and template paths. We took you through

this process to ensure you understood how Django templates are loaded and rendered to your browser.

In practice, Django provides a much easier way to do this.

Django's developers recognized that because this is such a common idiom, Django needed a shortcut that could do all this in one line of code.

This shortcut is a function called `render()`, which lives in the module `django.shortcuts`. Most of the time, you'll be using `render()` rather than loading templates and creating `Context` and `HttpResponse` objects manually – unless your employer judges your work by total lines of code written, that is.

Here's the ongoing `current_datetime` example rewritten to use `render()`:

```
from django.shortcuts import render

import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    return render(request, 'current_datetime.html', {'current_date': now})
```

What a difference! Let's step through the code changes:

- We no longer have to import `get_template`, `Template`, `Context`, or `HttpResponse`. Instead, we import `django.shortcuts.render`. The `import datetime` remains.
- Within the `current_datetime` function, we still calculate `now`, but the template loading, context creation, template rendering, and `HttpResponse` creation are all taken care of by the `render()` call. Because `render()` returns an `HttpResponse` object, we can simply `return` that value in the view.

The first argument to `render()` is the `request`, the second is the name of the template to use. The third argument, if given, should be a dictionary to use in creating a `Context` for that template. If you don't provide a third argument, `render()` will use an empty dictionary.

SUBDIRECTORIES IN GET_TEMPLATE()

It can get unwieldy to store all of your templates in a single directory. You might like to store templates in subdirectories of your template directory, and that's fine. In fact, we

recommend doing so; some more advanced Django features (such as the generic views system, which we cover in [Chapter 10](#)) expect this template layout as a default convention.

Storing templates in subdirectories of your template directory is easy. In your calls to `get_template()`, just include the subdirectory name and a slash before the template name, like so:

```
t = get_template('dateapp/current_datetime.html')
```

Because `render()` is a small wrapper around `get_template()`, you can do the same thing with the second argument to `render()`, like this:

```
return render(request, 'dateapp/current_datetime.html', {'current_date': now})
```

There's no limit to the depth of your subdirectory tree. Feel free to use as many subdirectories as you like.

Note

Windows users, be sure to use forward slashes rather than backslashes. `get_template()` assumes a Unix-style file name designation.

THE INCLUDE TEMPLATE TAG

Now that we've covered the template-loading mechanism, we can introduce a built-in template tag that takes advantage of it: `{% include %}`. This tag allows you to include the contents of another template. The argument to the tag should be the name of the template to include, and the template name can be either a variable or a hard-coded (quoted) string, in either single or double quotes. Anytime you have the same code in multiple templates, consider using an `{% include %}` to remove the duplication.

These two examples include the contents of the template `nav.html`. The examples are equivalent and illustrate that either single or double quotes are allowed:

```
{% include 'nav.html' %}  
{% include "nav.html" %}
```

This example includes the contents of the template `includes/nav.html`:

```
{% include 'includes/nav.html' %}
```

This example includes the contents of the template whose name is contained in the variable `template_name`:

```
{% include template_name %}
```

As in `get_template()`, the file name of the template is determined by either adding the path to the “templates” directory in the current Django app (if `APPS_DIR` is `True`) or by adding the template directory from `DIRS` to the requested template name.

Included templates are evaluated with the context of the template that’s including them. For example, consider these two templates:

```
# mypage.html  
  
{% include "includes/nav.html" %}  
  
{{ title }}  
  
# includes/nav.html  
  
You are in: {{ current_section }}
```

If you render `mypage.html` with a context containing `current_section`, then the variable will be available in the “included” template, as you would expect.

If, in an `{% include %}` tag, a template with the given name isn’t found, Django will do one of two things:

- If `DEBUG` is set to `True`, you’ll see the `TemplateDoesNotExist` exception on a Django error page.
- If `DEBUG` is set to `False`, the tag will fail silently, displaying nothing in the place of the tag.

Note

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.

TEMPLATE INHERITANCE

Our template examples so far have been tiny HTML snippets, but in the real world, you'll be using Django's template system to create entire HTML pages. This leads to a common Web development problem: across a Web site, how does one reduce the duplication and redundancy of common page areas, such as sitewide navigation?

A classic way of solving this problem is to use *server-side includes*, directives you can embed within your HTML pages to "include" one Web page inside another. Indeed, Django supports that approach, with the `{% include %}` template tag just described. But the preferred way of solving this problem with Django is to use a more elegant strategy called *template inheritance*.

In essence, template inheritance lets you build a base "skeleton" template that contains all the common parts of your site and defines "blocks" that child templates can override.

Let's see an example of this by creating a more complete template for our `current_datetime` view, by editing the `current_datetime.html` file:

```
<h1>My helpful timestamp site</h1>
```

```
It is now {{ current_date }}.
```

```
Thanks for visiting my site.
```

That looks just fine, but what happens when we want to create a template for another view – say, the `hours_ahead` view from [Chapter 2](#)? If we want again to make a nice, valid, full HTML template, we'd create something like:

```
<h1>My helpful timestamp site</h1>
```

```
In {{ hour_offset }} hour(s), it will be {{ next_time }}.
```

```
Thanks for visiting my site.
```

Clearly, we've just duplicated a lot of HTML. Imagine if we had a more typical site, including a navigation bar, a few style sheets, perhaps some JavaScript – we'd end up putting all sorts of redundant HTML into each template.

The server-side include solution to this problem is to factor out the common bits in both templates and save them in separate template snippets, which are then included in each template. Perhaps you'd store the top bit of the template in a file called `header.html`:

And perhaps you'd store the bottom bit in a file called `footer.html`:

Thanks for visiting my site.

With an include-based strategy, headers and footers are easy. It's the middle ground that's messy. In this example, both pages feature a title – “My helpful timestamp site” – but that title can't fit into `header.html` because the title on both pages is different. If we included the `h1` in the header, we'd have to include the title, which wouldn't allow us to customize it per page. See where this is going?

Django's template inheritance system solves these problems. You can think of it as an “inside-out” version of server-side includes. Instead of defining the snippets that are *common*, you define the snippets that are *different*.

The first step is to define a *base template* – a skeleton of your page that *child templates* will later fill in. Here's a base template for our ongoing example:

```
<h1>My helpful timestamp site</h1>
    {% block content %}{% endblock %}
    {% block footer %}
    Thanks for visiting my site.
    {% endblock %}
```

This template, which we'll call `base.html`, defines a simple HTML skeleton document that we'll use for all the pages on the site. It's the job of child templates to override, or add to, or leave alone the contents of the blocks. (If you're following along, save this file to your template directory as `base.html`.)

We're using a template tag here that you haven't seen before: the `{% block %}` tag. All the `{% block %}` tags do is tell the template engine that a child template may override those portions of the template.

Now that we have this base template, we can modify our existing `current_datetime.html` template to use it:

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

{% block title %}The current time{% endblock %}

{% block content %}

It is now {{ current_date }}.

{% endblock %}
```

While we're at it, let's create a template for the `hours_ahead` view from [Chapter 3](#). (If you're following along with code, we'll leave it up to you to change `hours_ahead` to use the template system instead of hard-coded HTML.) Here's what that could look like:

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

{% block title %}Future time{% endblock %}

{% block content %}

In {{ hour_offset }} hour(s), it will be {{ next_time }}.

{% endblock %}
```

Isn't this beautiful? Each template contains only the code that's *unique* to that template. No redundancy needed. If you need to make a site-wide design change, just make the change to `base.html`, and all of the other templates will immediately reflect the change.

Here's how it works. When you load the template `current_datetime.html`, the template engine sees the `{% extends %}` tag, noting that this template is a child template. The engine immediately loads the parent template – in this case, `base.html`.

At that point, the template engine notices the three `{% block %}` tags in `base.html` and replaces those blocks with the contents of the child template. So, the title we've defined in `{% block title %}` will be used, as will the `{% block content %}`.

Note that since the child template doesn't define the `footer` block, the template system uses the value from the parent template instead. Content within a `{% block %}` tag in a parent template is always used as a fallback.

Inheritance doesn't affect the template context. In other words, any template in the inheritance tree will have access to every one of your template variables from the context.

You can use as many levels of inheritance as needed. One common way of using inheritance is the following three-level approach:

1. Create a `base.html` template that holds the main look and feel of your site. This is the stuff that rarely, if ever, changes.
2. Create a `base_SECTION.html` template for each “section” of your site (e.g., `base_photos.html` and `base_forum.html`). These templates extend `base.html` and include section-specific styles/design.
3. Create individual templates for each type of page, such as a forum page or a photo gallery. These templates extend the appropriate section template.

This approach maximizes code reuse and makes it easy to add items to shared areas, such as section-wide navigation.

Here are some guidelines for working with template inheritance:

- If you use `{% extends %}` in a template, it must be the first template tag in that template. Otherwise, template inheritance won’t work.
- Generally, the more `{% block %}` tags in your base templates, the better. Remember, child templates don’t have to define all parent blocks, so you can fill in reasonable defaults in a number of blocks, and then define only the ones you need in the child templates. It’s better to have more hooks than fewer hooks.
- If you find yourself duplicating code in a number of templates, it probably means you should move that code to a `{% block %}` in a parent template.
- If you need to get the content of the block from the parent template, use `{{ block.super }}`, which is a “magic” variable providing the rendered text of the parent template. This is useful if you want to add to the contents of a parent block instead of completely overriding it.
- You may not 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.

- The template name you pass to `{% extends %}` is loaded using the same method that `get_template()` uses. That is, the template name is appended to your `DIRS` setting, or the “templates” folder in the current Django app.
- In most cases, the argument to `{% extends %}` will be a string, but it can also be a variable, if you don’t know the name of the parent template until runtime. This lets you do some cool, dynamic stuff.

WHAT'S NEXT?

You now have the basics of Django’s template system under your belt. What’s next?

Most modern Web sites are *database-driven*: the content of the Web site is stored in a relational database. This allows a clean separation of data and logic (in the same way views and templates allow the separation of logic and display.)

The next chapter covers the tools Django gives you to interact with a database.

CHAPTER 4: MODELS

In [Chapter 2](#), we covered the fundamentals of building dynamic Web sites with Django: setting up views and URLconfs. As we explained, a view is responsible for doing *some arbitrary logic*, and then returning a response. In one of the examples, our arbitrary logic was to calculate the current date and time.

In modern Web applications, the arbitrary logic often involves interacting with a database. Behind the scenes, a *database-driven Web site* connects to a database server, retrieves some data out of it, and displays that data on a Web page. The site might also provide ways for site visitors to populate the database on their own.

Many complex Web sites provide some combination of the two. Amazon.com, for instance, is a great example of a database-driven site. Each product page is essentially a query into Amazon’s product database formatted as HTML, and when you post a customer review, it gets inserted into the database of reviews.

Django is well suited for making database-driven Web sites, because it comes with easy yet powerful tools for performing database queries using Python. This chapter explains that functionality: Django’s database layer.

Note: While it's not strictly necessary to know basic relational database theory and SQL in order to use Django's database layer, it's highly recommended. An introduction to those concepts is beyond the scope of this book, but keep reading even if you're a database newbie. You'll probably be able to follow along and grasp concepts based on the context.

THE “DUMB” WAY TO DO DATABASE QUERIES IN VIEWS

Just as [Chapter 2](#) detailed a “dumb” way to produce output within a view (by hard-coding the text directly within the view), there's a “dumb” way to retrieve data from a database in a view. It's simple: just use any existing Python library to execute an SQL query and do something with the results.

In this example view, we use the `MySQLdb` library to connect to a MySQL database, retrieve some records, and feed them to a template for display as a Web page:

```
from django.shortcuts import render

import MySQLdb

def book_list(request):

    db=MySQLdb.connect(user='me',db='mydb',                                passwd='secret',
host='localhost')

    cursor = db.cursor()

    cursor.execute('SELECT name FROM books ORDER BY name')

    names = [row[0] for row in cursor.fetchall()]

    db.close()

    return render(request, 'book_list.html', {'names': names})
```

This approach works, but some problems should jump out at you immediately:

- We're hard-coding the database connection parameters. Ideally, these parameters would be stored in the Django configuration.
- We're having to write a fair bit of boilerplate code: creating a connection, creating a cursor, executing a statement, and closing the connection. Ideally, all we'd have to do is specify which results we wanted.
- It ties us to MySQL. If, down the road, we switch from MySQL to PostgreSQL, we'll most likely have to rewrite a large amount of our code. Ideally, the database server we're using would be abstracted, so that a database server change could be made in

a single place. (This feature is particularly relevant if you’re building an open-source Django application that you want to be used by as many people as possible.)

As you might expect, Django’s database layer solves these problems.

CONFIGURING THE DATABASE

With all of that philosophy in mind, let’s start exploring Django’s database layer. First, lets explore the initial configuration that was added to `settings.py` when we created the application:

```
# Database

# https://docs.djangoproject.com/en/1.8/ref/settings/#databases

DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.sqlite3',

        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),

    }

}
```

The default setup is pretty simple. Here’s a rundown of each setting.

- `ENGINE` tells Django which database engine to use. As we are using SQLite in the examples in this book, we will leave it to the default `django.db.backends.sqlite3`
- `NAME` tells Django the name of your database. For example:

```
'NAME': 'mydb',
```

Since we’re using SQLite, `startproject` created a full filesystem path to the database file for us.

This is it for the default setup – you don’t need to change anything to run the code in this book, I have included this simply to give you an idea of how simple it is to configure databases in Django.

For a detailed description on how to set up the various databases supported by Django, see [Chapter 23](#).

YOUR FIRST APP

Now that you've verified the connection is working, it's time to create a *Django app* – a bundle of Django code, including models and views, that lives together in a single Python package and represents a full Django application.

It's worth explaining the terminology here, because this tends to trip up beginners. We've already created a *project*, in [Chapter 1](#), so what's the difference between a *project* and an *app*? The difference is that of configuration vs. code:

- A project is an instance of a certain set of Django apps, plus the configuration for those apps.

Technically, the only requirement of a project is that it supplies a settings file, which defines the database connection information, the list of installed apps, the `DIRS`, and so forth.

- An app is a portable set of Django functionality, usually including models and views, that lives together in a single Python package.

For example, Django comes with a number of apps, such as the automatic admin interface. A key thing to note about these apps is that they're portable and reusable across multiple projects.

There are very few hard-and-fast rules about how you fit your Django code into this scheme. If you're building a simple Web site, you may use only a single app. If you're building a complex Web site with several unrelated pieces such as an e-commerce system and a message board, you'll probably want to split those into separate apps so that you'll be able to reuse them individually in the future.

Indeed, you don't necessarily need to create apps at all, as evidenced by the example view functions we've created so far in this book. In those cases, we simply created a file called `views.py`, filled it with view functions, and pointed our URLconf at those functions. No "apps" were needed.

However, there's one requirement regarding the app convention: if you're using Django's database layer (`models`), you must create a Django app. Models must live within apps. Thus, in order to start writing our models, we'll need to create a new app.

Within the `mysite` project directory, type this command to create a `books` app:

```
python manage.py startapp books
```

This command does not produce any output, but it does create a `books` directory within the `mysite` directory. Let's look at the contents of that directory:

```
books/
    /migrations
        __init__.py
    admin.py
    models.py
    tests.py
    views.py
```

These files will contain the models and views for this app.

Have a look at `models.py` and `views.py` in your favorite text editor. Both files are empty, except for comments and an import in `models.py`. This is the blank slate for your Django app.

DEFINING MODELS IN PYTHON

As we discussed earlier in this chapter, the “M” in “MTV” stands for “Model.” A Django model is a description of the data in your database, represented as Python code. It’s your data layout – the equivalent of your SQL `CREATE TABLE` statements – except it’s in Python instead of SQL, and it includes more than just database column definitions. Django uses a model to execute SQL code behind the scenes and return convenient Python data structures representing the rows in your database tables. Django also uses models to represent higher-level concepts that SQL can’t necessarily handle.

If you’re familiar with databases, your immediate thought might be, “Isn’t it redundant to define data models in Python instead of in SQL?” Django works the way it does for several reasons:

- Introspection requires overhead and is imperfect. In order to provide convenient data-access APIs, Django needs to know the database layout *somewhat*, and there are two ways of accomplishing this. The first way would be to explicitly describe the data in Python, and the second way would be to introspect the database at runtime to determine the data models.

This second way seems cleaner, because the metadata about your tables lives in only one place, but it introduces a few problems. First, introspecting a database at runtime obviously requires overhead. If the framework had to introspect the database each time it processed a request, or even only when the Web server was initialized, this would incur an unacceptable level of overhead. (While some believe that level of overhead is acceptable, Django's developers aim to trim as much framework overhead as possible.) Second, some databases, notably older versions of MySQL, do not store sufficient metadata for accurate and complete introspection.

- Writing Python is fun, and keeping everything in Python limits the number of times your brain has to do a “context switch.” It helps productivity if you keep yourself in a single programming environment/mentality for as long as possible. Having to write SQL, then Python, and then SQL again is disruptive.
- Having data models stored as code rather than in your database makes it easier to keep your models under version control. This way, you can easily keep track of changes to your data layouts.
- SQL allows for only a certain level of metadata about a data layout. Most database systems, for example, do not provide a specialized data type for representing email addresses or URLs. Django models do. The advantage of higher-level data types is higher productivity and more reusable code.
- SQL is inconsistent across database platforms. If you’re distributing a Web application, for example, it’s much more pragmatic to distribute a Python module that describes your data layout than separate sets of `CREATE TABLE` statements for MySQL, PostgreSQL, and SQLite.

A drawback of this approach, however, is that it’s possible for the Python code to get out of sync with what’s actually in the database. If you make changes to a Django model, you’ll need to make the same changes inside your database to keep your database consistent with the model. I’ll show you how to handle this problem when we discuss *migrations* later in this chapter.

Finally, we should note that Django includes a utility that can generate models by introspecting an existing database. This is useful for quickly getting up and running with legacy data. We’ll cover this in [Chapter 23](#).

YOUR FIRST MODEL

As an ongoing example in this chapter and the next chapter, we'll focus on a basic book/author/publisher data layout. We use this as our example because the conceptual relationships between books, authors, and publishers are well known, and this is a common data layout used in introductory SQL textbooks. You're also reading a book that was written by authors and produced by a publisher!

We'll suppose the following concepts, fields, and relationships:

- An author has a first name, a last name and an email address.
- A publisher has a name, a street address, a city, a state/province, a country, and a Web site.
- A book has a title and a publication date. It also has one or more authors (a many-to-many relationship with authors) and a single publisher (a one-to-many relationship – aka foreign key – to publishers).

The first step in using this database layout with Django is to express it as Python code. In the `models.py` file that was created by the `startapp` command, enter the following:

```
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 Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
```

```
    publisher = models.ForeignKey(Publisher)  
    publication_date = models.DateField()
```

Let's quickly examine this code to cover the basics. The first thing to notice is that each model is represented by a Python class that is a subclass of `django.db.models.Model`. The parent class, `Model`, contains all the machinery necessary to make these objects capable of interacting with a database – and that leaves our models responsible solely for defining their fields, in a nice and compact syntax. Believe it or not, this is all the code we need to write to have basic data access with Django.

Each model generally corresponds to a single database table, and each attribute on a model generally corresponds to a column in that database table. The attribute name corresponds to the column's name, and the type of field (e.g., `CharField`) corresponds to the database column type (e.g., `varchar`). For example, the `Publisher` model is equivalent to the following table (assuming PostgreSQL `CREATE TABLE` syntax):

```
CREATE TABLE "books_publisher" (  
    "id" serial NOT NULL PRIMARY KEY,  
    "name" varchar(30) NOT NULL,  
    "address" varchar(50) NOT NULL,  
    "city" varchar(60) NOT NULL,  
    "state_province" varchar(30) NOT NULL,  
    "country" varchar(50) NOT NULL,  
    "website" varchar(200) NOT NULL  
) ;
```

Indeed, Django can generate that `CREATE TABLE` statement automatically, as we'll show you in a moment.

The exception to the one-class-per-database-table rule is the case of many-to-many relationships. In our example models, `Book` has a `ManyToManyField` called `authors`. This designates that a book has one or many authors, but the `Book` database table doesn't get an `authors` column. Rather, Django creates an additional table – a many-to-many “join table” – that handles the mapping of books to authors.

For a full list of field types and model syntax options, see [Appendix B](#).

Finally, note we haven't explicitly defined a primary key in any of these models. Unless you instruct it otherwise, Django automatically gives every model an auto-incrementing integer primary key field called `id`. Each Django model is required to have a single-column primary key.

INSTALLING THE MODEL

We've written the code; now let's create the tables in our database. In order to do that, the first step is to *activate* these models in our Django project. We do that by adding the `books` app to the list of "installed apps" in the settings file.

Edit the `settings.py` file again, and look for the `INSTALLED_APPS` setting. `INSTALLED_APPS` tells Django which apps are activated for a given project. By default, it looks something like this:

```
INSTALLED_APPS = (  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
)
```

To register our "books" app, add '`books`' to the `INSTALLED_APPS` list, so the setting ends up looking like this:

```
INSTALLED_APPS = (  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'books',
```

)

You'll need to be sure to include the trailing comma in `INSTALLED_APPS`, because it's a single-element tuple. By the way, it is best practice to put a comma after *every* element of a tuple, regardless of whether the tuple has only a single element. This avoids the issue of forgetting commas, and there's no penalty for using that extra comma.)

'books' refers to the "books" app we're working on. Each app in `INSTALLED_APPS` is represented by its full Python path – that is, the path of packages, separated by dots, leading to the app package.

Now that the Django app has been activated in the settings file, we can create the database tables in our database. First, let's validate the models by running this command:

```
python manage.py check
```

The `check` command runs the Django system check framework – a set of static checks for validating Django projects. If all is well, you'll see the message `System check identified no issues (0 silenced)`. If you don't, make sure you typed in the model code correctly. The error output should give you helpful information about what was wrong with the code.

Any time you think you have problems with your models, run `python manage.py check`. It tends to catch all the common model problems.

If your models are valid, run the following command to tell Django that you have made some changes to your models (in this case, you have made a new one):

```
python manage.py makemigrations books
```

You should see something similar to the following:

```
Migrations for 'books':
```

```
  0001_initial.py:
```

- Create model Author
- Create model Book
- Create model Publisher
- Add field publisher to book

Migrations are how Django stores changes to your models (and thus your database schema) – they're just files on disk. In this instance, you will find a file names `0001_initial.py` in the 'migrations' folder of the `books` app.

The `migrate` command will take your latest migration file and update your database schema automatically, 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 books 0001
```

You should see something similar to the following (reformatted for readability):

```
BEGIN;
```

```
CREATE TABLE "books_author" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(40) NOT NULL,
    "email" varchar(254) NOT NULL
);

CREATE TABLE "books_book" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "title" varchar(100) NOT NULL,
    "publication_date" date NOT NULL
);

CREATE TABLE "books_book_authors" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "book_id" integer NOT NULL REFERENCES "books_book" ("id"),
    "author_id" integer NOT NULL REFERENCES "books_author" ("id"),
    UNIQUE ("book_id", "author_id")
);

CREATE TABLE "books_publisher" (
```

```

"id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
"name" varchar(30) NOT NULL,
"address" varchar(50) NOT NULL,
"city" varchar(60) NOT NULL,
"state_province" varchar(30) NOT NULL,
"country" varchar(50) NOT NULL,
"website" varchar(200) NOT NULL

);

CREATE TABLE "books_book__new" (
"id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
"title" varchar(100) NOT NULL,
"publication_date" date NOT NULL,
"publisher_id" integer NOT NULL REFERENCES
"books_publisher" ("id")

);

INSERT INTO "books_book__new" ("id", "publisher_id", "title",
"publication_date") SELECT "id", NULL, "title", "publication_date" FROM
"books_book";
DROP TABLE "books_book";
ALTER TABLE "books_book__new" RENAME TO "books_book";
CREATE INDEX "books_book_2604cbea" ON "books_book" ("publisher_id");
COMMIT;

```

Note the following:

- Table names are automatically generated by combining the name of the app (`books`) and the lowercase name of the model (`publisher`, `book`, and `author`). You can override this behavior, as detailed in Appendix B.

- As we mentioned earlier, Django adds a primary key for each table automatically – the `id` fields. You can override this, too.
- By convention, Django appends `"_id"` to the foreign key field name. As you might have guessed, you can override this behavior, too.
- The foreign key relationship is made explicit by a `REFERENCES` statement.
- These `CREATE TABLE` statements are tailored to the database you’re using, so database-specific field types such as `auto_increment` (MySQL), `serial` (PostgreSQL), or `integer primary key` (SQLite) are handled for you automatically. The same goes for quoting of column names (e.g., using double quotes or single quotes). This example output is in PostgreSQL syntax.

The `sqlmigrate` command doesn’t actually create the tables or otherwise touch your database – it just prints output to the screen so you can see what SQL Django would execute if you asked it. If you wanted to, you could copy and paste this SQL into your database client, however Django provides an easier way of committing the SQL to the database: the `migrate` command:

```
python manage.py migrate
```

Run that command, and you’ll see something like this:

Operations to perform:

```
  Apply all migrations: books
```

Running migrations:

```
  Rendering model states... DONE
```

```
# ...
```

```
  Applying books.0001_initial... OK
```

```
# ...
```

In case you were wondering what all the extras are (commented out above), the first time you run `migrate`, Django will also create all the system tables that Django needs for the inbuilt apps.

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, however there are some caveats. For more information on migrations, see [Chapter 23](#).

BASIC DATA ACCESS

Once you've created a model, Django automatically provides a high-level Python API for working with those models. Try it out by running `python manage.py shell` and typing the following:

```
>>> from books.models import Publisher  
  
>>> p1 = Publisher(name='Apress', address='2855 Telegraph Avenue',  
...     city='Berkeley', state_province='CA', country='U.S.A.',  
...     website='http://www.apress.com/')  
  
>>> p1.save()  
  
>>> p2 = Publisher(name="O'Reilly", address='10 Fawcett St.',  
...     city='Cambridge', state_province='MA', country='U.S.A.',  
...     website='http://www.oreilly.com/')  
  
>>> p2.save()  
  
>>> publisher_list = Publisher.objects.all()  
  
>>> publisher_list  
[, ]
```

These few lines of code accomplish quite a bit. Here are the highlights:

- First, we import our `Publisher` model class. This lets us interact with the database table that contains publishers.
- We create a `Publisher` object by instantiating it with values for each field – `name`, `address`, etc.
- To save the object to the database, call its `save()` method. Behind the scenes, Django executes an SQL `INSERT` statement here.
- To retrieve publishers from the database, use the attribute `Publisher.objects`, which you can think of as a set of all publishers. Fetch a list of *all* `Publisher` objects in the database with the statement `Publisher.objects.all()`. Behind the scenes, Django executes an SQL `SELECT` statement here.

One thing is worth mentioning, in case it wasn't clear from this example. When you're creating objects using the Django model API, Django doesn't save the objects to the database until you call the `save()` method:

```
p1 = Publisher(...)

# At this point, p1 is not saved to the database yet!

p1.save()

# Now it is.
```

If you want to create an object and save it to the database in a single step, use the `objects.create()` method. This example is equivalent to the example above:

```
>>> p1 = Publisher.objects.create(name='Apress',
...       address='2855 Telegraph Avenue',
...       city='Berkeley', state_province='CA', country='U.S.A.',
...       website='http://www.apress.com/')

>>> p2 = Publisher.objects.create(name="O'Reilly",
...       address='10 Fawcett St.', city='Cambridge',
...       state_province='MA', country='U.S.A.',
...       website='http://www.oreilly.com/')

>>> publisher_list = Publisher.objects.all()

>>> publisher_list
```

Naturally, you can do quite a lot with the Django database API – but first, let's take care of a small annoyance.

ADDING MODEL STRING REPRESENTATIONS

When we printed out the list of publishers, all we got was this unhelpful display that makes it difficult to tell the `Publisher` objects apart:

```
[<Publisher: Publisher object>, <Publisher: Publisher object>]
```

We can fix this easily by adding a method called `__str__()` to our `Publisher` class. A `__str__()` method tells Python how to display a human-readable representation of an object. You can see this in action by adding a `__str__()` method to the three models:

```

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()

    def __str__(self):
        return self.name


class Author(models.Model):

    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()

    def __str__(self):
        return u'%s %s' % (self.first_name, self.last_name)


class Book(models.Model):

    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()

    def __str__(self):
        return self.title

```

As you can see, a `__str__()` method can do whatever it needs to do in order to return a representation of an object. Here, the `__str__()` methods for `Publisher` and `Book` simply return the object's name and title, respectively, but the `__str__()` for `Author` is slightly more complex – it pieces together the `first_name` and `last_name` fields, separated by a space.

The only requirement for `__str__()` is that it return a string object. If `__str__()` doesn't return a string object – if it returns, say, an integer – then Python will raise a `TypeError` with a message like:

```
TypeError: __str__ returned non-string (type int).
```

For the `__str__()` changes to take effect, exit out of the Python shell and enter it again with `python manage.py shell`. (This is the simplest way to make code changes take effect.) Now the list of `Publisher` objects is much easier to understand:

```
>>> from books.models import Publisher  
>>> publisher_list = Publisher.objects.all()  
>>> publisher_list  
[<Publisher: Apress>, <Publisher: O'Reilly>]
```

Make sure any model you define has a `__str__()` method – not only for your own convenience when using the interactive interpreter, but also because Django uses the output of `__str__()` in several places when it needs to display objects.

Finally, note that `__str__()` is a good example of adding *behavior* to models. A Django model describes more than the database table layout for an object; it also describes any functionality that object knows how to do. `__str__()` is one example of such functionality – a model knows how to display itself.

INSERTING AND UPDATING DATA

You've already seen this done: to insert a row into your database, first create an instance of your model using keyword arguments, like so:

```
>>> p = Publisher(name='Apress',  
...                 address='2855 Telegraph Ave.',  
...                 city='Berkeley',  
...                 state_province='CA',
```

```
...         country='U.S.A.',  
...         website='http://www.apress.com/')
```

As we noted above, this act of instantiating a model class does *not* touch the database. The record isn't saved into the database until you call `save()`, like this:

```
>>> p.save()
```

In SQL, this can roughly be translated into the following:

```
INSERT INTO books_publisher  
(name, address, city, state_province, country, website)  
VALUES  
('Apress', '2855 Telegraph Ave.', 'Berkeley', 'CA',  
'U.S.A.', 'http://www.apress.com/');
```

Because the `Publisher` model uses an autoincrementing primary key `id`, the initial call to `save()` does one more thing: it calculates the primary key value for the record and sets it to the `id` attribute on the instance:

```
>>> p.id  
52    # this will differ based on your own data
```

Subsequent calls to `save()` will save the record in place, without creating a new record (i.e., performing an SQL `UPDATE` statement instead of an `INSERT`):

```
>>> p.name = 'Apress Publishing'  
>>> p.save()
```

The preceding `save()` statement will result in roughly the following SQL:

```
UPDATE books_publisher SET  
  name = 'Apress Publishing',  
  address = '2855 Telegraph Ave.',  
  city = 'Berkeley',  
  state_province = 'CA',  
  country = 'U.S.A.',  
  website = 'http://www.apress.com'
```

```
WHERE id = 52;
```

Yes, note that *all* of the fields will be updated, not just the ones that have been changed. Depending on your application, this may cause a race condition. See “Updating Multiple Objects in One Statement” below to find out how to execute this (slightly different) query:

```
UPDATE books_publisher SET  
    name = 'Apress Publishing'  
WHERE id=52;
```

SELECTING OBJECTS

Knowing how to create and update database records is essential, but chances are that the Web applications you’ll build will be doing more querying of existing objects than creating new ones. We’ve already seen a way to retrieve *every* record for a given model:

```
>>> Publisher.objects.all()  
[<Publisher: Apress>, <Publisher: O'Reilly>]
```

This roughly translates to this SQL:

```
SELECT id, name, address, city, state_province, country, website  
FROM books_publisher;
```

Note

Notice that Django doesn’t use `SELECT *` when looking up data and instead lists all fields explicitly. This is by design: in certain circumstances `SELECT *` can be slower, and (more important) listing fields more closely follows one tenet of the Zen of Python: “Explicit is better than implicit.”

For more on the Zen of Python, try typing `import this` at a Python prompt.

Let’s take a close look at each part of this `Publisher.objects.all()` line:

- First, we have the model we defined, `Publisher`. No surprise here: when you want to look up data, you use the model for that data.

- Next, we have the `objects` attribute. This is called a *manager*. Managers are discussed in detail in [Chapter 09](#). For now, all you need to know is that managers take care of all “table-level” operations on data including, most important, data lookup.

All models automatically get a `objects` manager; you’ll use it any time you want to look up model instances.

- Finally, we have `all()`. This is a method on the `objects` manager that returns all the rows in the database. Though this object *looks* like a list, it’s actually a *QuerySet* – an object that represents a specific set of rows from the database. [Appendix C](#) deals with *QuerySets* in detail. For the rest of this chapter, we’ll just treat them like the lists they emulate.

Any database lookup is going to follow this general pattern – we’ll call methods on the manager attached to the model we want to query against.

FILTERING DATA

Naturally, it’s rare to want to select *everything* from a database at once; in most cases, you’ll want to deal with a subset of your data. In the Django API, you can filter your data using the `filter()` method:

```
>>> Publisher.objects.filter(name='Apress')
[<Publisher: Apress>]
```

`filter()` takes keyword arguments that get translated into the appropriate SQL `WHERE` clauses. The preceding example would get translated into something like this:

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
WHERE name = 'Apress';
```

You can pass multiple arguments into `filter()` to narrow down things further:

```
>>> Publisher.objects.filter(country="U.S.A.", state_province="CA")
[<Publisher: Apress>]
```

Those multiple arguments get translated into SQL `AND` clauses. Thus, the example in the code snippet translates into the following:

```
SELECT id, name, address, city, state_province, country, website
```

```
FROM books_publisher  
WHERE country = 'U.S.A.'  
AND state_province = 'CA';
```

Notice that by default the lookups use the SQL `=` operator to do exact match lookups. Other lookup types are available:

```
>>> Publisher.objects.filter(name__contains="press")  
[<Publisher: Apress>]
```

That's a *double underscore* there between `name` and `contains`. Like Python itself, Django uses the double underscore to signal that something "magic" is happening – here, the `__contains` part gets translated by Django into a SQL `LIKE` statement:

```
SELECT id, name, address, city, state_province, country, website  
FROM books_publisher  
WHERE name LIKE '%press%';
```

Many other types of lookups are available, including `icontains` (case-insensitive `LIKE`), `startswith` and `endswith`, and `range` (SQL `BETWEEN` queries). [Appendix C](#) describes all of these lookup types in detail.

RETRIEVING SINGLE OBJECTS

The `filter()` examples above all returned a `QuerySet`, which you can treat like a list. Sometimes it's more convenient to fetch only a single object, as opposed to a list. That's what the `get()` method is for:

```
>>> Publisher.objects.get(name="Apress")  
<Publisher: Apress>
```

Instead of a list (rather, `QuerySet`), only a single object is returned. Because of that, a query resulting in multiple objects will cause an exception:

```
>>> Publisher.objects.get(country="U.S.A.")  
Traceback (most recent call last):  
...  

```

```
MultipleObjectsReturned: get() returned more than one Publisher --  
    it returned 2! Lookup parameters were {'country': 'U.S.A.'}
```

A query that returns no objects also causes an exception:

```
>>> Publisher.objects.get(name="Penguin")
```

```
Traceback (most recent call last):
```

```
...
```

```
DoesNotExist: Publisher matching query does not exist.
```

The `DoesNotExist` exception is an attribute of the model's class – `Publisher.DoesNotExist`. In your applications, you'll want to trap these exceptions, like this:

```
try:  
    p = Publisher.objects.get(name='Apress')  
  
except Publisher.DoesNotExist:  
  
    print ("Apress isn't in the database yet.")  
  
else:  
  
    print ("Apress is in the database.")
```

ORDERING DATA

As you play around with the previous examples, you might discover that the objects are being returned in a seemingly random order. You aren't imagining things; so far we haven't told the database how to order its results, so we're simply getting back data in some arbitrary order chosen by the database.

In your Django applications, you'll probably want to order your results according to a certain value – say, alphabetically. To do this, use the `order_by()` method:

```
>>> Publisher.objects.order_by("name")  
[<Publisher: Apress>, <Publisher: O'Reilly>]
```

This doesn't look much different from the earlier `all()` example, but the SQL now includes a specific ordering:

```
SELECT id, name, address, city, state_province, country, website
```

```
FROM books_publisher  
ORDER BY name;
```

You can order by any field you like:

```
>>> Publisher.objects.order_by("address")  
[<Publisher: O'Reilly>, <Publisher: Apress>]
```

```
>>> Publisher.objects.order_by("state_province")  
[<Publisher: Apress>, <Publisher: O'Reilly>]
```

To order by multiple fields (where the second field is used to disambiguate ordering in cases where the first is the same), use multiple arguments:

```
>>> Publisher.objects.order_by("state_province", "address")  
[<Publisher: Apress>, <Publisher: O'Reilly>]
```

You can also specify reverse ordering by prefixing the field name with a - (that's a minus character):

```
>>> Publisher.objects.order_by("-name")  
[<Publisher: O'Reilly>, <Publisher: Apress>]
```

While this flexibility is useful, using `order_by()` all the time can be quite repetitive. Most of the time you'll have a particular field you usually want to order by. In these cases, Django lets you specify a default ordering in the model:

```
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()  
  
  
    def __str__(self):
```

```
    return self.name

class Meta:
    ordering = ['name']
```

Here, we've introduced a new concept: the `Meta` class, which is a class that's embedded within the `Publisher` class definition (i.e., it's indented to be within `class Publisher`). You can use this `Meta` class on any model to specify various model-specific options. A full reference of `Meta` options is available in [Appendix B](#), but for now, we're concerned with the `ordering` option. If you specify this, it tells Django that unless an ordering is given explicitly with `order_by()`, all `Publisher` objects should be ordered by the `name` field whenever they're retrieved with the Django database API.

CHAINING LOOKUPS

You've seen how you can filter data, and you've seen how you can order it. Often, of course, you'll need to do both. In these cases, you simply "chain" the lookups together:

```
>>> Publisher.objects.filter(country="U.S.A.").order_by("-name")
[<Publisher: O'Reilly>, <Publisher: Apress>]
```

As you might expect, this translates to a SQL query with both a `WHERE` and an `ORDER BY`:

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
WHERE country = 'U.S.A'
ORDER BY name DESC;
```

SLICING DATA

Another common need is to look up only a fixed number of rows. Imagine you have thousands of publishers in your database, but you want to display only the first one. You can do this using Python's standard list slicing syntax:

```
>>> Publisher.objects.order_by('name')[0]
<Publisher: Apress>
```

This translates roughly to:

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
ORDER BY name
LIMIT 1;
```

Similarly, you can retrieve a specific subset of data using Python's range-slicing syntax:

```
>>> Publisher.objects.order_by('name')[0:2]
```

This returns two objects, translating roughly to:

```
SELECT id, name, address, city, state_province, country, website
FROM books_publisher
ORDER BY name
OFFSET 0 LIMIT 2;
```

Note that negative slicing is *not* supported:

```
>>> Publisher.objects.order_by('name')[-1]
```

```
Traceback (most recent call last):
```

```
...
```

```
AssertionError: Negative indexing is not supported.
```

This is easy to get around, though. Just change the `order_by()` statement, like this:

```
>>> Publisher.objects.order_by('-name')[0]
```

UPDATING MULTIPLE OBJECTS IN ONE STATEMENT

We pointed out in the “Inserting and Updating Data” section that the model `save()` method updates *all* columns in a row. Depending on your application, you may want to update only a subset of columns.

For example, let's say we want to update the Apress `Publisher` to change the name from 'Apress' to 'Apress Publishing'. Using `save()`, it would look something like this:

```
>>> p = Publisher.objects.get(name='Apress')
>>> p.name = 'Apress Publishing'
>>> p.save()
```

This roughly translates to the following SQL:

```
SELECT id, name, address, city, state_province, country, website  
FROM books_publisher  
WHERE name = 'Apress';  
  
UPDATE books_publisher SET  
    name = 'Apress Publishing',  
    address = '2855 Telegraph Ave.',  
    city = 'Berkeley',  
    state_province = 'CA',  
    country = 'U.S.A.',  
    website = 'http://www.apress.com'  
WHERE id = 52;
```

(Note that this example assumes Apress has a publisher ID of 52.)

You can see in this example that Django's `save()` method sets *all* of the column values, not just the `name` column. If you're in an environment where other columns of the database might change due to some other process, it's smarter to change *only* the column you need to change. To do this, use the `update()` method on `QuerySet` objects. Here's an example:

```
>>> Publisher.objects.filter(id=52).update(name='Apress Publishing')
```

The SQL translation here is much more efficient and has no chance of race conditions:

```
UPDATE books_publisher  
SET name = 'Apress Publishing'  
WHERE id = 52;
```

The `update()` method works on any `QuerySet`, which means you can edit multiple records in bulk. Here's how you might change the `country` from 'U.S.A.' to USA in each Publisher record:

```
>>> Publisher.objects.all().update(country='USA')
```

The `update()` method has a return value – an integer representing how many records changed. In the above example, we got 2.

DELETING OBJECTS

To delete an object from your database, simply call the object's `delete()` method:

```
>>> p = Publisher.objects.get(name="O'Reilly")
>>> p.delete()
>>> Publisher.objects.all()
[<Publisher: Apress Publishing>]
```

You can also delete objects in bulk by calling `delete()` on the result of any `QuerySet`. This is similar to the `update()` method we showed in the last section:

```
>>> Publisher.objects.filter(country='USA').delete()
>>> Publisher.objects.all().delete()
>>> Publisher.objects.all()
[]
```

Be careful deleting your data! As a precaution against deleting all of the data in a particular table, Django requires you to explicitly use `all()` if you want to delete *everything* in your table. For example, this won't work:

```
>>> Publisher.objects.delete()
Traceback (most recent call last):
  File "", line 1, in 
AttributeError: 'Manager' object has no attribute 'delete'
```

But it'll work if you add the `all()` method:

```
>>> Publisher.objects.all().delete()
```

If you're just deleting a subset of your data, you don't need to include `all()`. To repeat a previous example:

```
>>> Publisher.objects.filter(country='USA').delete()
```

WHAT'S NEXT?

Having read this chapter, you have enough knowledge of Django models to be able to write basic database applications. [Chapter 9](#) will provide some information on more advanced usage of Django's database layer.

Once you've defined your models, the next step is to populate your database with data. You might have legacy data, in which case [Chapter 23](#) will give you advice about integrating with legacy databases. You might rely on site users to supply your data, in which case [Chapter 6](#) will teach you how to process user-submitted form data.

But in some cases, you or your team might need to enter data manually, in which case it would be helpful to have a Web-based interface for entering and managing data. The next chapter covers [Django's admin interface](#), which exists precisely for that reason.

CHAPTER 5: THE DJANGO ADMIN SITE

For most modern Web sites, an *admin interface* is an essential part of the infrastructure.

This is a Web-based interface, limited to trusted site administrators, that enables the adding, editing and deletion of site content. Some common examples: the interface you use to post to your blog, the backend site managers use to moderate user-generated comments, the tool your clients use to update the press releases on the Web site you built for them.

There's a problem with admin interfaces, though: it's boring to build them. Web development is fun when you're developing public-facing functionality, but building admin interfaces is always the same. You have to authenticate users, display and handle forms, validate input, and so on. It's boring, and it's repetitive.

So what's Django's approach to these boring, repetitive tasks? It does it all for you. With Django, building an admin interface is a solved problem.

In this chapter we will be exploring Django's automatic admin interface: checking out how it provides a convenient interface to our models, and some of the other useful things we can do with it.

USING THE ADMIN SITE

When you ran `django-admin startproject mysite` in [Chapter 1](#), Django created and configured the default admin site for you. All that you need to do is create an admin user (superuser) and then you can log into the admin site.

To create an admin user, 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.

Recall from Tutorial 1 that you start the development server 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:

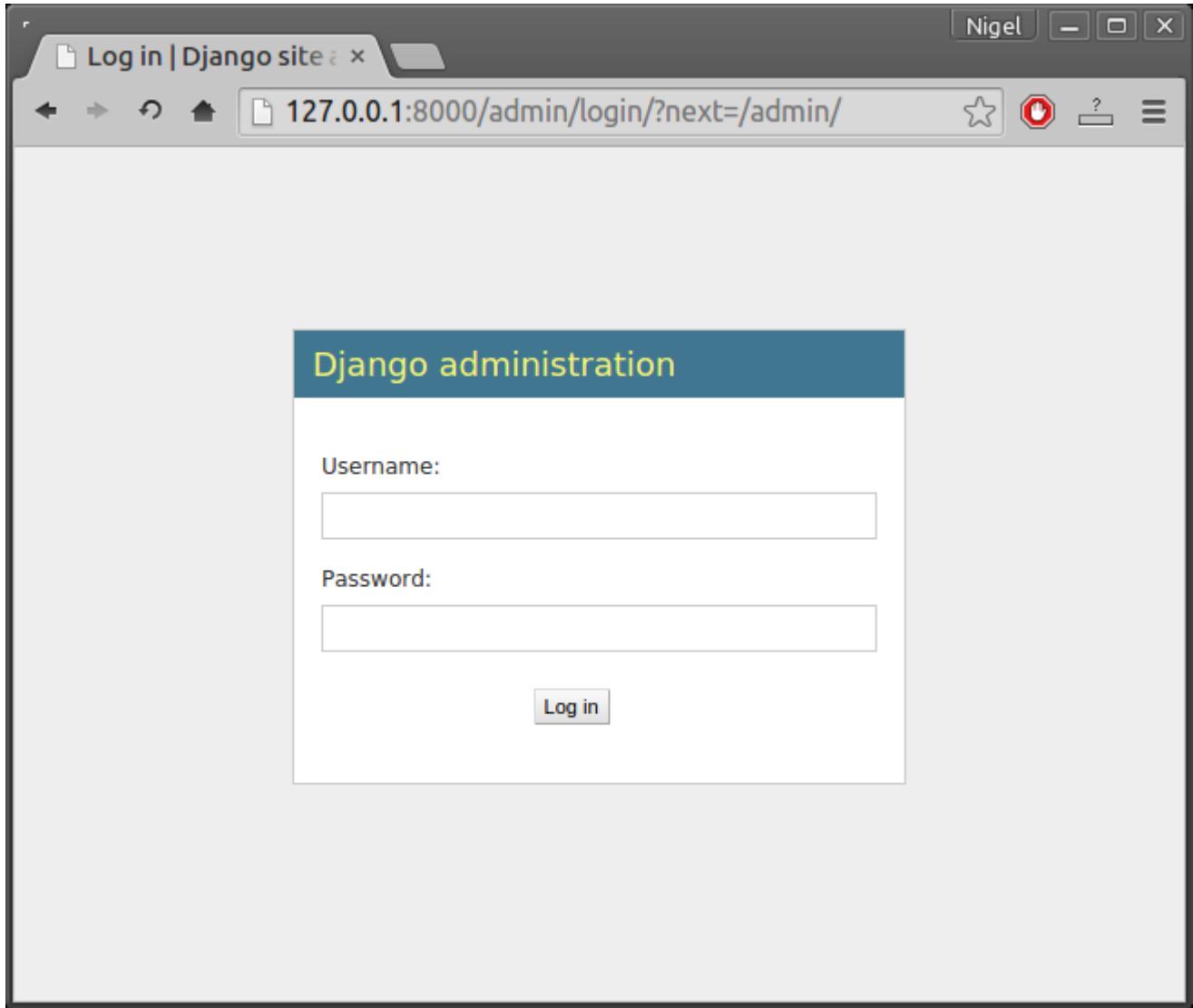


Figure 5-1: Django admin login screen

Since translation is turned on by default, the login screen may be displayed in your own language, depending on your browser’s settings and on whether 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:

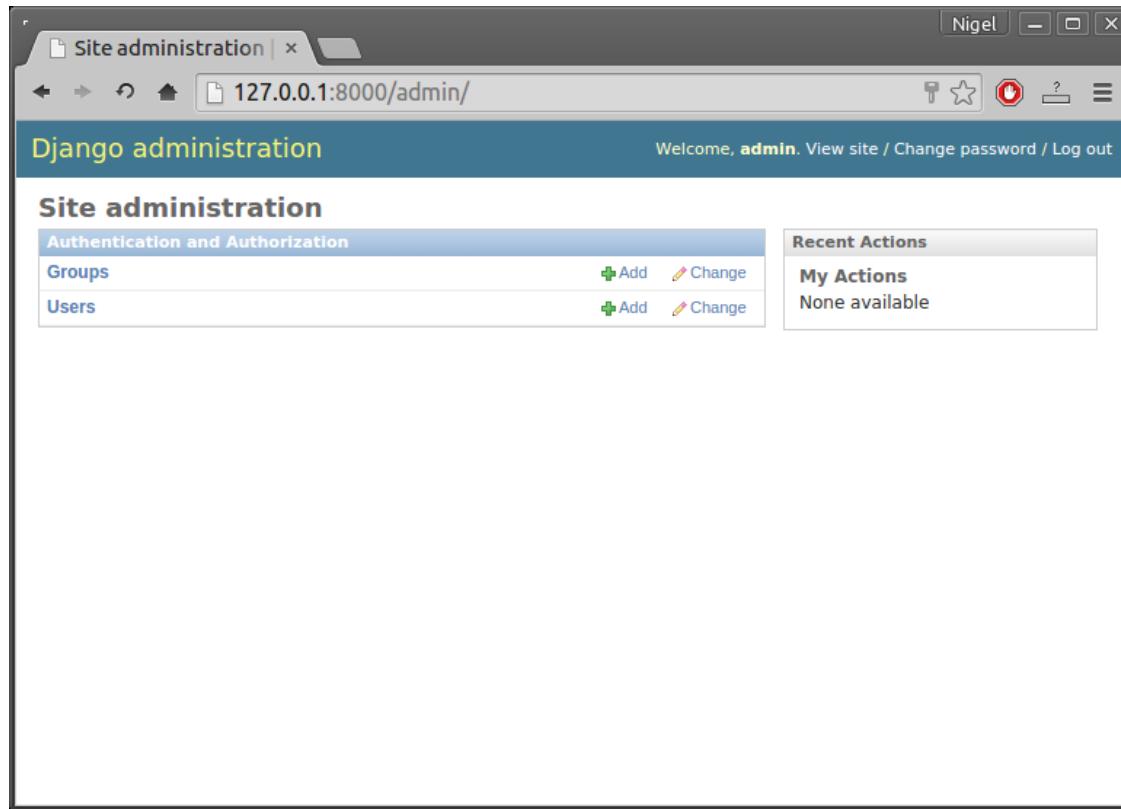


Figure 5-2: Django admin home page

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.

The admin site is designed to be used by nontechnical users, and as such it should be pretty self-explanatory. Nevertheless, we'll give you a quick walkthrough of the basic features.

Each type of data in the Django admin site has a *change list* and an *edit form*. Change lists show you all the available objects in the database, and edit forms let you add, change or delete particular records in your database.

Click the “Change” link in the “Users” row to load the change list page for users.

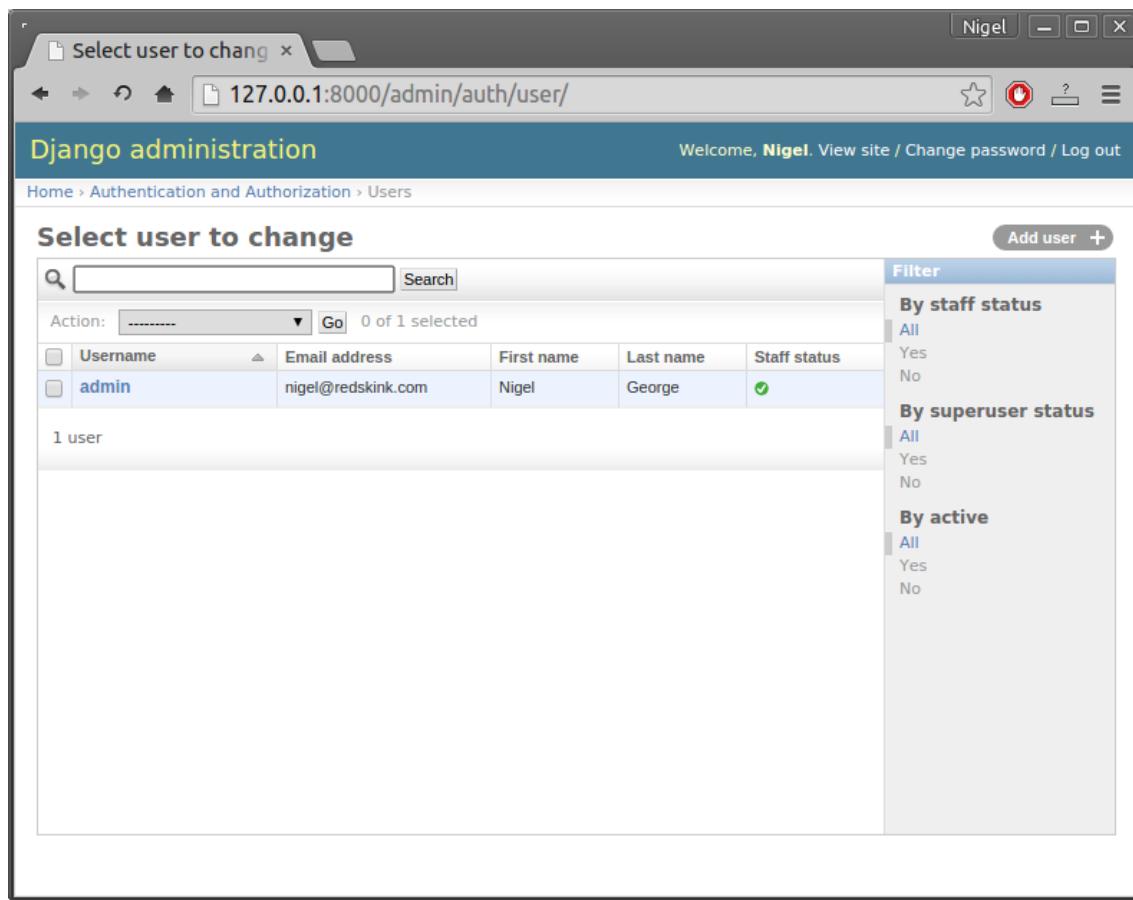


Figure 5-3. The user change list page

This page displays all users in the database; you can think of it as a prettied-up Web version of a `SELECT * FROM auth_user;` SQL query. If you're following along with our ongoing example, you'll only see one user here, assuming you've added only one, but once you have more users, you'll probably find the filtering, sorting and searching options useful. Filtering options are at right, sorting is available by clicking a column header, and the search box at the top lets you search by username.

Click the username of the user you created, and you'll see the edit form for that user.

Django administration

Welcome, **Nigel**. View site / Change password / Log out

Home > Authentication and Authorization > Users > admin

Change user

History

Username:	<input type="text" value="admin"/>	Required. 30 characters or fewer. Letters, digits and @/_/+/-/_ only.
Password:	algorithm: pbkdf2_sha256 iterations: 20000 salt: cZYq6v***** hash: eFNqwY***** Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using this form .	
Personal info		
First name:	<input type="text" value="Nigel"/>	
Last name:	<input type="text" value="George"/>	

Figure 5-4. The user edit form

This page lets you change the attributes of the user, like the first/last names and various permissions. (Note that to change a user's password, you should click "change password form" under the password field rather than editing the hashed code.) Another thing to note here is that fields of different types get different widgets – for example, date/time fields have calendar controls, boolean fields have checkboxes, character fields have simple text input fields.

You can delete a record by clicking the delete button at the bottom left of its edit form. That'll take you to a confirmation page, which, in some cases, will display any dependent objects that will be deleted, too. (For example, if you delete a publisher, any book with that publisher will be deleted, too!)

You can add a record by clicking "Add" in the appropriate column of the admin home page. This will give you an empty version of the edit page, ready for you to fill out.

You'll also notice that the admin interface also handles input validation for you. Try leaving a required field blank or putting an invalid date into a date field, and you'll see those errors when you try to save, as shown in Figure 5-5.

The screenshot shows a web browser window titled "Add user | Django site". The URL in the address bar is "127.0.0.1:8000/admin/auth/user/add/". The page header says "Django administration" and "Welcome, admin. View site / Change password / Log out". Below the header, the breadcrumb navigation shows "Home > Authentication and Authorization > Users > Add user". The main content area is titled "Add user" and contains a message: "First, enter a username and password. Then, you'll be able to edit more user options.". A red error box at the top left says "Please correct the errors below." Three fields are highlighted with red error bars: "Username" (empty), "Password" (empty), and "Password confirmation" (empty). Each field has a descriptive error message below it: "This field is required.", "Required. 30 characters or fewer. Letters, digits and @/./+/_- only.", and "Enter the same password as above, for verification.". At the bottom right of the form are three buttons: "Save and add another", "Save and continue editing", and a blue "Save" button.

Figure 5-5. An edit form displaying errors

When you edit an existing object, you'll notice a History link in the upper-right corner of the window. Every change made through the admin interface is logged, and you can examine this log by clicking the History link (see Figure 5-6).

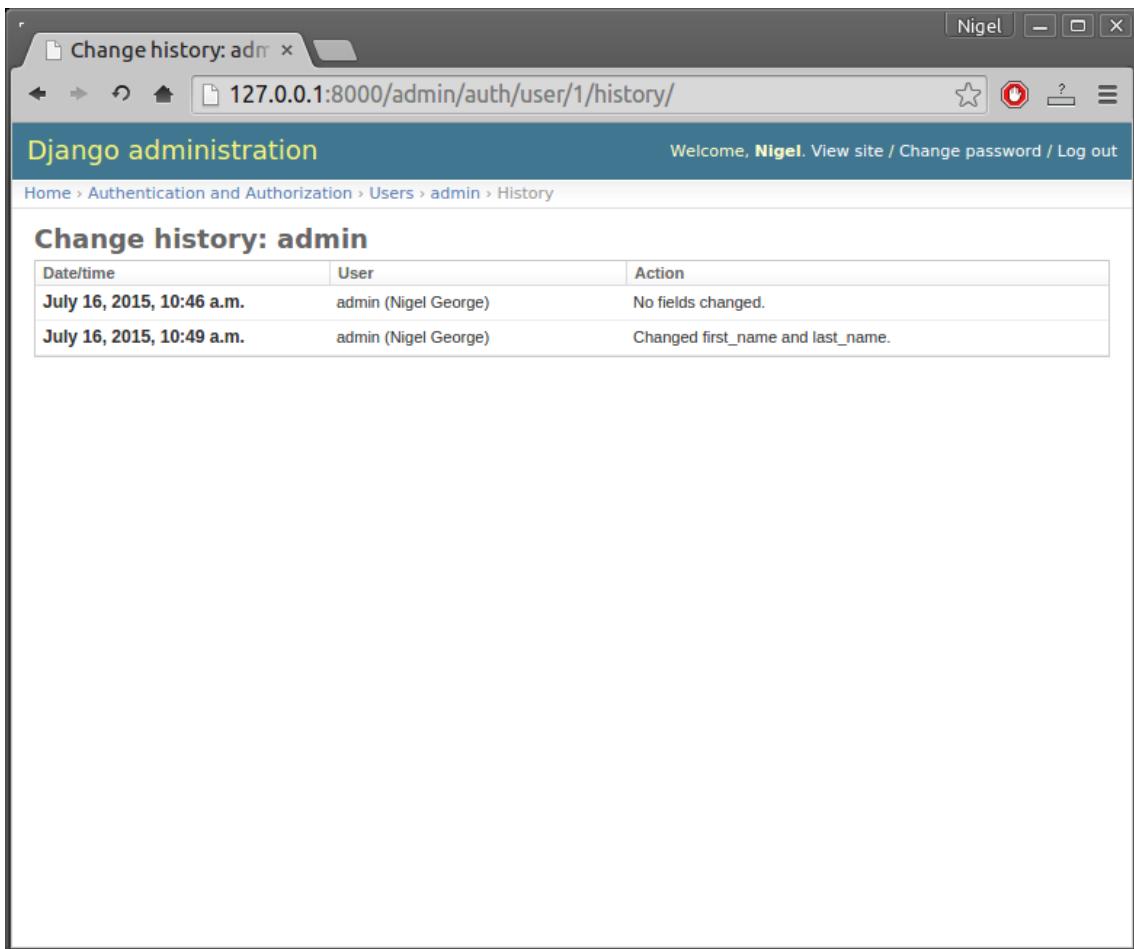


Figure 5-6. An object history page

ADDING YOUR MODELS TO THE ADMIN SITE

There's one crucial part we haven't done yet. Let's add our own models to the admin site, so we can add, change and delete objects in our custom database tables using this nice interface. We'll continue the `books` example from [Chapter 4](#), where we defined three models: `Publisher`, `Author` and `Book`.

Within the `books` directory (`mysite/books`), `startapp` should have created a file called `admin.py`, if not, simply create one yourself and type in the following lines of code:

```
from django.contrib import admin  
  
from .models import Publisher, Author, Book  
  
admin.site.register(Publisher)
```

```
admin.site.register(Author)  
admin.site.register(Book)
```

This code tells the Django admin site to offer an interface for each of these models.

Once you've done this, go to your admin home page in your Web browser (<http://127.0.0.1:8000/admin/>), and you should see a "Books" section with links for Authors, Books and Publishers. (You might have to stop and start the `runserver` for the changes to take effect.)

You now have a fully functional admin interface for each of those three models. That was easy!

Take some time to add and change records, to populate your database with some data. If you followed Chapter 5's examples of creating `Publisher` objects (and you didn't delete them), you'll already see those records on the publisher change list page.

One feature worth mentioning here is the admin site's handling of foreign keys and many-to-many relationships, both of which appear in the `Book` model. As a reminder, here's what the `Book` model looks like:

```
class Book(models.Model):  
  
    title = models.CharField(max_length=100)  
  
    authors = models.ManyToManyField(Author)  
  
    publisher = models.ForeignKey(Publisher)  
  
    publication_date = models.DateField()  
  
  
    def __str__(self):  
        return self.title
```

On the Django admin site's "Add book" page (<http://127.0.0.1:8000/admin/books/book/add/>), the publisher (a `ForeignKey`) is represented by a select box, and the authors field (a `ManyToManyField`) is represented by a multiple-select box. Both fields sit next to a green plus sign icon that lets you add related records of that type. For example, if you click the green plus sign next to the "Publisher" field, you'll get a pop-up window that lets you add a publisher. After you successfully create the publisher in the pop-up, the "Add book" form will be updated with the newly created publisher. Slick.

HOW THE ADMIN SITE WORKS

Behind the scenes, how does the admin site work? It's pretty straightforward.

When Django loads your URLconf from `urls.py` at server startup, it executes the `admin.autodiscover()` statement that we added as part of activating the admin. This function iterates over your `INSTALLED_APPS` setting and looks for a file called `admin.py` in each installed app. If an `admin.py` exists in a given app, it executes the code in that file.

In the `admin.py` in our `books` app, each call to `admin.site.register()` simply registers the given model with the admin. The admin site will only display an edit/change interface for models that have been explicitly registered.

The app `django.contrib.auth` includes its own `admin.py`, which is why Users and Groups showed up automatically in the admin. Other `django.contrib` apps, such as `django.contrib.redirects`, also add themselves to the admin, as do many third-party Django applications you might download from the Web.

Beyond that, the Django admin site is just a Django application, with its own models, templates, views and URLpatterns. You add it to your application by hooking it into your URLconf, just as you hook in your own views. You can inspect its templates, views and URLpatterns by poking around in `django/contrib/admin` in your copy of the Django codebase – but don't be tempted to change anything directly in there, as there are plenty of hooks for you to customize the way the admin site works. (If you do decide to poke around the Django admin application, keep in mind it does some rather complicated things in reading metadata about models, so it would probably take a good amount of time to read and understand the code.)

MAKING FIELDS OPTIONAL

After you play around with the admin site for a while, you'll probably notice a limitation – the edit forms require every field to be filled out, whereas in many cases you'd want certain fields to be optional. Let's say, for example, that we want our `Author` model's `email` field to be optional – that is, a blank string should be allowed. In the real world, you might not have an e-mail address on file for every author.

To specify that the `email` field is optional, edit the `Author` model (which, as you'll recall from [Chapter 4](#), lives in `mysite/books/models.py`). Simply add `blank=True` to the `email` field, like so:

```
class Author(models.Model):  
    first_name = models.CharField(max_length=30)  
    last_name = models.CharField(max_length=40)  
    email = models.EmailField(blank=True)
```

This tells Django that a blank value is indeed allowed for authors' e-mail addresses. By default, all fields have `blank=False`, which means blank values are not allowed.

There's something interesting happening here. Until now, with the exception of the `__str__()` method, our models have served as definitions of our database tables – Pythonic expressions of SQL `CREATE TABLE` statements, essentially. In adding `blank=True`, we have begun expanding our model beyond a simple definition of what the database table looks like. Now, our model class is starting to become a richer collection of knowledge about what `Author` objects are and what they can do. Not only is the `email` field represented by a `VARCHAR` column in the database; it's also an optional field in contexts such as the Django admin site.

Once you've added that `blank=True`, reload the "Add author" edit form (<http://127.0.0.1:8000/admin/books/author/add/>), and you'll notice the field's label – "Email" – is no longer bolded. This signifies it's not a required field. You can now add authors without needing to provide e-mail addresses; you won't get the loud red "This field is required" message anymore, if the field is submitted empty.

MAKING DATE AND NUMERIC FIELDS OPTIONAL

A common gotcha related to `blank=True` has to do with date and numeric fields, but it requires a fair amount of background explanation.

SQL has its own way of specifying blank values – a special value called `NULL`. `NULL` could mean "unknown," or "invalid," or some other application-specific meaning.

In SQL, a value of `NULL` is different than an empty string, just as the special Python object `None` is different than an empty Python string (""). This means it's possible for a particular character field (e.g., a `VARCHAR` column) to contain both `NULL` values and empty string values.

This can cause unwanted ambiguity and confusion: "Why does this record have a `NULL` but this other one has an empty string? Is there a difference, or was the data just entered inconsistently?" And: "How do I get all the records that have a blank value – should I look for both `NULL` records and empty strings, or do I only select the ones with empty strings?"

To help avoid such ambiguity, Django's automatically generated `CREATE TABLE` statements (which were covered in [Chapter 4](#)) add an explicit `NOT NULL` to each column definition. For example, here's the generated statement for our `Author` model, from Chapter 4:

```
CREATE TABLE "books_author" (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(40) NOT NULL,
    "email" varchar(75) NOT NULL
);
```

In most cases, this default behavior is optimal for your application and will save you from data-inconsistency headaches. And it works nicely with the rest of Django, such as the Django admin site, which inserts an empty string (*not* a `NULL` value) when you leave a character field blank.

But there's an exception with database column types that do not accept empty strings as valid values – such as dates, times and numbers. If you try to insert an empty string into a date or integer column, you'll likely get a database error, depending on which database you're using. (PostgreSQL, which is strict, will raise an exception here; MySQL might accept it or might not, depending on the version you're using, the time of day and the phase of the moon.) In this case, `NULL` is the only way to specify an empty value. In Django models, you can specify that `NULL` is allowed by adding `null=True` to a field.

So that's a long way of saying this: if you want to allow blank values in a date field (e.g., `DateField`, `TimeField`, `DateTimeField`) or numeric field (e.g., `IntegerField`, `DecimalField`, `FloatField`), you'll need to use both `null=True` *and* `blank=True`.

For sake of example, let's change our `Book` model to allow a blank `publication_date`. Here's the revised code:

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField(blank=True, null=True)
```

Adding `null=True` is more complicated than adding `blank=True`, because `null=True` changes the semantics of the database – that is, it changes the `CREATE TABLE` statement to remove the `NOT NULL` from the `publication_date` field. To complete this change, we'll need to update the database.

For a number of reasons, Django does not attempt to automate changes to database schemas, so it's your own responsibility to execute the `python manage.py migrate` command whenever you make such a change to a model.

Bringing this back to the admin site, now the “Add book” edit form should allow for empty publication date values.

CUSTOMIZING FIELD LABELS

On the admin site's edit forms, each field's label is generated from its model field name. The algorithm is simple: Django just replaces underscores with spaces and capitalizes the first character, so, for example, the `Book` model's `publication_date` field has the label “Publication date.”

However, field names don't always lend themselves to nice admin field labels, so in some cases you might want to customize a label. You can do this by specifying `verbose_name` in the appropriate model field.

For example, here's how we can change the label of the `Author.email` field to “e-mail,” with a hyphen:

```
class Author(models.Model):  
    first_name = models.CharField(max_length=30)  
    last_name = models.CharField(max_length=40)  
    email = models.EmailField(blank=True, verbose_name='e-mail')
```

Make that change and reload the server, and you should see the field's new label on the author edit form.

Note that you shouldn't capitalize the first letter of a `verbose_name` unless it should *always* be capitalized (e.g., “USA state”). Django will automatically capitalize it when it needs to, and it will use the exact `verbose_name` value in other places that don't require capitalization.

CUSTOM MODELADMIN CLASSES

The changes we've made so far – `blank=True`, `null=True` and `verbose_name` – are really model-level changes, not admin-level changes. That is, these changes are fundamentally a part of the model and just so happen to be used by the admin site; there's nothing admin-specific about them.

Beyond these, the Django admin site offers a wealth of options that let you customize how the admin site works for a particular model. Such options live in *ModelAdmin classes*, which are classes that contain configuration for a specific model in a specific admin site instance.

CUSTOMIZING CHANGE LISTS

Let's dive into admin customization by specifying the fields that are displayed on the change list for our `Author` model. By default, the change list displays the result of `__str__()` for each object. In [Chapter 4](#), we defined the `__str__()` method for `Author` objects to display the first name and last name together:

```
class Author(models.Model):  
    first_name = models.CharField(max_length=30)  
    last_name = models.CharField(max_length=40)  
    email = models.EmailField(blank=True, verbose_name='e-mail')  
  
    def __str__(self):  
        return u'%s %s' % (self.first_name, self.last_name)
```

As a result, the change list for `Author` objects displays each other's first name and last name together, as you can see in Figure 5-7.

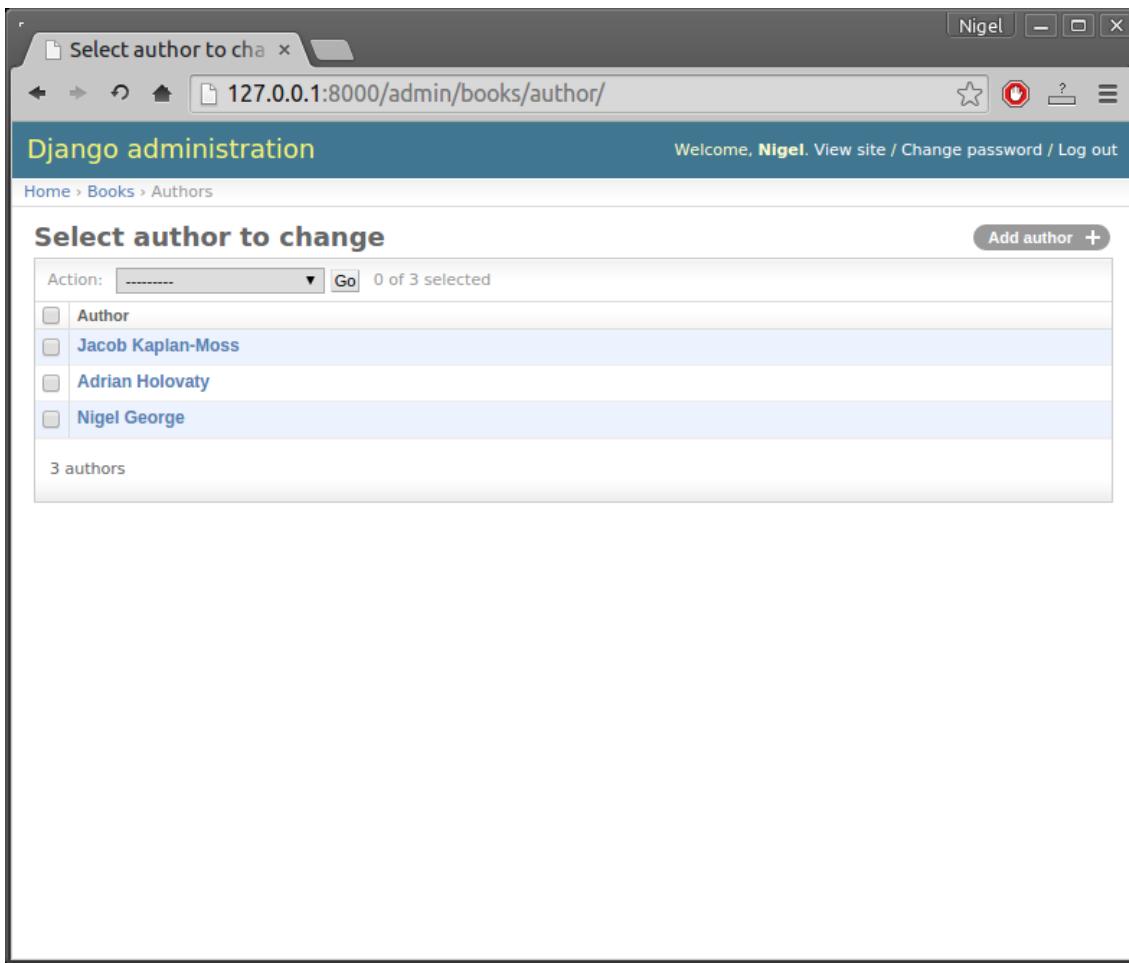


Figure 5-7. The author change list page

We can improve on this default behavior by adding a few other fields to the change list display. It'd be handy, for example, to see each author's e-mail address in this list, and it'd be nice to be able to sort by first and last name.

To make this happen, we'll define a `ModelAdmin` class for the `Author` model. This class is the key to customizing the admin, and one of the most basic things it lets you do is specify the list of fields to display on change list pages. Edit `admin.py` to make these changes:

```
from django.contrib import admin

from mysite.books.models import Publisher, Author, Book

class AuthorAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'email')
```

```
admin.site.register(Publisher)  
admin.site.register(Author, AuthorAdmin)  
admin.site.register(Book)
```

Here's what we've done:

- We created the class `AuthorAdmin`. This class, which subclasses `django.contrib.admin.ModelAdmin`, holds custom configuration for a specific admin model. We've only specified one customization – `list_display`, which is set to a tuple of field names to display on the change list page. These field names must exist in the model, of course.
- We altered the `admin.site.register()` call to add `AuthorAdmin` after `Author`. You can read this as: "Register the `Author` model with the `AuthorAdmin` options."

The `admin.site.register()` function takes a `ModelAdmin` subclass as an optional second argument. If you don't specify a second argument (as is the case for `Publisher` and `Book`), Django will use the default admin options for that model.

With that tweak made, reload the author change list page, and you'll see it's now displaying three columns – the first name, last name and e-mail address. In addition, each of those columns is sortable by clicking on the column header. (See Figure 5-8.)

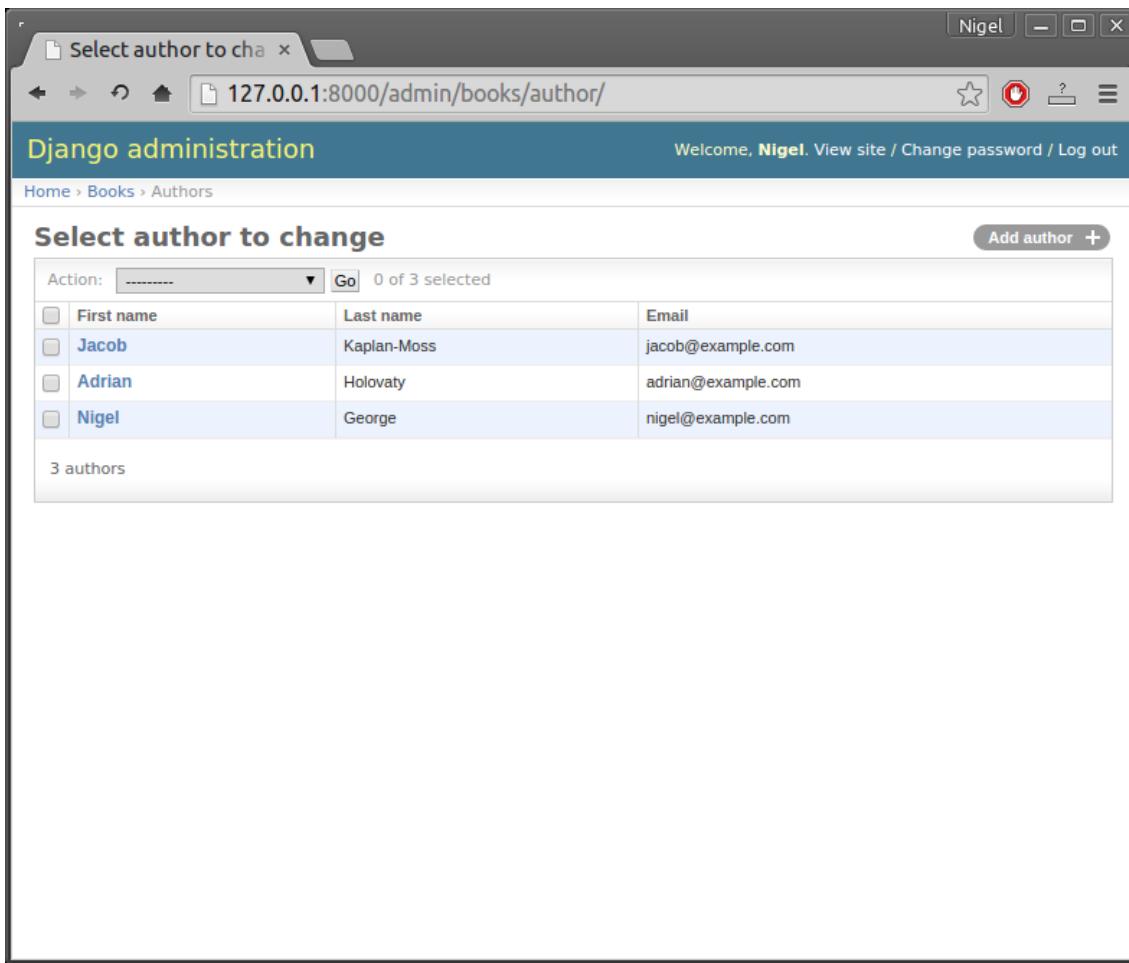


Figure 5-8. The author change list page after list_display

Next, let's add a simple search bar. Add `search_fields` to the `AuthorAdmin`, like so:

```
class AuthorAdmin(admin.ModelAdmin):  
    list_display = ('first_name', 'last_name', 'email')  
    search_fields = ('first_name', 'last_name')
```

Reload the page in your browser, and you should see a search bar at the top. (See Figure 5-9.) We've just told the admin change list page to include a search bar that searches against the `first_name` and `last_name` fields. As a user might expect, this is case-insensitive and searches both fields, so searching for the string "bar" would find both an author with the first name Barney and an author with the last name Hobarson.

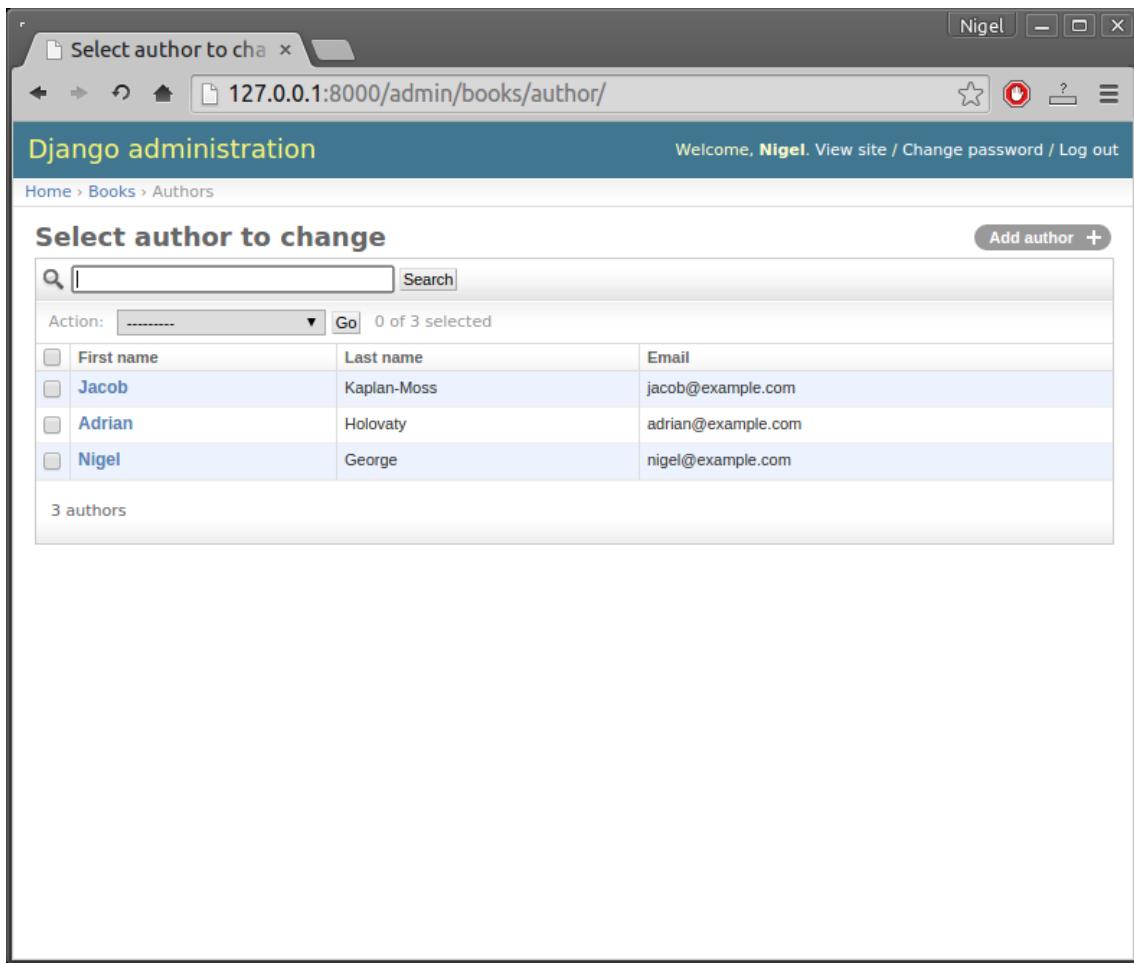


Figure 5-9. The author change list page after `search_fields`

Next, let's add some date filters to our `Book` model's change list page:

```
from django.contrib import admin

from mysite.books.models import Publisher, Author, Book


class AuthorAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'email')
    search_fields = ('first_name', 'last_name')


class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'publisher', 'publication_date')
```

```

list_filter = ('publication_date',)

admin.site.register(Publisher)
admin.site.register(Author, AuthorAdmin)
admin.site.register(Book, BookAdmin)

```

Here, because we're dealing with a different set of options, we created a separate `ModelAdmin` class – `BookAdmin`. First, we defined a `list_display` just to make the change list look a bit nicer. Then, we used `list_filter`, which is set to a tuple of fields to use to create filters along the right side of the change list page. For date fields, Django provides shortcuts to filter the list to “Today,” “Past 7 days,” “This month” and “This year” – shortcuts that Django’s developers have found hit the common cases for filtering by date. Figure 5-10 shows what that looks like.

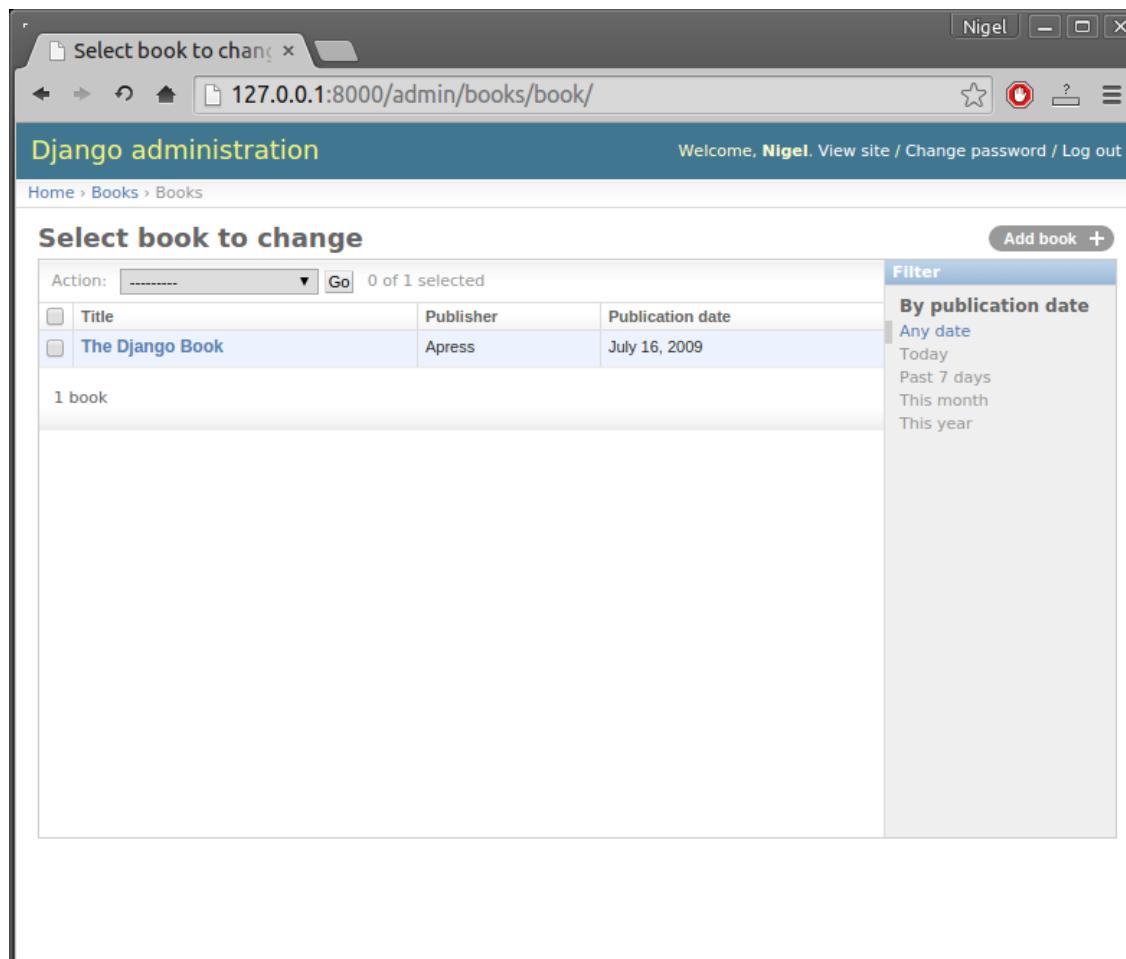


Figure 5-10. The book change list page after list_filter

`list_filter` also works on fields of other types, not just `DateField`. (Try it with `BooleanField` and `ForeignKey` fields, for example.) The filters show up as long as there are at least 2 values to choose from.

Another way to offer date filters is to use the `date_hierarchy` admin option, like this:

```
class BookAdmin(admin.ModelAdmin):  
    list_display = ('title', 'publisher', 'publication_date')  
    list_filter = ('publication_date',)  
    date_hierarchy = 'publication_date'
```

With this in place, the change list page gets a date drill-down navigation bar at the top of the list, as shown in Figure 5-11. It starts with a list of available years, then drills down into months and individual days.

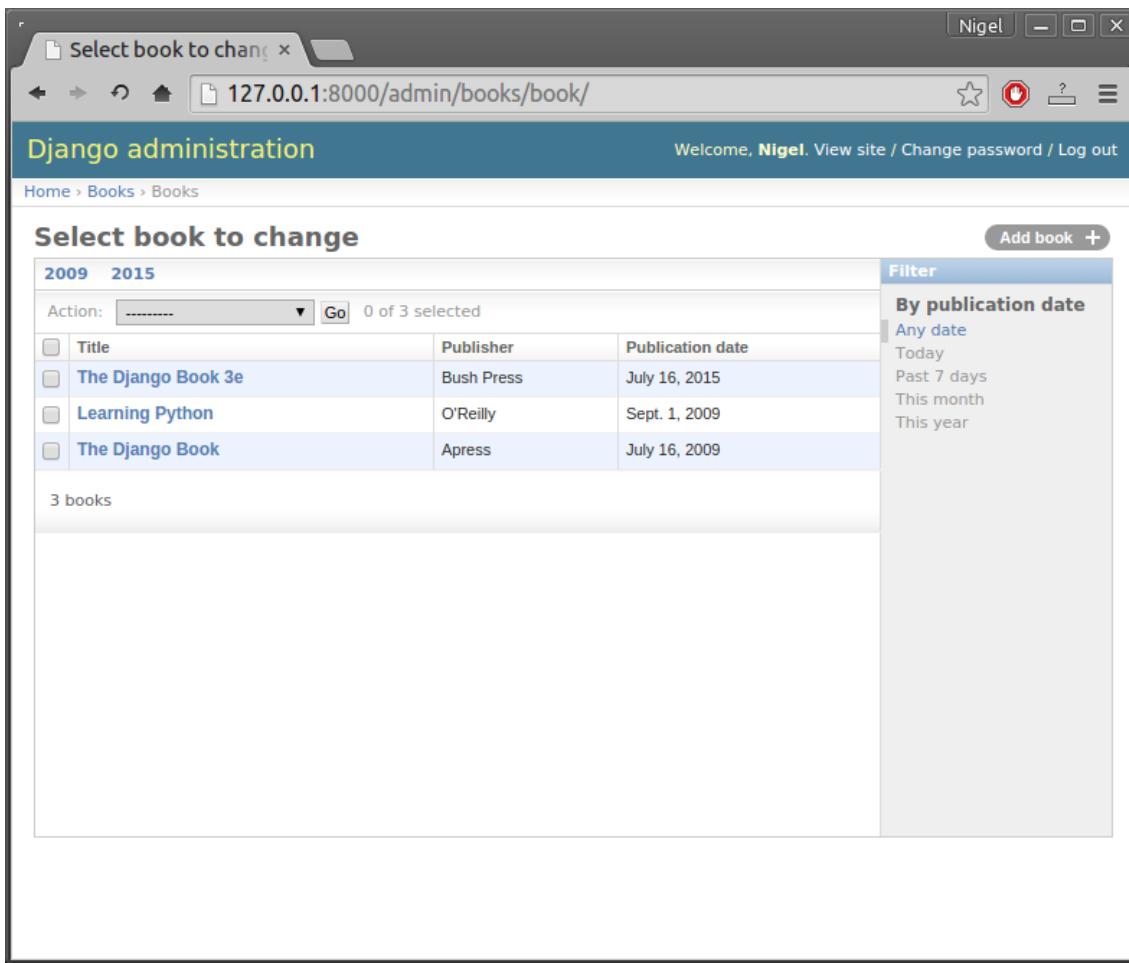


Figure 5-11. The book change list page after `date_hierarchy`

Note that `date_hierarchy` takes a *string*, not a tuple, because only one date field can be used to make the hierarchy.

Finally, let's change the default ordering so that books on the change list page are always ordered descending by their publication date. By default, the change list orders objects according to their model's `ordering` within `class Meta` (which we covered in [Chapter 4](#)) – but you haven't specified this `ordering` value, then the ordering is undefined.

```
class BookAdmin(admin.ModelAdmin):

    list_display = ('title', 'publisher', 'publication_date')

    list_filter = ('publication_date',)

    date_hierarchy = 'publication_date'

    ordering = ('-publication_date',)
```

This admin ordering option works exactly as the ordering in models' class Meta, except that it only uses the first field name in the list. Just pass a list or tuple of field names, and add a minus sign to a field to use descending sort order.

Reload the book change list to see this in action. Note that the “Publication date” header now includes a small arrow that indicates which way the records are sorted. (See Figure 5-12.)

The screenshot shows a web browser window titled "Select book to change" at the URL "127.0.0.1:8000/admin/books/book/". The browser title bar says "Nigel". The main content area is the "Django administration" interface. At the top left is the breadcrumb "Home > Books > Books". On the right are links to "View site / Change password / Log out". Below this is a "Select book to change" heading with a "Add book +" button. A "Filter" sidebar on the right is expanded, showing a dropdown menu "By publication date" with options: "Any date", "Today", "Past 7 days", "This month", and "This year". The main table lists three books:

Action	Title	Publisher	Publication date
<input type="checkbox"/>	The Django Book 3e	Bush Press	July 16, 2015
<input type="checkbox"/>	Learning Python	O'Reilly	Sept. 1, 2009
<input type="checkbox"/>	The Django Book	Apress	July 16, 2009

Below the table, it says "3 books". The table has a header row with columns for Action, Title, Publisher, and Publication date. The "Action" column contains checkboxes for each row. The "Title" column lists the book titles. The "Publisher" column lists the publishers. The "Publication date" column lists the publication dates. The table has a header row with columns for Action, Title, Publisher, and Publication date. The "Action" column contains checkboxes for each row. The "Title" column lists the book titles. The "Publisher" column lists the publishers. The "Publication date" column lists the publication dates.

Figure 5-12. The book change list page after ordering

We've covered the main change list options here. Using these options, you can make a very powerful, production-ready data-editing interface with only a few lines of code.

CUSTOMIZING EDIT FORMS

Just as the change list can be customized, edit forms can be customized in many ways.

First, let's customize the way fields are ordered. By default, the order of fields in an edit form corresponds to the order they're defined in the model. We can change that using the `fields` option in our `ModelAdmin` subclass:

```
class BookAdmin(admin.ModelAdmin):  
  
    list_display = ('title', 'publisher', 'publication_date')  
  
    list_filter = ('publication_date',)  
  
    date_hierarchy = 'publication_date'  
  
    ordering = ('-publication_date',)  
  
    fields = ('title', 'authors', 'publisher', 'publication_date')
```

After this change, the edit form for books will use the given ordering for fields. It's slightly more natural to have the authors after the book title. Of course, the field order should depend on your data-entry workflow. Every form is different.

Another useful thing the `fields` option lets you do is to *exclude* certain fields from being edited entirely. Just leave out the field(s) you want to exclude. You might use this if your admin users are only trusted to edit a certain segment of your data, or if part of your fields are changed by some outside, automated process. For example, in our book database, we could hide the `publication_date` field from being editable:

```
class BookAdmin(admin.ModelAdmin):  
  
    list_display = ('title', 'publisher', 'publication_date')  
  
    list_filter = ('publication_date',)  
  
    date_hierarchy = 'publication_date'  
  
    ordering = ('-publication_date',)  
  
    fields = ('title', 'authors', 'publisher')
```

As a result, the edit form for books doesn't offer a way to specify the publication date. This could be useful, say, if you're an editor who prefers that his authors not push back publication dates. (This is purely a hypothetical example, of course.)

When a user uses this incomplete form to add a new book, Django will simply set the `publication_date` to `None` – so make sure that field has `null=True`.

Another commonly used edit-form customization has to do with many-to-many fields. As we've seen on the edit form for books, the admin site represents each `ManyToManyField` as

a multiple-select boxes, which is the most logical HTML input widget to use – but multiple-select boxes can be difficult to use. If you want to select multiple items, you have to hold down the control key, or command on a Mac, to do so. The admin site helpfully inserts a bit of text that explains this, but, still, it gets unwieldy when your field contains hundreds of options.

The admin site's solution is `filter_horizontal`. Let's add that to `BookAdmin` and see what it does.

```
class BookAdmin(admin.ModelAdmin):  
  
    list_display = ('title', 'publisher', 'publication_date')  
  
    list_filter = ('publication_date',)  
  
    date_hierarchy = 'publication_date'  
  
    ordering = ('-publication_date',)  
  
    filter_horizontal = ('authors',)
```

(If you're following along, note that we've also removed the `fields` option to restore all the fields in the edit form.)

Reload the edit form for books, and you'll see that the "Authors" section now uses a fancy JavaScript filter interface that lets you search through the options dynamically and move specific authors from "Available authors" to the "Chosen authors" box, and vice versa.

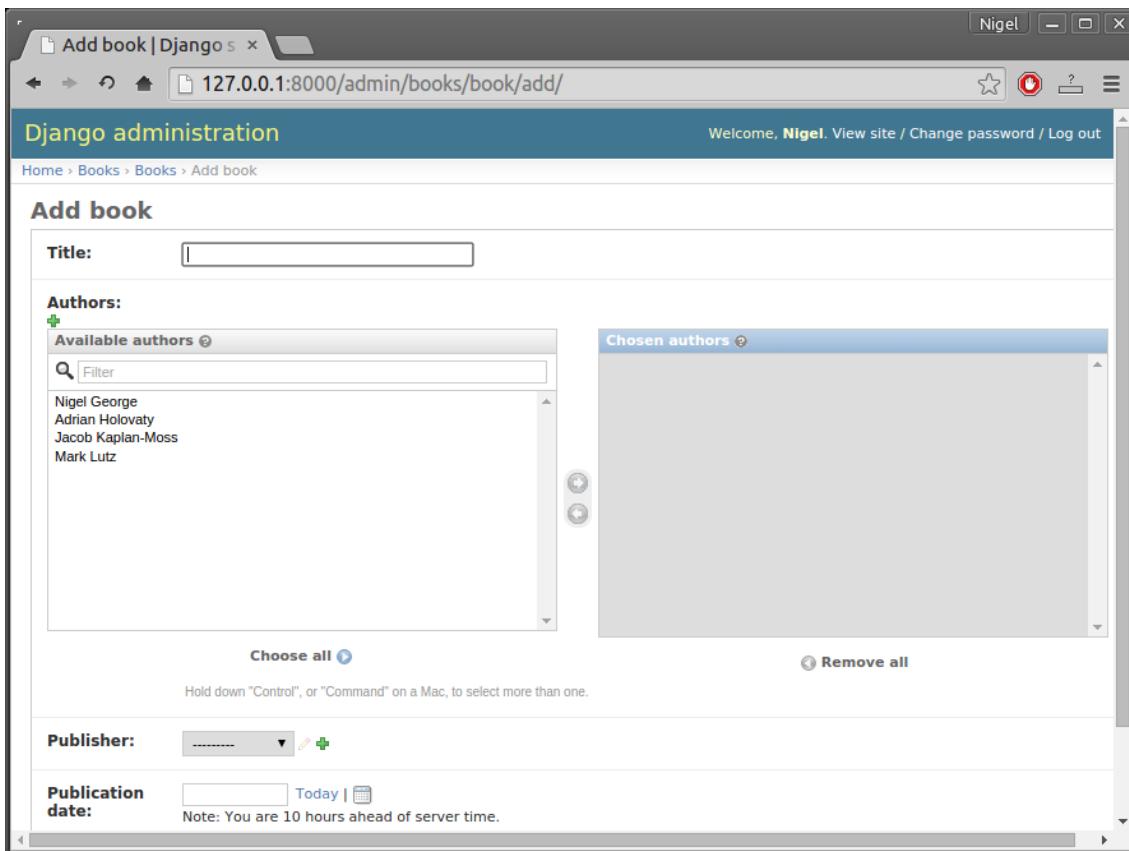


Figure 5-13. The book edit form after adding filter_horizontal

We'd highly recommend using `filter_horizontal` for any `ManyToManyField` that has more than 10 items. It's far easier to use than a simple multiple-select widget. Also, note you can use `filter_horizontal` for multiple fields – just specify each name in the tuple.

`ModelAdmin` classes also support a `filter_vertical` option. This works exactly as `filter_horizontal`, but the resulting JavaScript interface stacks the two boxes vertically instead of horizontally. It's a matter of personal taste.

`filter_horizontal` and `filter_vertical` only work on `ManyToManyField` fields, not `ForeignKey` fields. By default, the admin site uses simple `<select>` boxes for `ForeignKey` fields, but, as for `ManyToManyField`, sometimes you don't want to incur the overhead of having to select all the related objects to display in the drop-down. For example, if our book database grows to include thousands of publishers, the "Add book" form could take a while to load, because it would have to load every publisher for display in the `<select>` box.

The way to fix this is to use an option called `raw_id_fields`. Set this to a tuple of `ForeignKey` field names, and those fields will be displayed in the admin with a simple text input box (`<input type="text">`) instead of a `<select>`. See Figure 5-14.

```

class BookAdmin(admin.ModelAdmin):

    list_display = ('title', 'publisher', 'publication_date')

    list_filter = ('publication_date',)

    date_hierarchy = 'publication_date'

    ordering = ('-publication_date',)

    filter_horizontal = ('authors',)

    raw_id_fields = ('publisher',)

```

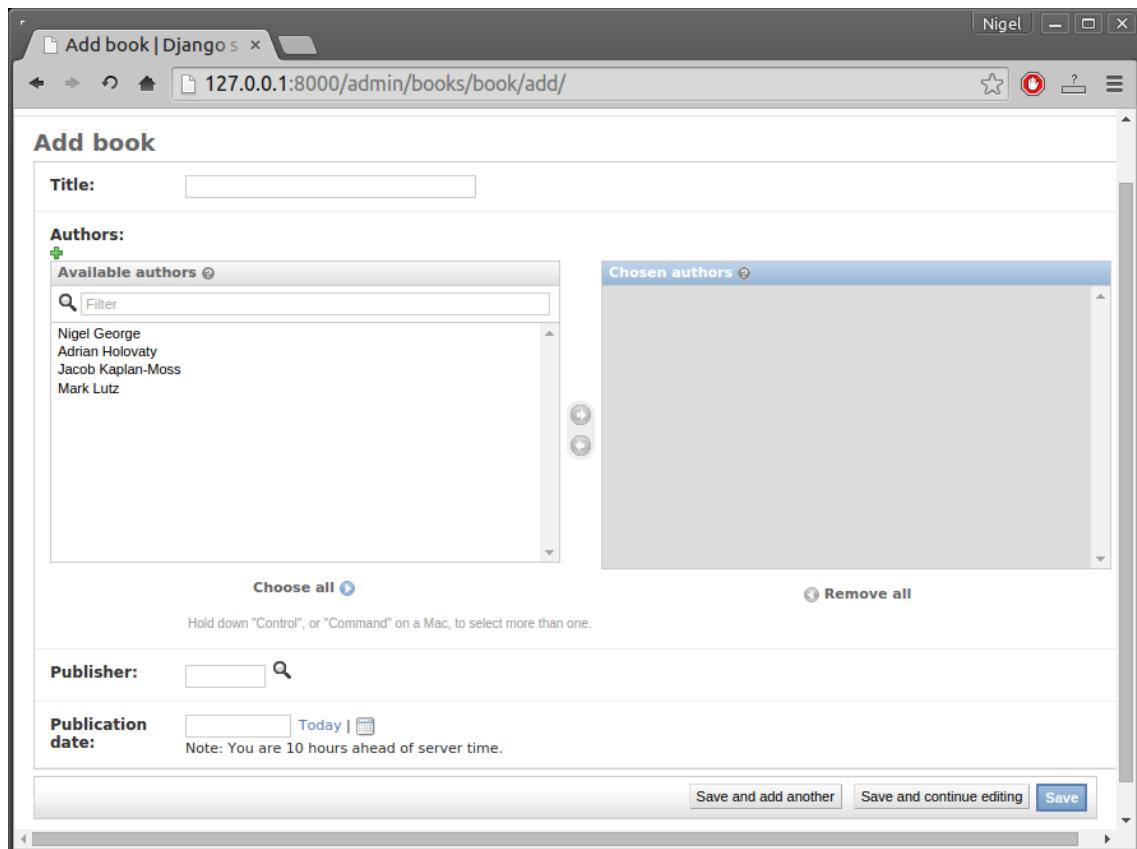


Figure 5-14. The book edit form after adding raw_id_fields

What do you enter in this input box? The database ID of the publisher. Given that humans don't normally memorize database IDs, there's also a magnifying-glass icon that you can click to pull up a pop-up window, from which you can select the publisher to add.

USERS, GROUPS, AND PERMISSIONS

Because you're logged in as a superuser, you have access to create, edit, and delete any object. Naturally, different environments require different permission systems – not everybody can or should be a superuser. Django's admin site uses a permissions system that you can use to give specific users access only to the portions of the interface that they need.

These user accounts are meant to be generic enough to be used outside of the admin interface, but we'll just treat them as admin user accounts for now. In [Chapter 11](#), we'll cover how to manage users site-wide (i.e., not just the admin site) with Django's authentication system.

You can edit users and permissions through the admin interface just like any other object. We saw this earlier in this chapter, when we played around with the User and Group sections of the admin. User objects have the standard username, password, e-mail and real name fields you might expect, along with a set of fields that define what the user is allowed to do in the admin interface. First, there's a set of three boolean flags:

- The “active” flag controls whether the user is active at all. If this flag is off and the user tries to log in, he won't be allowed in, even with a valid password.
- The “staff” flag controls whether the user is allowed to log in to the admin interface (i.e., whether that user is considered a “staff member” in your organization). Since this same user system can be used to control access to public (i.e., non-admin) sites (see [Chapter 11](#)), this flag differentiates between public users and administrators.
- The “superuser” flag gives the user full access to add, create and delete any item in the admin interface. If a user has this flag set, then all regular permissions (or lack thereof) are ignored for that user.

“Normal” admin users – that is, active, non-superuser staff members – are granted admin access through assigned permissions. Each object editable through the admin interface (e.g., books, authors, publishers) has three permissions: a *create* permission, an *edit* permission and a *delete* permission. Assigning permissions to a user grants the user access to do what is described by those permissions.

When you create a user, that user has no permissions, and it's up to you to give the user specific permissions. For example, you can give a user permission to add and change publishers, but not permission to delete them. Note that these permissions are defined per-model, not per-object – so they let you say “John can make changes to any book,” but they don't let you say “John can make changes to any book published by Apress.” The latter

functionality, per-object permissions, is a bit more complicated and is outside the scope of this book but is covered in the Django documentation.

Note

Access to edit users and permissions is also controlled by this permission system. If you give someone permission to edit users, they will be able to edit their own permissions, which might not be what you want! Giving a user permission to edit users is essentially turning a user into a superuser.

You can also assign users to groups. A *group* is simply a set of permissions to apply to all members of that group. Groups are useful for granting identical permissions to a subset of users.

WHEN AND WHY TO USE THE ADMIN INTERFACE – AND WHEN NOT TO

After having worked through this chapter, you should have a good idea of how to use Django’s admin site. But we want to make a point of covering *when* and *why* you might want to use it – and when *not* to use it.

Django’s admin site especially shines when nontechnical users need to be able to enter data; that’s the purpose behind the feature, after all. At the newspaper where Django was first developed, development of a typical online feature – say, a special report on water quality in the municipal supply – would go something like this:

- The reporter responsible for the project meets with one of the developers and describes the available data.
- The developer designs Django models to fit this data and then opens up the admin site to the reporter.
- The reporter inspects the admin site to point out any missing or extraneous fields – better now than later. The developer changes the models iteratively.

- When the models are agreed upon, the reporter begins entering data using the admin site. At the same time, the programmer can focus on developing the publicly accessible views/templates (the fun part!).

In other words, the *raison d'être* of Django's admin interface is facilitating the simultaneous work of content producers and programmers.

However, beyond these obvious data entry tasks, the admin site is useful in a few other cases:

- *Inspecting data models*: Once you've defined a few models, it can be quite useful to call them up in the admin interface and enter some dummy data. In some cases, this might reveal data-modeling mistakes or other problems with your models.
- *Managing acquired data*: For applications that rely on data coming from external sources (e.g., users or Web crawlers), the admin site gives you an easy way to inspect or edit this data. You can think of it as a less powerful, but more convenient, version of your database's command-line utility.
- *Quick and dirty data-management apps*: You can use the admin site to build yourself a very lightweight data management app – say, to keep track of expenses. If you're just building something for your own needs, not for public consumption, the admin site can take you a long way. In this sense, you can think of it as a beefed up, relational version of a spreadsheet.

The admin site is *not*, however, a be-all and end-all. It's not intended to be a *public* interface to data, nor is it intended to allow for sophisticated sorting and searching of your data. As we said early in this chapter, it's for trusted site administrators. Keeping this sweet spot in mind is the key to effective admin-site usage.

WHAT'S NEXT?

So far we've created a few models and configured a top-notch interface for editing data. In the next chapter we'll move on to the real "meat and potatoes" of Web development: [form creation and processing](#).

CHAPTER 6: FORMS

HTML forms are the backbone of interactive Web sites, from the simplicity of Google’s single search box to ubiquitous blog comment-submission forms to complex custom data-entry interfaces. This chapter covers how you can use Django to access user-submitted form data, validate it and do something with it. Along the way, we’ll cover `HttpRequest` and `Form` objects.

GETTING DATA FROM THE REQUEST OBJECT

We introduced `HttpRequest` objects in [Chapter 2](#) when we first covered view functions, but we didn’t have much to say about them at the time. Recall that each view function takes an `HttpRequest` object as its first parameter, as in our `hello()` view:

```
from django.http import HttpResponse
```

```
def hello(request):  
    return HttpResponse("Hello world")
```

`HttpRequest` objects, such as the variable `request` here, have a number of interesting attributes and methods that you should familiarize yourself with, so that you know what’s possible. You can use these attributes to get information about the current request (i.e., the user/Web browser that’s loading the current page on your Django-powered site), at the time the view function is executed.

INFORMATION ABOUT THE URL

`HttpRequest` objects contain several pieces of information about the currently requested URL:

Attribute/method	Description	Example
<code>request.path</code>	The full path, not including the domain but including the leading slash.	<code>"/hello/"</code>
<code>request.get_host()</code>	The host (i.e., the “domain,” in common parlance).	<code>"127.0.0.1:8000"</code> or <code>"www.example.com"</code>

Attribute/method	Description	Example
<code>request.get_full_path()</code>	The path, plus a query string (if available).	<code>"/hello/?print=true"</code>
<code>request.is_secure()</code>	True if the request was made via HTTPS. Otherwise, False.	True or False

Always use these attributes/methods instead of hard-coding URLs in your views. This makes for more flexible code that can be reused in other places. A simplistic example:

```
# BAD!

def current_url_view_bad(request):
    return HttpResponse("Welcome to the page at /current/")

# GOOD

def current_url_view_good(request):
    return HttpResponse("Welcome to the page at %s" % request.path)
```

OTHER INFORMATION ABOUT THE REQUEST

`request.META` is a Python dictionary containing all available HTTP headers for the given request – including the user’s IP address and user agent (generally the name and version of the Web browser). Note that the full list of available headers depends on which headers the user sent and which headers your Web server sets. Some commonly available keys in this dictionary are:

- `HTTP_REFERER` – The referring URL, if any. (Note the misspelling of `REFERER`.)
- `HTTP_USER_AGENT` – The user’s browser’s user-agent string, if any. This looks something like: `"Mozilla/5.0 (X11; U; Linux i686; fr-FR; rv:1.8.1.17) Gecko/20080829 Firefox/2.0.0.17"`.
- `REMOTE_ADDR` – The IP address of the client, e.g., `"12.345.67.89"`. (If the request has passed through any proxies, then this might be a comma-separated list of IP addresses, e.g., `"12.345.67.89,23.456.78.90"`.)

Note that because `request.META` is just a basic Python dictionary, you’ll get a `KeyError` exception if you try to access a key that doesn’t exist. (Because HTTP headers are *external*

data – that is, they’re submitted by your users’ browsers – they shouldn’t be trusted, and you should always design your application to fail gracefully if a particular header is empty or doesn’t exist.) You should either use a `try/except` clause or the `get()` method to handle the case of undefined keys:

```
# BAD!

def ua_display_bad(request):
    ua = request.META['HTTP_USER_AGENT'] # Might raise KeyError!
    return HttpResponse("Your browser is %s" % ua)

# GOOD (VERSION 1)

def ua_display_good1(request):
    try:
        ua = request.META['HTTP_USER_AGENT']
    except KeyError:
        ua = 'unknown'
    return HttpResponse("Your browser is %s" % ua)

# GOOD (VERSION 2)

def ua_display_good2(request):
    ua = request.META.get('HTTP_USER_AGENT', 'unknown')
    return HttpResponse("Your browser is %s" % ua)
```

We encourage you to write a small view that displays all of the `request.META` data so you can get to know what’s in there. Here’s what that view might look like:

```
def display_meta(request):
    values = request.META.items()
    values.sort()
    html = []
    for k, v in values:
        html.append('<tr><td>%s</td><td>%s</td></tr>' % (k, v))
```

```
return HttpResponse('<table>%s</table>' % '\n'.join(html))
```

Another good way to see what sort of information that the request object contains is to look closely at the Django error pages when you crash the system – there is a wealth of useful information in there – including all the HTTP headers and other request objects (`request.path` for example).

INFORMATION ABOUT SUBMITTED DATA

Beyond basic metadata about the request, `HttpRequest` objects have two attributes that contain information submitted by the user: `request.GET` and `request.POST`. Both of these are dictionary-like objects that give you access to GET and POST data.

Dictionary-like objects

When we say `request.GET` and `request.POST` are “dictionary-like” objects, we mean that they behave like standard Python dictionaries but aren’t technically dictionaries under the hood. For example, `request.GET` and `request.POST` both have `get()`, `keys()` and `values()` methods, and you can iterate over the keys by doing `for key in request.GET`.

So why the distinction? Because both `request.GET` and `request.POST` have additional methods that normal dictionaries don’t have. We’ll get into these in a short while.

You might have encountered the similar term “file-like objects” – Python objects that have a few basic methods, like `read()`, that let them act as stand-ins for “real” file objects.

`POST` data generally is submitted from an HTML `<form>`, while `GET` data can come from a `<form>` or the query string in the page’s URL.

A SIMPLE FORM-HANDLING EXAMPLE

Continuing this book’s ongoing example of books, authors and publishers, let’s create a simple view that lets users search our book database by title.

Generally, there are two parts to developing a form: the HTML user interface and the backend view code that processes the submitted data. The first part is easy; let's just set up a view that displays a search form:

```
from django.shortcuts import render

def search_form(request):
    return render(request, 'search_form.html')
```

As we learned in [Chapter 3](#), this view can live anywhere on your Python path. For sake of argument, put it in `books/views.py`.

The accompanying template, `search_form.html`, could look like this:

```
<html>
<head>
    <title>Search</title>
</head>
<body>
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

Save this file to your `mysite/templates` directory you created in [Chapter 3](#), or you can create a new folder `books/templates`. Just make sure you have `'APP_DIRS'` in your settings file set to True.

The URLpattern in `urls.py` could look like this:

```
from books import views

urlpatterns = [
```

```
# ...

url(r'^search-form/$', views.search_form),
# ...
]
```

(Note that we're importing the `views` module directly, instead of something like `from mysite.views import search_form`, because the former is less verbose. We'll cover this importing approach in more detail in [Chapter 7](#).)

Now, if you run the `runserver` and visit `http://127.0.0.1:8000/search-form/`, you'll see the search interface. Simple enough.

Try submitting the form, though, and you'll get a Django 404 error. The form points to the URL `/search/`, which hasn't yet been implemented. Let's fix that with a second view function:

```
# urls.py
```

```
urlpatterns = [
# ...
url(r'^search-form/$', views.search_form),
url(r'^search/$', views.search),
# ...
]
```

```
# books/views.py
```

```
from django.http import HttpResponseRedirect
```

```
# ...

def search(request):
    if 'q' in request.GET:
```

```

message = 'You searched for: %r' % request.GET['q']

else:

    message = 'You submitted an empty form.'

return HttpResponse(message)

```

For the moment, this merely displays the user's search term, so we can make sure the data is being submitted to Django properly, and so you can get a feel for how the search term flows through the system. In short:

1. The HTML <form> defines a variable `q`. When it's submitted, the value of `q` is sent via GET (`method="get"`) to the URL /search/.
2. The Django view that handles the URL /search/ (`search()`) has access to the `q` value in `request.GET`.

An important thing to point out here is that we explicitly check that '`q`' exists in `request.GET`. As we pointed out in the `request.META` section above, you shouldn't trust anything submitted by users or even assume that they've submitted anything in the first place. If we didn't add this check, any submission of an empty form would raise `KeyError` in the view:

```

# BAD!

def bad_search(request):

    # The following line will raise KeyError if 'q' hasn't
    # been submitted!

    message = 'You searched for: %r' % request.GET['q']

    return HttpResponse(message)

```

Query string parameters

Because GET data is passed in the query string (e.g., /search/?q=django), you can use `request.GET` to access query string variables. In Chapter 2's introduction of [Django's URLconf system](#), we compared Django's pretty URLs to more traditional PHP/Java URLs such as /time/plus?hours=3 and said we'd show you how to do the latter in Chapter 6. Now you know how to access query string parameters in your views (like `hours=3` in this example) – use `request.GET`.

`POST` data works the same way as `GET` data – just use `request.POST` instead of `request.GET`. What's the difference between `GET` and `POST`? Use `GET` when the act of submitting the form is just a request to “get” data. Use `POST` whenever the act of submitting the form will have some side effect – *changing* data, or sending an e-mail, or something else that's beyond simple *display* of data. In our book-search example, we're using `GET` because the query doesn't change any data on our server. (See <http://www.w3.org/2001/tag/doc/whenToUseGet.html> if you want to learn more about `GET` and `POST`.)

Now that we've verified `request.GET` is being passed in properly, let's hook the user's search query into our book database (again, in `views.py`):

```
from django.http import HttpResponseRedirect  
from django.shortcuts import render  
from books.models import Book  
  
def search(request):  
  
    if 'q' in request.GET and request.GET['q']:  
  
        q = request.GET['q']  
  
        books = Book.objects.filter(title__icontains=q)  
  
        return render(request, 'search_results.html',  
                      {'books': books, 'query': q})  
  
    else:  
  
        return HttpResponseRedirect('Please submit a search term.')
```

A couple of notes on what we did here:

- Aside from checking that '`q`' exists in `request.GET`, we also make sure that `request.GET['q']` is a non-empty value before passing it to the database query.
- We're using `Book.objects.filter(title__icontains=q)` to query our book table for all books whose title includes the given submission. The `icontains` is a lookup type (as explained in [Chapter 4](#) and [Appendix B](#)), and the statement can be roughly translated as “Get the books whose title contains `q`, without being case-sensitive.”

This is a very simple way to do a book search. We wouldn't recommend using a simple `icontains` query on a large production database, as it can be slow. (In the real world,

you'd want to use a custom search system of some sort. Search the Web for *open-source full-text search* to get an idea of the possibilities.)

- We pass `books`, a list of `Book` objects, to the template. The `search_results.html` file might include something like this:

```
<html>
  <head>
    <title>Book Search</title>
  </head>
  <body>
    <p>You searched for: <strong>{{ query }}</strong></p>
    {% if books %}
      <p>Found {{ books|length }} book{{ books|pluralize }}.</p>
      <ul>
        {% for book in books %}
          <li>{{ book.title }}</li>
        {% endfor %}
      </ul>
    {% else %}
      <p>No books matched your search criteria.</p>
    {% endif %}
  </body>
</html>
```

Note usage of the `pluralize` template filter, which outputs an “s” if appropriate, based on the number of books found.

IMPROVING OUR SIMPLE FORM-HANDLING EXAMPLE

As in previous chapters, we've shown you the simplest thing that could possibly work. Now we'll point out some problems and show you how to improve it.

First, our `search()` view's handling of an empty query is poor – we're just displaying a "Please submit a search term." message, requiring the user to hit the browser's back button. This is horrid and unprofessional, and if you ever actually implement something like this in the wild, your Django privileges will be revoked.

It would be much better to redisplay the form, with an error above it, so that the user can try again immediately. The easiest way to do that would be to render the template again, like this:

```
from django.http import HttpResponseRedirect  
from django.shortcuts import render  
from books.models import Book  
  
def search_form(request):  
    return render(request, 'search_form.html')  
  
def search(request):  
    if 'q' in request.GET and request.GET['q']:  
        q = request.GET['q']  
        books = Book.objects.filter(title__icontains=q)  
        return render(request, 'search_results.html',  
                      {'books': books, 'query': q})  
    else:  
        return render(request, 'search_form.html', {'error': True})
```

(Note that we've included `search_form()` here so you can see both views in one place.)

Here, we've improved `search()` to render the `search_form.html` template again, if the query is empty. And because we need to display an error message in that template, we pass a template variable. Now we can edit `search_form.html` to check for the `error` variable:

```
<html>  
<head>  
    <title>Search</title>  
</head>
```

```

<body>

  {%- if error %}

    <p style="color: red;">Please submit a search term.</p>

  {% endif %}

  <form action="/search/" method="get">

    <input type="text" name="q">

    <input type="submit" value="Search">

  </form>

</body>

</html>

```

We can still use this template from our original view, `search_form()`, because `search_form()` doesn't pass `error` to the template – so the error message won't show up in that case.

With this change in place, it's a better application, but it now begs the question: is a dedicated `search_form()` view really necessary? As it stands, a request to the URL `/search/` (without any GET parameters) will display the empty form (but with an error). We can remove the `search_form()` view, along with its associated URLpattern, as long as we change `search()` to hide the error message when somebody visits `/search/` with no GET parameters:

```

def search(request):

    error = False

    if 'q' in request.GET:

        q = request.GET['q']

        if not q:

            error = True

    else:

        books = Book.objects.filter(title__icontains=q)

        return render(request, 'search_results.html',
                      {'books': books, 'query': q})

    return render(request, 'search_form.html',

```

```
{'error': error})
```

In this updated view, if a user visits `/search/` with no GET parameters, he'll see the search form with no error message. If a user submits the form with an empty value for '`q`', he'll see the search form *with* an error message. And, finally, if a user submits the form with a non-empty value for '`q`', he'll see the search results.

We can make one final improvement to this application, to remove a bit of redundancy. Now that we've rolled the two views and URLs into one and `/search/` handles both search-form display and result display, the HTML `<form>` in `search_form.html` doesn't have to hard-code a URL. Instead of this:

```
<form action="/search/" method="get">
```

It can be changed to this:

```
<form action="" method="get">
```

The `action=""` means "Submit the form to the same URL as the current page." With this change in place, you won't have to remember to change the `action` if you ever hook the `search()` view to another URL.

SIMPLE VALIDATION

Our search example is still reasonably simple, particularly in terms of its data validation; we're merely checking to make sure the search query isn't empty. Many HTML forms include a level of validation that's more complex than making sure the value is non-empty. We've all seen the error messages on Web sites:

- "Please enter a valid e-mail address. 'foo' is not an e-mail address."
- "Please enter a valid five-digit U.S. ZIP code. '123' is not a ZIP code."
- "Please enter a valid date in the format YYYY-MM-DD."
- "Please enter a password that is at least 8 characters long and contains at least one number."

A note on JavaScript validation

This is beyond the scope of this book, but you can use JavaScript to validate data on the client side, directly in the browser. But be warned: even if you do this, you *must* validate data on the server side, too. Some people have JavaScript turned off, and some malicious users might submit raw, unvalidated data directly to your form handler to see whether they can cause mischief.

There's nothing you can do about this, other than *always* validate user-submitted data server-side (i.e., in your Django views). You should think of JavaScript validation as a bonus usability feature, not as your only means of validating.

Let's tweak our `search()` view so that it validates that the search term is less than or equal to 20 characters long. (For sake of example, let's say anything longer than that might make the query too slow.) How might we do that? The simplest possible thing would be to embed the logic directly in the view, like this:

```
def search(request):
    error = False
    if 'q' in request.GET:
        q = request.GET['q']
        if not q:
            error = True
        elif len(q) > 20:
            error = True
        else:
            books = Book.objects.filter(title__icontains=q)
            return render(request, 'search_results.html',
                          {'books': books, 'query': q})
    return render(request, 'search_form.html',
                  {'error': error})
```

Now, if you try submitting a search query greater than 20 characters long, it won't let you search; you'll get an error message. But that error message in `search_form.html` currently

says "Please submit a search term." – so we'll have to change it to be accurate for both cases:

```
<html>
<head>
    <title>Search</title>
</head>
<body>
    {%- if error %}
        <p style="color: red;">Please submit a search term 20 characters or
shorter.</p>
    {%- endif %}
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

There's something ugly about this. Our one-size-fits-all error message is potentially confusing. Why should the error message for an empty form submission mention anything about a 20-character limit? Error messages should be specific, unambiguous and not confusing.

The problem is in the fact that we're using a simple boolean value for `error`, whereas we should be using a *list* of error message strings. Here's how we might fix that:

```
def search(request):
    errors = []
    if 'q' in request.GET:
        q = request.GET['q']
        if not q:
            errors.append('Enter a search term.')
        elif len(q) > 20:
```

```

        errors.append('Please enter at most 20 characters.')

    else:

        books = Book.objects.filter(title__icontains=q)

        return render(request, 'search_results.html',
                      {'books': books, 'query': q})

    return render(request, 'search_form.html',
                  {'errors': errors})

```

Then, we need make a small tweak to the `search_form.html` template to reflect that it's now passed an `errors` list instead of an `error` boolean value:

```

<html>

<head>
    <title>Search</title>
</head>

<body>

    {% if errors %}

        <ul>

            {% for error in errors %}

                <li>{{ error }}</li>

            {% endfor %}

        </ul>

    {% endif %}

    <form action="/search/" method="get">

        <input type="text" name="q">

        <input type="submit" value="Search">

    </form>

</body>

</html>

```

MAKING A CONTACT FORM

Although we iterated over the book search form example several times and improved it nicely, it's still fundamentally simple: just a single field, 'q'. As forms get more complex, we have to repeat the above steps over and over again for each form field we use. This introduces a lot of cruft and a lot of opportunities for human error. Lucky for us, the Django developers thought of this and built into Django a higher-level library that handles form- and validation-related tasks.

YOUR FIRST FORM CLASS

Django comes with a form library, called `django.forms`, that handles many of the issues we've been exploring this chapter – from HTML form display to validation. Let's dive in and rework our contact form application using the Django forms framework.

The primary way to use the forms framework is to define a `Form` class for each HTML `<form>` you're dealing with. In our case, we only have one `<form>`, so we'll have one `Form` class. This class can live anywhere you want – including directly in your `views.py` file – but community convention is to keep `Form` classes in a separate file called `forms.py`. Create this file in the same directory as your `mysite/views.py`, and enter the following:

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField()
    email = forms.EmailField(required=False)
    message = forms.CharField()
```

This is pretty intuitive, and it's similar to Django's model syntax. Each field in the form is represented by a type of `Field` class – `CharField` and `EmailField` are the only types of fields used here – as attributes of a `Form` class. Each field is required by default, so to make `email` optional, we specify `required=False`.

Let's hop into the Python interactive interpreter and see what this class can do. The first thing it can do is display itself as HTML:

```
>>> from mysite.forms import ContactForm
>>> f = ContactForm()
>>> print(f)
```

```

<tr><th><label for="id_subject">Subject:</label></th><td><input type="text"
name="subject" id="id_subject" /></td></tr>

<tr><th><label for="id_email">Email:</label></th><td><input type="text"
name="email" id="id_email" /></td></tr>

<tr><th><label for="id_message">Message:</label></th><td><input type="text"
name="message" id="id_message" /></td></tr>

```

Django adds a label to each field, along with `<label>` tags for accessibility. The idea is to make the default behavior as optimal as possible.

This default output is in the format of an HTML `<table>`, but there are a few other built-in outputs:

```

>>> print(f.as_ul())

<li><label for="id_subject">Subject:</label> <input type="text"
name="subject" id="id_subject" /></li>

<li><label for="id_email">Email:</label> <input type="text" name="email"
id="id_email" /></li>

<li><label for="id_message">Message:</label> <input type="text"
name="message" id="id_message" /></li>

>>> print(f.as_p())

<p><label for="id_subject">Subject:</label> <input type="text"
name="subject" id="id_subject" /></p>

<p><label for="id_email">Email:</label> <input type="text" name="email"
id="id_email" /></p>

<p><label for="id_message">Message:</label> <input type="text"
name="message" id="id_message" /></p>

```

Note that the opening and closing `<table>`, `` and `<form>` tags aren't included in the output, so that you can add any additional rows and customization if necessary.

These methods are just shortcuts for the common case of “display the entire form.” You can also display the HTML for a particular field:

```

>>> print(f['subject'])

<input id="id_subject" name="subject" type="text" />

>>> print f['message']

<input id="id_message" name="message" type="text" />

```

The second thing `Form` objects can do is validate data. To validate data, create a new `Form` object and pass it a dictionary of data that maps field names to data:

```
>>> f = ContactForm({'subject': 'Hello', 'email': 'adrian@example.com',  
'message': 'Nice site!'})
```

Once you've associated data with a `Form` instance, you've created a "bound" form:

```
>>> f.is_bound
```

```
True
```

Call the `is_valid()` method on any bound `Form` to find out whether its data is valid. We've passed a valid value for each field, so the `Form` in its entirety is valid:

```
>>> f.is_valid()
```

```
True
```

If we don't pass the `email` field, it's still valid, because we've specified `required=False` for that field:

```
>>> f = ContactForm({'subject': 'Hello', 'message': 'Nice site!'})
```

```
>>> f.is_valid()
```

```
True
```

But, if we leave off either `subject` or `message`, the `Form` is no longer valid:

```
>>> f = ContactForm({'subject': 'Hello'})
```

```
>>> f.is_valid()
```

```
False
```

```
>>> f = ContactForm({'subject': 'Hello', 'message': ''})
```

```
>>> f.is_valid()
```

```
False
```

You can drill down to get field-specific error messages:

```
>>> f = ContactForm({'subject': 'Hello', 'message': ''})
```

```
>>> f['message'].errors
```

```
['This field is required.']}
```

```
>>> f['subject'].errors  
[]  
  
>>> f['email'].errors  
[]
```

Each bound `Form` instance has an `errors` attribute that gives you a dictionary mapping field names to error-message lists:

```
>>> f = ContactForm({'subject': 'Hello', 'message': ''})  
  
>>> f.errors  
  
{'message': ['This field is required.']}
```

Finally, for `Form` instances whose data has been found to be valid, a `cleaned_data` attribute is available. This is a dictionary of the submitted data, “cleaned up.” Django’s forms framework not only validates data, it cleans it up by converting values to the appropriate Python types:

```
>>> f = ContactForm({'subject': 'Hello', 'email': 'adrian@example.com',  
'message': 'Nice site!'} )  
  
>>> f.is_valid() True  
  
>>> f.cleaned_data  
  
{'message': 'Nice site!', 'email': 'adrian@example.com', 'subject':  
'Hello'}
```

Our contact form only deals with strings, which are “cleaned” into string objects – but if we were to use an `IntegerField` or `DateField`, the forms framework would ensure that `cleaned_data` used proper Python integers or `datetime.date` objects for the given fields.

TYING FORM OBJECTS INTO VIEWS

Our contact form is not much good to us unless we have some way of displaying it to the user. To do this, we need to first update our `mysite/views`:

```
# views.py  
  
from django.shortcuts import render
```

```

from mysite.forms import ContactForm
from django.http import HttpResponseRedirect
from django.core.mail import send_mail

# ...

def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
            send_mail(
                cd['subject'],
                cd['message'],
                cd.get('email', 'noreply@example.com'),
                ['siteowner@example.com'],
            )
            return HttpResponseRedirect('/contact/thanks/')
    else:
        form = ContactForm()
    return render(request, 'contact_form.html', {'form': form})

```

Next, we have to create our contact form (save this to `mysite/templates`):

```

# contact_form.html

<html>
<head>
    <title>Contact us</title>
</head>

```

```

<body>

    <h1>Contact us</h1>

    {% if form.errors %}

        <p style="color: red;">
            Please correct the error{{ form.errors|pluralize }} below.
        </p>

    {% endif %}

    <form action="" method="post">

        <table>
            {{ form.as_table }}
        </table>

        {% csrf_token %}

        <input type="submit" value="Submit">

    </form>

</body>

</html>

```

And finally, we need to change our `urls.py` to display our contact form at `/contact/`.

```

# ...

from mysite.views import hello, current_datetime, hours_ahead, contact

urlpatterns = [
    # ...
    url(r'^contact/$', contact),
]

```

Since we're creating a POST form (which can have the effect of modifying data), we need to worry about Cross Site Request Forgery. 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. More details about `{% csrf_token %}` can be found in [Chapter 21](#).

Try running this locally. Load the form, submit it with none of the fields filled out, submit it with an invalid e-mail address, then finally submit it with valid data. (Of course, unless you have configured a mail-server, you will get a `ConnectionRefusedError` when `send_mail()` is called, but that's another issue.)

Finally, if you are feeling clever enough to configure a mail-server (Appendix H shows you how to do this), you will need to write the view at `contact/thanks`.

CHANGING HOW FIELDS ARE RENDERED

Probably the first thing you'll notice when you render this form locally is that the `message` field is displayed as an `<input type="text">`, and it ought to be a `<textarea>`. We can fix that by setting the field's `widget`:

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField()
    email = forms.EmailField(required=False)
    message = forms.CharField(widget=forms.Textarea)
```

The forms framework separates out the presentation logic for each field into a set of widgets. Each field type has a default widget, but you can easily override the default, or provide a custom widget of your own.

Think of the `Field` classes as representing *validation logic*, while widgets represent *presentation logic*.

SETTING A MAXIMUM LENGTH

One of the most common validation needs is to check that a field is of a certain size. For good measure, we should improve our `ContactForm` to limit the `subject` to 100 characters. To do that, just supply a `max_length` to the `CharField`, like this:

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    email = forms.EmailField(required=False)
    message = forms.CharField(widget=forms.Textarea)
```

An optional `min_length` argument is also available.

SETTING INITIAL VALUES

As an improvement to this form, let's add an *initial value* for the `subject` field: "I love your site!" (A little power of suggestion can't hurt.) To do this, we can use the `initial` argument when we create a `Form` instance:

```
def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
            send_mail(
                cd['subject'],
                cd['message'],
                cd.get('email', 'noreply@example.com'),
                ['siteowner@example.com'],
            )
            return HttpResponseRedirect('/contact/thanks/')
```

```

else:

    form = ContactForm(
        initial={'subject': 'I love your site!'}
    )

    return render(request, 'contact_form.html', {'form': form})

```

Now, the `subject` field will be displayed prepopulated with that kind statement.

Note that there is a difference between passing *initial* data and passing data that *binds* the form. The biggest difference is that if you're just passing *initial* data, then the form will be *unbound*, which means it won't have any error messages.

CUSTOM VALIDATION RULES

Imagine we've launched our feedback form, and the e-mails have started tumbling in. There's just one problem: some of the submitted messages are just one or two words, which isn't long enough for us to make sense of. We decide to adopt a new validation policy: four words or more, please.

There are a number of ways to hook custom validation into a Django form. If our rule is something we will reuse again and again, we can create a custom field type. Most custom validations are one-off affairs, though, and can be tied directly to the `Form` class.

We want additional validation on the `message` field, so we add a `clean_message()` method to our `Form` class:

```

from django import forms

class ContactForm(forms.Form):

    subject = forms.CharField(max_length=100)
    email = forms.EmailField(required=False)
    message = forms.CharField(widget=forms.Textarea)

    def clean_message(self):
        message = self.cleaned_data['message']

```

```

num_words = len(message.split())
if num_words < 4:
    raise forms.ValidationError("Not enough words!")
return message

```

Django's form system automatically looks for any method whose name starts with `clean_` and ends with the name of a field. If any such method exists, it's called during validation.

Specifically, the `clean_message()` method will be called *after* the default validation logic for a given field (in this case, the validation logic for a required `CharField`). Because the field data has already been partially processed, we pull it out of `self.cleaned_data`. Also, we don't have to worry about checking that the value exists and is non-empty; that's done by the default validator.

We naively use a combination of `len()` and `split()` to count the number of words. If the user has entered too few words, we raise a `forms.ValidationError`. The string attached to this exception will be displayed to the user as an item in the error list.

It's important that we explicitly return the cleaned value for the field at the end of the method. This allows us to modify the value (or convert it to a different Python type) within our custom validation method. If we forget the `return` statement, then `None` will be returned, and the original value will be lost.

For more on Django's validation tools, see Appendix H.

SPECIFYING LABELS

By default, the labels on Django's auto-generated form HTML are created by replacing underscores with spaces and capitalizing the first letter – so the label for the `email` field is "Email". (Sound familiar? It's the same simple algorithm that Django's models use to calculate default `verbose_name` values for fields. We covered this in [Chapter 4](#).)

But, as with Django's models, we can customize the label for a given field. Just use `label`, like so:

```

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    email = forms.EmailField(required=False, label='Your e-mail address')

```

```
message = forms.CharField(widget=forms.Textarea)
```

CUSTOMIZING FORM DESIGN

Our `contact_form.html` template uses `{{ form.as_table }}` to display the form, but we can display the form in other ways to get more granular control over display.

The quickest way to customize forms' presentation is with CSS. Error lists, in particular, could do with some visual enhancement, and the auto-generated error lists use `<ul class="errorlist">` precisely so that you can target them with CSS. The following CSS really makes our errors stand out:

```
<style type="text/css">

    ul.errorlist {
        margin: 0;
        padding: 0;
    }

    .errorlist li {
        background-color: red;
        color: white;
        display: block;
        font-size: 10px;
        margin: 0 0 3px;
        padding: 4px 5px;
    }

</style>
```

While it's convenient to have our form's HTML generated for us, in many cases you'll want to override the default rendering. `{{ form.as_table }}` and friends are useful shortcuts while you develop your application, but everything about the way a form is displayed can be overridden, mostly within the template itself, and you'll probably find yourself doing this.

Each field's `widget` (`<input type="text">`, `<select>`, `<textarea>`, etc.) can be rendered individually by accessing `{{ form.fieldname }}` in the template, and any errors associated

with a field are available as `{{ form.fieldname.errors }}`. With this in mind, we can construct a custom template for our contact form with the following template code:

```
<html>
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>

    {% if form.errors %}
        <p style="color: red;">
            Please correct the error{{ form.errors|pluralize }} below.
        </p>
    {% endif %}

    <form action="" method="post">
        <div class="field">
            {{ form.subject.errors }}
            <label for="id_subject">Subject:</label>
            {{ form.subject }}
        </div>
        <div class="field">
            {{ form.email.errors }}
            <label for="id_email">Your e-mail address:</label>
            {{ form.email }}
        </div>
        <div class="field">
            {{ form.message.errors }}
        </div>
    </form>

```

```

<label for="id_message">Message:</label>
{{ form.message }}
</div>
<input type="submit" value="Submit">
</form>
</body>
</html>

```

`{{ form.message.errors }}` displays a `<ul class="errorlist">` if errors are present and a blank string if the field is valid (or the form is unbound). We can also treat `form.message.errors` as a Boolean or even iterate over it as a list. For example:

```

<div class="field{% if form.message.errors %} errors{% endif %}>
  {% if form.message.errors %}
    <ul>
      {% for error in form.message.errors %}
        <li><strong>{{ error }}</strong></li>
      {% endfor %}
    </ul>
  {% endif %}
  <label for="id_message">Message:</label>
  {{ form.message }}
</div>

```

In the case of validation errors, this will add an “errors” class to the containing `<div>` and display the list of errors in an unordered list.

WHAT'S NEXT?

This chapter concludes the introductory material in this book – the so-called “core curriculum.” The next section of the book, Chapters 7 to 13, goes into more detail about advanced Django usage, including how to [deploy a Django application](#) (Chapter 13).

After these first seven chapters, you should know enough to start writing your own Django projects. The rest of the material in this book will help fill in the missing pieces as you need them.

We'll start in [Chapter 7](#), by doubling back and taking a closer look at views and URLconfs (introduced first in [Chapter 2](#)).

CHAPTER 7: ADVANCED VIEWS AND URLCONFS

In Chapter 2, we explained the basics of [Django view functions and URLconfs](#). This chapter goes into more detail about advanced functionality in those two pieces of the framework.

URLCONF TRICKS

There's nothing "special" about URLconfs – like anything else in Django, they're just Python code. You can take advantage of this in several ways, as described in the sections that follow.

STREAMLINING FUNCTION IMPORTS

Consider this URLconf, which builds on the example in Chapter 2:

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

from django.contrib import admin

from mysite.views import hello, current_datetime, hours_ahead


urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^hello/$', hello),
    url(r'^time/$', current_datetime),
    url(r'^time/plus/(\d{1,2})/$', hours_ahead),
]
```

As explained in Chapter 2, each entry in the URLconf includes its associated view function, passed directly as a function object. This means it's necessary to import the view functions at the top of the module.

But as a Django application grows in complexity, its URLconf grows, too, and keeping those imports can be tedious to manage. (For each new view function, you have to remember to import it, and the import statement tends to get overly long if you use this approach.) It's possible to avoid that tedium by importing the `views` module itself. This example URLconf is equivalent to the previous one:

```
from django.conf.urls import include, url
from . import views

urlpatterns = [
    url(r'^hello/$', views.hello),
    url(r'^time/$', views.current_datetime),
    url(r'^time/plus/(\d{1,2})/$', views.hours_ahead),
]
```

SPECIAL-CASING URLs IN DEBUG MODE

Speaking of constructing `urlpatterns` dynamically, you might want to take advantage of this technique to alter your URLconf's behavior while in Django's debug mode. To do this, just check the value of the `DEBUG` setting at runtime, like so:

```
from django.conf import settings
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^$', views.homepage),
    url(r'^(\d{4})/([a-z]{3})/$', views.archive_month),
]
```

```
if settings.DEBUG:  
    urlpatterns += [url(r'^debuginfo/$', views.debug),]
```

In this example, the URL `/debuginfo/` will only be available if your `DEBUG` setting is set to `True`.

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.

Keyword Arguments vs. Positional Arguments

A Python function can be called using keyword arguments or positional arguments – and, in some cases, both at the same time. In a keyword argument call, you specify the names of the arguments along with the values you're passing. In a positional argument call, you simply pass the arguments without explicitly specifying which argument matches which value; the association is implicit in the arguments' order.

For example, consider this simple function:

```
def sell(item, price, quantity):  
    print "Selling %s unit(s) of %s at %s" % (quantity, item, price)
```

To call it with positional arguments, you specify the arguments in the order in which they're listed in the function definition:

```
sell('Socks', '$2.50', 6)
```

To call it with keyword arguments, you specify the names of the arguments along with the values. The following statements are equivalent:

```
sell(item='Socks', price='$2.50', quantity=6)  
sell(item='Socks', quantity=6, price='$2.50')  
sell(price='$2.50', item='Socks', quantity=6)  
sell(price='$2.50', quantity=6, item='Socks')
```

```
sell(quantity=6, item='Socks', price='$2.50')

sell(quantity=6, price='$2.50', item='Socks')
```

Finally, you can mix keyword and positional arguments, as long as all positional arguments are listed before keyword arguments. The following statements are equivalent to the previous examples:

```
sell('Socks', '$2.50', quantity=6)

sell('Socks', price='$2.50', quantity=6)

sell('Socks', quantity=6, price='$2.50')
```

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.

For example, say we have a list of book reviews on our books site, and we want to retrieve reviews for certain dates, or date ranges. Here's a sample URLconf:

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^reviews/2003/$', views.special_case_2003),
    url(r'^reviews/([0-9]{4})/$', views.year_archive),
    url(r'^reviews/([0-9]{4})/([0-9]{2})/$', views.month_archive),
    url(r'^reviews/([0-9]{4})/([0-9]{2})/([0-9]+)/$', views.review_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 `^reviews`, not `^/reviews`.
- 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.

Example requests:

- A request to `/reviews/2005/03/` would match the third entry in the list. Django would call the function `views.month_archive(request, '2005', '03')`.
- `/reviews/2005/3/` would not match any URL patterns, because the third entry in the list requires two digits for the month.
- `/reviews/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.
- `/reviews/2003` would not match any of these patterns, because each pattern requires that the URL end with a slash.
- `/reviews/2003/03/03/` would match the final pattern. Django would call the function `views.review_detail(request, '2003', '03', '03')`.

Here's the above example URLconf, rewritten to use named groups:

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^reviews/2003/$', views.special_case_2003),
    url(r'^reviews/(?P<year>[0-9]{4})/$', views.year_archive),
    url(r'^reviews/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/$', views.month_archive),
    url(r'^reviews/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<day>[0-9]{2})/$', views.review_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 `/reviews/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 /reviews/2003/03/03/ would call the function `views.review_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 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 `http://www.example.com/myapp/`, the URLconf will look for `myapp/`.

In a request to `http://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'^reviews/(?P<year>[0-9]{4})/$', views.year_archive),
```

...the `year` argument 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:

```
# URLconf

from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^reviews/$', views.page),
    url(r'^reviews/page(?P<num>[0-9]+)/$', views.page),
]

# View (in reviews/views.py)

def page(request, num="1"):
    # Output the appropriate page of review 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.

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.

The variables are:

- handler404
- handler500
- handler403
- handler400

Their default values should suffice for most projects, but further customization is possible by assigning values to them.

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.

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 Web site itself. It includes a number of other URLconfs:

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

urlpatterns = [
    # ...
    url(r'^community/', include('django_website.aggregator.urls')),
    url(r'^contact/', include('django_website.contact.urls')),
    # ...
]
```

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/(?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+)/reviews/$', include('foo.urls.reviews')),
]

# In foo/urls/reviews.py

from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^$', views.reviews.index),
    url(r'^archive/$', views.reviews.archive),
]

```

In the above example, the captured "username" variable is passed to the included URLconf, as expected.

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'^reviews/(?P<year>[0-9]{4})/$', views.year_archive, {'foo': 'bar'}),
]
```

In this example, for a request to `/reviews/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 (see [Chapter 15](#)).

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'^reviews/', include('inner'), {'reviewid': 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'^reviews/', include('inner')),
```

```

]

# inner.py

from django.conf.urls import url


urlpatterns = [
    url(r'^archive/$', views.archive, {'reviewid': 3}),
    url(r'^about/$', views.about, {'reviewid': 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 not having to hard-code these URLs (a laborious, non-scalable and error-prone strategy) or having to devise ad-hoc mechanisms for generating URLs that are parallel to the design described by the URLconf and as such in danger of producing stale URLs at some point.

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 all over the project source code to search and replace outdated URLs.

The piece of information we have available as a starting point 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 `django.core.urlresolvers.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'^reviews/([0-9]{4})/$', views.year_archive, name='reviews-year-archive'),
    #...
]
```

According to this design, the URL for the archive corresponding to year *nnnn* is `/reviews/nnnn/`.

You can obtain these in template code by using:

```
<a href="{% url 'reviews-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 'reviews-year-archive' yearvar %}">{{ yearvar }}</a></li>

{%- endfor %}

</ul>

```

Or in Python code:

```

from django.core.urlresolvers import reverse
from django.http import HttpResponseRedirect

def redirect_to_year(request):
    # ...
    year = 2012
    # ...
    return HttpResponseRedirect(reverse('reviews-year-archive',
                                     args=(year,)))

```

If, for some reason, it was decided that the URLs where content for yearly review 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. 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 once instance of the `admin``.

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 `'members:reviews:index'` would look for a pattern named '`index`' in the namespace '`reviews`' that is itself defined within the top-level namespace '`members`'.

REVERSING NAMESPACED URLs

When given a namespaced URL (e.g. `'reviews: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, `'reviews'`). 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 as an attribute on the request. Applications that expect to have multiple deployments should set the `current_app` attribute on the `request` being processed.

The current application can also be specified manually as an argument to the `reverse()` function.

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 `reviews` called `'reviews'`).
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.

URL NAMESPACES AND INCLUDED URLCONFS

URL namespaces of included URLconfs can be specified in two ways.

Firstly, you can provide the application and instance namespaces as arguments to `include()` when you construct your URL patterns. For example,:

```
url(r'^reviews/',     include('reviews.urls',      namespace='author-reviews',
                             app_name='reviews')),
```

This will include the URLs defined in `reviews.urls` into the application namespace '`reviews`', with the instance namespace '`author-reviews`'.

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 3-tuple containing:

```
(<list of url() instances>, <application namespace>, <instance namespace>)
```

For example:

```
from django.conf.urls import include, url  
  
from . import views  
  
  
reviews_patterns = [  
  
    url(r'^$', views.IndexView.as_view(), name='index'),  
  
    url(r'^(?P<pk>\d+)/$', views.DetailView.as_view(), name='detail'),  
  
]  
  
  
url(r'^reviews/', include((reviews_patterns, 'reviews', 'author-reviews'))),
```

This will include the nominated URL patterns into the given application and instance namespace.

For example, the Django admin is deployed as instances of `AdminSite`. `AdminSite` objects have a `urls` attribute: A 3-tuple that contains all the patterns in the corresponding admin site, plus the application namespace '`admin`', and the name of the admin instance. It is this `urls` attribute that you `include()` into your projects `urlpatterns` when you deploy an admin instance.

Be sure to pass a tuple to `include()`. If you simply pass three arguments: `include(reviews_patterns, 'reviews', 'author-reviews')`, Django won't throw an error but due to the signature of `include()`, '`reviews`' will be the instance namespace and '`author-reviews`' will be the application namespace instead of vice versa.

WHAT'S NEXT?

This chapter has provided many advanced tips and tricks for views and URLconfs. Next, in Chapter 8, we'll give this advanced treatment to [Django's template system](#).

CHAPTER 8: ADVANCED TEMPLATES

Although most of your interactions with Django's template language will be in the role of template author, you may want to customize and extend the template engine – either to make it do something it doesn't already do, or to make your job easier in some other way.

This chapter delves deep into the guts of Django's template system. It covers what you need to know if you plan to extend the system or if you're just curious about how it works. It also covers the auto-escaping feature, a security measure you'll no doubt notice over time as you continue to use Django.

TEMPLATE LANGUAGE REVIEW

First, let's quickly review a number of terms introduced in [Chapter 3](#):

- A *template* is a text document, or a normal Python string, that is marked up using the Django template language. A template can contain template tags and variables.
- A *template tag* is a symbol within a template that does something. This definition is deliberately vague. For example, a template tag can produce content, serve as a control structure (an `if` statement or `for` loop), grab content from a database, or enable access to other template tags.

Template tags are surrounded by `{%` and `%}`:

```
{% if is_logged_in %}  
    Thanks for logging in!  
{% else %}  
    Please log in.  
{% endif %}
```

- A *variable* is a symbol within a template that outputs a value.

Variable tags are surrounded by `{{` and `}`:

```
My first name is {{ first_name }}. My last name is {{ last_name }}.
```

- A *context* is a `name->value` mapping (similar to a Python dictionary) that is passed to a template.
- A template *renders* a context by replacing the variable “holes” with values from the context and executing all template tags.

For more details about the basics of these terms, refer back to [Chapter 3](#).

The rest of this chapter discusses ways of extending the template engine. First, though, let’s take a quick look at a few internals left out of Chapter 3 for simplicity.

REQUESTCONTEXT AND CONTEXT PROCESSORS

When rendering a template, you need a context. This can be an instance of `django.template.Context`, but Django also comes with a subclass, `django.template.RequestContext`, that acts slightly differently. `RequestContext` adds a bunch of variables to your template context by default – things like the `HttpRequest` object or information about the currently logged-in user. The `render()` shortcut creates a `RequestContext` unless it is passed a different context instance explicitly.

Use `RequestContext` when you don’t want to have to specify the same set of variables in a series of templates. For example, consider these two views:

```
from django.template import loader, Context

def view_1(request):
    # ...
    t = loader.get_template('template1.html')
    c = Context({
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR'],
        'message': 'I am view 1.'
    })
```

```

    return t.render(c)

def view_2(request):
    # ...
    t = loader.get_template('template2.html')
    c = Context({
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR'],
        'message': 'I am the second view.'
    })
    return t.render(c)

```

(Note that we're deliberately *not* using the `render()` shortcut in these examples – we're manually loading the templates, constructing the context objects and rendering the templates. We're “spelling out” all of the steps for the purpose of clarity.)

Each view passes the same three variables – `app`, `user` and `ip_address` – to its template. Wouldn't it be nice if we could remove that redundancy?

`RequestContext` and **context processors** were created to solve this problem. Context processors let you specify a number of variables that get set in each context automatically – without you having to specify the variables in each `render()` call. The catch is that you have to use `RequestContext` instead of `Context` when you render a template.

The most low-level way of using context processors is to create some processors and pass them to `RequestContext`. Here's how the above example could be written with context processors:

```

from django.template import loader, RequestContext

def custom_proc(request):
    "A context processor that provides 'app', 'user' and 'ip_address'."
    return {

```

```

    'app': 'My app',
    'user': request.user,
    'ip_address': request.META['REMOTE_ADDR']
}

def view_1(request):
    # ...
    t = loader.get_template('template1.html')
    c = RequestContext(request, {'message': 'I am view 1.'},
                      processors=[custom_proc])
    return t.render(c)

def view_2(request):
    # ...
    t = loader.get_template('template2.html')
    c = RequestContext(request, {'message': 'I am the second view.'},
                      processors=[custom_proc])
    return t.render(c)

```

Let's step through this code:

- First, we define a function `custom_proc`. This is a context processor – it takes an `HttpRequest` object and returns a dictionary of variables to use in the template context. That's all it does.
- We've changed the two view functions to use `RequestContext` instead of `Context`. There are two differences in how the context is constructed. One, `RequestContext` requires the first argument to be an `HttpRequest` object – the one that was passed into the view function in the first place (`request`). Two, `RequestContext` takes an optional `processors` argument, which is a list or tuple of context processor functions to use. Here, we pass in `custom_proc`, the custom processor we defined above.

- Each view no longer has to include `app`, `user` or `ip_address` in its context construction, because those are provided by `custom_proc`.
- Each view *still* has the flexibility to introduce any custom template variables it might need. In this example, the `message` template variable is set differently in each view.

In Chapter 3, we introduced the `render()` shortcut, which saves you from having to call `loader.get_template()`, then create a `Context`, then call the `render()` method on the template. In order to demonstrate the lower-level workings of context processors, the above examples didn't use `render()`. But it's possible – and preferable – to use context processors with `render()`. Do this with the `context_instance` argument, like so:

```
from django.shortcuts import render

from django.template import RequestContext

def custom_proc(request):
    "A context processor that provides 'app', 'user' and 'ip_address'."
    return {
        'app': 'My app',
        'user': request.user,
        'ip_address': request.META['REMOTE_ADDR']
    }

def view_1(request):
    #
    return render(request, 'template1.html',
                  {'message': 'I am view 1.'},
                  context_instance=RequestContext(request, processors=[custom_proc]))

def view_2(request):
    #
    return render(request, 'template2.html',
```

```
{'message': 'I am the second view.'},  
context_instance=RequestContext(request, processors=[custom_proc]))
```

Here, we've trimmed down each view's template rendering code to a single (wrapped) line.

This is an improvement, but, evaluating the conciseness of this code, we have to admit we're now almost overdosing on the *other* end of the spectrum. We've removed redundancy in data (our template variables) at the cost of adding redundancy in code (in the `processors` call). Using context processors doesn't save you much typing if you have to type `processors` all the time.

For that reason, Django provides support for *global* context processors. The `context_processors` setting (in your `settings.py`) designates which context processors should *always* be applied to `RequestContext`. This removes the need to specify `processors` each time you use `RequestContext`.

By default, `context_processors` is set to 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',  
],
```

This setting is a list of callables that use the same interface as our `custom_proc` function above – functions that take a `request` object as their argument and return a dictionary of items to be merged into the context. Note that the values in `context_processors` are specified as *strings*, which means the processors are required to be somewhere on your Python path (so you can refer to them from the setting).

Each processor is applied in order. That is, 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.

Django provides a number of simple context processors, including the ones that are enabled by default:

AUTH

```
django.contrib.auth.context_processors.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.

DEBUG

```
django.template.context_processors.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 query and lazily generated on access.

i18N

```
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.

MEDIA

```
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.

STATIC

```
django.template.context_processors.static
```

If this processor is enabled, every `RequestContext` will contain a variable `STATIC_URL`, providing the value of the `STATIC_URL` setting.

CSRF

```
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](#) (see chapter 21).

REQUEST

```
django.template.context_processors.request
```

If this processor is enabled, every `RequestContext` will contain a variable `request`, which is the current `HttpRequest`.

MESSAGES

```
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 (see Appendix H).
- `DEFAULT_MESSAGE_LEVELS` – A mapping of the message level names to their numeric value.

GUIDELINES FOR WRITING YOUR OWN CONTEXT PROCESSORS

A context processor has a very simple interface: It's just 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.

Here are a few tips for rolling your own:

- Make each context processor responsible for the smallest subset of functionality possible. It's easy to use multiple processors, so you might as well split functionality into logical pieces for future reuse.
- Keep in mind that any context processor in `TEMPLATE_CONTEXT_PROCESSORS` will be available in *every* template powered by that settings file, so try to pick variable names that are unlikely to conflict with variable names your templates might be using

independently. As variable names are case-sensitive, it's not a bad idea to use all caps for variables that a processor provides.

- 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. With that said, the convention is to save them in a file called `context_processors.py` within your app or project.

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 his 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. (For more on security, see [Chapter 21](#).)

To avoid this problem, you have two options:

- One, you can make sure to run each untrusted variable through the `escape` filter, which converts potentially harmful HTML characters to harmless 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 trusted 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 e-mail 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 '``', 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 on to 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 & 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 `Hello!`:

```

<h1>This & that</h1>

<b>Hello!</b>

```

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 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.

AUTOMATIC ESCAPING OF STRING LITERALS IN FILTER ARGUMENTS

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 < 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.

INSIDE TEMPLATE LOADING

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; the most commonly used of them, the `apps` loader, is described below.

FILESYSTEM LOADER

```
class filesystem.Loader Loads templates from the filesystem, according to DIRS  
<TEMPLATES-DIRS>.
```

This loader is enabled by default. However it won't find any templates until you set `DIRS <TEMPLATES-DIRS>` to a non-empty list:

```
TEMPLATES = [{  
  
    'BACKEND': 'django.template.backends.django.DjangoTemplates',  
  
    'DIRS': [os.path.join(BASE_DIR, 'templates')],  
  
}]
```

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.reviews', 'myproject.music']
```

...then `get_template('foo.html')` will look for `foo.html` in these directories, in this order:

- `/path/to/myproject/reviews/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.reviewss`. You must then make sure that your `myproject.reviews` 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,  
}]
```

OTHER LOADERS

The remaining template loaders are:

- `django.template.loaders.eggs.Loader`
- `django.template.loaders.cached.Loader`
- `django.template.loaders.locmem.Loader`

These 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`. Details on these advanced loaders, as well as building your own custom loader, can be found on the Django Project website.

EXTENDING THE TEMPLATE SYSTEM

Now that you understand a bit more about the internals of the template system, let's look at how to extend the system with custom code.

Most template customization comes in the form of custom template tags and/or filters. Although the Django template language comes with many built-in tags and filters, you'll probably assemble your own libraries of tags and filters that fit your own needs. Fortunately, it's quite easy to define your own functionality.

CODE LAYOUT

Custom template tags and filters must live inside a Django app. If they relate to an existing app it makes sense to bundle them there; otherwise, you should create a new app to hold them.

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. After adding this 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 `review_extras.py`, your app layout might look like this:

```
reviews/
    __init__.py
    models.py
    templatetags/
        __init__.py
        review_extras.py
    views.py
```

And in your template you would use the following:

```
{% load review_extras %}
```

The app that contains the custom tags must be in `INSTALLED_APPS` in order for the `{% load %}<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 %}<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()
```

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.

CREATING A TEMPLATE LIBRARY

Whether you're writing custom tags or filters, the first thing to do is to create a **template library** – a small bit of infrastructure Django can hook into.

Creating a template library is a two-step process:

- First, decide which Django application should house the template library. If you've created an app via `manage.py startapp`, you can put it in there, or you can create another app solely for the template library. We'd recommend the latter, because your filters might be useful to you in future projects.

Whichever route you take, make sure to add the app to your `INSTALLED_APPS` setting. We'll explain this shortly.

- Second, create a `templatetags` directory in the appropriate Django application's package. It should be on the same level as `models.py`, `views.py`, and so forth. For example:

```
books/
    __init__.py
    models.py
    templatetags/
        views.py
```

Create two empty files in the `templatetags` directory: an `__init__.py` file (to indicate to Python that this is a package containing Python code) and a file that will contain your custom tag/filter definitions. The name of the latter file is what you'll use to load the tags later. For example, if your custom tags/filters are in a file called `review_extras.py`, you'd write the following in a template:

```
{% load review_extras %}
```

The `{% load %}` tag looks at your `INSTALLED_APPS` setting and only allows the loading of template libraries within installed Django applications. This is a security feature; it allows you to host Python code for many template libraries on a single computer without enabling access to all of them for every Django installation.

If you write a template library that isn't tied to any particular models/views, it's valid and quite normal to have a Django application package that contains only a `templatetags` package. 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 application.

Once you've created that Python module, you'll just have to write a bit of Python code, depending on whether you're writing filters or tags.

To be a valid tag library, the module must contain a module-level variable named `register` that is an instance of `template.Library`. This is the data structure in which all the tags and filters are registered. So, near the top of your module, insert the following:

```
from django import template
register = template.Library()
```

Note

For a fine selection 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. Some applications in `django.contrib` also contain template libraries.

Once you've created this `register` variable, you'll use it to create template filters and tags.

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 %}<load>` tag.

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.
    ...

```

- **Strings marked as “needing escaping”** are *always* escaped on output, regardless of whether they are in an `autoescape` block or not. These strings are only escaped once, however, even if auto-escaping applies.

Internally, these strings are of type `EscapeBytes` or `EscapeText`. Generally you don’t have to worry about these; they exist for the implementation of the `escape` filter.

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 than 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.

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=None):
    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.

Warning

Avoiding XSS vulnerabilities when reusing built-in filters

Be careful when reusing Django's built-in filters. You'll need to pass `autoescape=True` to the filter in order to get the proper autoescaping behavior and avoid a cross-site script vulnerability.

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
def urlize_and_linebreaks(text):
    return linebreaksbr(urlize(text, autoescape=True), autoescape=True)
```

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.

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.

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 %}
```

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 produces a list of books for a given `Author` object. We’ll use the tag like this:

```
{% books_for_author author %}
```

The result will be something like this:

```
<ul>
    <li>The Cat In The Hat</li>
```

```
<li>Hop On Pop</li>  
<li>Green Eggs And Ham</li>  
</ul>
```

First, we define the function that takes the argument and produces a dictionary of data for the result. Notice that we need to return only a dictionary, not anything more complex. This will be used as the context for the template fragment:

```
def books_for_author(author):  
    books = Book.objects.filter(authors__id=author.id)  
    return {'books': books}
```

Next, we create the template used to render the tag's output. Following our example, the template is very simple:

```
<ul>  
{% for book in books %}  
    <li>{{ book.title }}</li>  
{% endfor %}  
</ul>
```

Finally, we create and register the inclusion tag by calling the `inclusion_tag()` method on a `Library` object.

Following our example, if the preceding template is in a file called `book_snippet.html` in a directory that's searched by the template loader, we register the tag like this:

```
# Here, register is a django.template.Library instance, as before  
@register.inclusion_tag('book_snippet.html')  
  
def show_reviews(review):  
    ...
```

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('book_snippet.html')
```

```
register.inclusion_tag(t) (show_reviews)
```

...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 an inclusion 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`.)

The template `link.html` might contain the following:

```
Jump directly to <a href="{{ link }}>{{ title }}</a>.
```

Then, anytime 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()
```

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 where `DEBUG` and `TEMPLATE_DEBUG` are `False`. In some cases however, particularly if `TEMPLATE_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. 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 %}<autoescape>` 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, cylevars):
        self.cycle_iter = itertools.cycle(cylevars)

    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, cylevars):
        self.cylevars = cylevars

    def render(self, context):
        if self not in context.render_context:

```

```
        context.render_context[self] = itertools.cycle(self.cylevars)

    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 `cylevars` 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`.

Note

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, you might want to consider using the assignment tag shortcut we introduced above.

PARSING UNTIL ANOTHER BLOCK TAG

Template tags can work in tandem. For instance, the standard `{% comment %}<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 ''

```

Note

The actual implementation of `{% comment %}<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 %}<for>` in `django/template/defaulttags.py` and `{% if %}<if>` in `django/template/smartyif.py`.

WHAT'S NEXT

Continuing this section's theme of advanced topics, the next chapter covers advanced usage of [Django models](#).

CHAPTER 9: ADVANCED MODELS

In Chapter 4, we presented an introduction to [Django's database layer](#) – how to define models and how to use the database API to create, retrieve, update and delete records. In this chapter, we'll introduce you to some more advanced features of this part of Django.

RELATED OBJECTS

Recall our book models from Chapter 4:

```
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()

    def __str__(self):
        return self.name

class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
```

```

email = models.EmailField()

def __str__(self):
    return '%s %s' % (self.first_name, self.last_name)

class Book(models.Model):

    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()

    def __str__(self):
        return self.title

```

As we explained in [Chapter 4](#), accessing the value for a particular field on a database object is as straightforward as using an attribute. For example, to determine the title of the book with ID 50, we'd do the following:

```

>>> from mysite.books.models import Book
>>> b = Book.objects.get(id=50)
>>> b.title
'The Django Book'

```

But one thing we didn't mention previously is that related objects – fields expressed as either a `ForeignKey` or `ManyToManyField` – act slightly differently.

ACCESSING FOREIGN KEY VALUES

When you access a field that's a `ForeignKey`, you'll get the related model object. For example:

```

>>> b = Book.objects.get(id=50)
>>> b.publisher
<Publisher: Apress Publishing>

```

```
>>> b.publisher.website  
'http://www.apress.com/'
```

With `ForeignKey` fields, it works the other way, too, but it's slightly different due to the non-symmetrical nature of the relationship. To get a list of books for a given publisher, use `publisher.book_set.all()`, like this:

```
>>> p = Publisher.objects.get(name='Apress Publishing')  
>>> p.book_set.all()  
[<Book: The Django Book>, <Book: Dive Into Python>, ...]
```

Behind the scenes, `book_set` is just a `QuerySet` (as covered in Chapter 4), and it can be filtered and sliced like any other `QuerySet`. For example:

```
>>> p = Publisher.objects.get(name='Apress Publishing')  
>>> p.book_set.filter(title__icontains='django')  
[<Book: The Django Book>, <Book: Pro Django>]
```

The attribute name `book_set` is generated by appending the lower case model name to `_set`.

ACCESSING MANY-TO-MANY VALUES

Many-to-many values work like foreign-key values, except we deal with `QuerySet` values instead of model instances. For example, here's how to view the authors for a book:

```
>>> b = Book.objects.get(id=50)  
>>> b.authors.all()  
[<Author: Adrian Holovaty>, <Author: Jacob Kaplan-Moss>]  
>>> b.authors.filter(first_name='Adrian')  
[<Author: Adrian Holovaty>]  
>>> b.authors.filter(first_name='Adam')  
[]
```

It works in reverse, too. To view all of the books for an author, use `author.book_set`, like this:

```
>>> a = Author.objects.get(first_name='Adrian', last_name='Holovaty')
```

```
>>> a.book_set.all()  
[<Book: The Django Book>, <Book: Adrian's Other Book>]
```

Here, as with `ForeignKey` fields, the attribute name `book_set` is generated by appending the lower case model name to `_set`.

MANAGERS

In the statement `Book.objects.all()`, `objects` is a special attribute through which you query your database. In Chapter 4, we briefly identified this as the model’s *manager*. Now it’s time to dive a bit deeper into what managers are and how you can use them.

In short, a model’s manager is an object through which Django models perform database queries. Each Django model has at least one manager, and you can create custom managers in order to customize database access.

There are two reasons you might want to create a custom 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, which are explained later in this chapter.)

For example, let’s give our `Book` model a manager method `title_count()` that takes a keyword and returns the number of books that have a title containing that keyword. (This example is slightly contrived, but it demonstrates how managers work.)

```
# models.py  
  
from django.db import models  
  
# ... Author and Publisher models here ...  
  
class BookManager(models.Manager):  
    def title_count(self, keyword):
```

```

        return self.filter(title__icontains=keyword).count()

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
    num_pages = models.IntegerField(blank=True, null=True)
    objects = BookManager()

    def __str__(self):
        return self.title

```

With this manager in place, we can now do this:

```

>>> Book.objects.title_count('django')
4
>>> Book.objects.title_count('python')
18

```

Here are some notes about the code:

- We've created a `BookManager` class that extends `django.db.models.Manager`. This has a single method, `title_count()`, which does the calculation. Note that the method uses `self.filter()`, where `self` refers to the manager itself.
- We've assigned `BookManager()` to the `objects` attribute on the model. This has the effect of replacing the "default" manager for the model, which is called `objects` and is automatically created if you don't specify a custom manager. We call it `objects` rather than something else, so as to be consistent with automatically created managers.

Why would we want to add a method such as `title_count()`? To encapsulate commonly executed queries so that we don't have to duplicate code.

MODIFYING INITIAL MANAGER QUERYSETS

A manager's base QuerySet returns all objects in the system. For example, Book.objects.all() returns all books in the book 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.

```
from django.db import models

# 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. Note that we explicitly set objects to a vanilla Manager instance, because if we hadn't, the only available manager would be dahl_objects.

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 MaleManager(models.Manager):  
  
    def get_queryset(self):  
  
        return super(MaleManager, self).get_queryset().filter(sex='M')  
  
  
class FemaleManager(models.Manager):  
  
    def get_queryset(self):  
  
        return super(FemaleManager, self).get_queryset().filter(sex='F')  
  
  
class Person(models.Model):  
  
    first_name = models.CharField(max_length=50)  
  
    last_name = models.CharField(max_length=50)  
  
    sex = models.CharField(max_length=1, choices=((('M', 'Male'), ('F', 'Female'))))  
  
    people = models.Manager()  
  
    men = MaleManager()  
  
    women = FemaleManager()
```

This example allows you to request `Person.men.all()`, `Person.women.all()`, and `Person.people.all()`, yielding predictable results.

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 this first `Manager` defined in a class as the “default” `Manager`, and several parts of Django (though not the admin application) will use that `Manager` exclusively for that model. As a result, it's

often a good idea to be careful in your choice of default manager, in order to avoid a situation where overriding of `get_queryset()` results in an inability to retrieve objects you'd like to work with.

MODEL METHODS

Define custom methods on a model to add custom “row-level” functionality to your objects. Whereas managers 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.

An example is the easiest way to explain this. Here’s a model with 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 [model instance reference](#) in Appendix A has a complete list of methods automatically given to each model. You can override most of these – see below – but there are a couple that you’ll almost always want to define:

```
__str__()
```

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.

```
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`. 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 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](#).

Warning

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.

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'd 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.

Warning

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.

Warning

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.

Warning

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 %.2f." % (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.

Note

Dictionary params are not supported with the SQLite backend; with this backend, you must pass parameters as a list.

Warning

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):
    cursor = connection.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 cost, you can return results as a `dict` by using something like this:

```
def dictfetchall(cursor):  
  
    "Returns all rows from a cursor as a dict"  
  
    desc = cursor.description  
  
    return [  
  
        dict(zip([col[0] for col in desc], row))  
  
        for row in cursor.fetchall()  
  
    ]
```

Here is an example of the difference between the two:

```
>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2");  
  
>>> cursor.fetchall()  
  
((54360982L, None), (54360880L, None))  
  
  
>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2");  
  
>>> dictfetchall(cursor)  
  
[{'parent_id': None, 'id': 54360982L}, {'parent_id': None, 'id': 54360880L}]
```

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()
```

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
        cursor = connection.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)
    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.

WHAT'S NEXT?

In the next chapter, we'll show you Django's "[generic views](#)" framework, which lets you save time in building Web sites that follow common patterns.

CHAPTER 10: GENERIC VIEWS

Here again is a recurring theme of this book: at its worst, Web development is boring and monotonous. So far, we've covered how 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 display 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 when displaying database data in views.

Finally, display views are only one part of Django’s comprehensive class-based view system. For a full introduction and detail description of the other [class-based views](#) Django provides, see Appendix C.

GENERIC VIEWS OF OBJECTS

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):
    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):
    return self.name

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField('Author')
    publisher = models.ForeignKey(Publisher)
    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.

Note

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. Appendix C 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
```

Note

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.

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.

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.

Note

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.

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
```

Note

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`.

WHAT'S NEXT?

In this chapter we looked at only a couple of the generic views Django ships with, but the general ideas presented here should apply pretty closely to any generic view. [Appendix C](#) covers all the available views in detail, and it's recommended reading if you want to get the most out of this powerful feature.

This concludes the section of this book devoted to advanced usage of models, templates and views. The following chapters cover a range of functions that are very common in modern

commercial websites. We will start with a subject essential to building interactive websites – [user management](#).

CHAPTER 11: USER AUTHENTICATION IN DJANGO

A significant percentage of modern, interactive websites allow some form of user interaction – from allowing simple comments on a blog, to full editorial control of articles on a news site. If a site offers any sort of eCommerce, authentication and authorization of paying customers is essential.

Just managing users – lost usernames, forgotten passwords and keeping information up to date – can be a real pain. As a programmer, writing an authentication system can be even worse.

Lucky for us, Django comes with a user authentication system installed automatically for your convenience when you ran `django-admin startproject`.

Django provides a default implementation for managing user accounts, groups, permissions and cookie-based user sessions out of the box.

Like most things in Django, the default implementation is fully extendible and customizable to suit your project's needs. So let's jump right in.

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)

USING THE DJANGO AUTHENTICATION SYSTEM

Django's authentication system in its default 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 also supports extensive extension and customization of authentication.

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

CREATING USERS

The simplest, and least error prone way to create and manage users is through the Django admin. Django also provides built in views and forms to allow users to log in and out and change their own password.

We will be looking at user management via the admin and generic user forms a bit later in this chapter, but first, lets look at how we would handle user authentication directly.

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()
```

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 the `--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. 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 two options:

`manage.py changepassword *username* <changepassword>` 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 of the user 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()
```

Changing a user's password will log out all their sessions if the `SessionAuthenticationMiddleware` is enabled.

AUTHENTICATING USERS

`authenticate(**credentials)` To authenticate a given username and password, use `authenticate()`. It takes credentials in the form of keyword arguments, for the default configuration this is `username` and `password`, and it returns a `User` object if the password is valid for the given username. If the password is invalid, `authenticate()` returns `None`. Example:

```
from django.contrib.auth import authenticate  
  
user = authenticate(username='john', password='secret')  
  
if user is not None:  
  
    # the password verified for the user  
  
    if user.is_active:  
  
        print("User is valid, active and authenticated")  
  
    else:  
  
        print("The password is valid, but the account has been  
disabled!")  
  
    else:  
  
        # the authentication system was unable to verify the username and  
        # password  
  
        print("The username and password were incorrect.")
```

Note

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.

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`.

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 `BookReview` model in `books`:

```
from books.models import BookReview
from django.contrib.auth.models import Group, Permission
from django.contrib.contenttypes.models import ContentType
```

```
content_type = ContentType.objects.get_for_model(BookReview)

permission = Permission.objects.create(codename='can_publish',
                                       name='Can Publish Reviews',
                                       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('books.change_bar')

    permission = Permission.objects.get(codename='change_bar')
    user.user_permissions.add(permission)

    # Checking the cached permission set
    user.has_perm('books.change_bar')  # False

    # Request new instance of User
    user = get_object_or_404(User, pk=user_id)

    # Permission cache is repopulated from the database
    user.has_perm('books.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()`

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)
```

```

if user.is_active:

    login(request, user)

    # Redirect to a success page.

else:

    # Return a 'disabled account' error message

else:

    # Return an 'invalid login' error message.

```

Calling authenticate() first

When you're manually logging a user in, you *must* call `authenticate()` before you call `login()`. `authenticate()` sets an attribute on the `User` noting which authentication backend successfully authenticated that user, and this information is needed later during the login process. An error will be raised if you try to login a user object retrieved from the database directly.

HOW TO LOG A USER OUT

`logout()`

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.shortcuts import redirect

def my_view(request):
    if not request.user.is_authenticated():
        return redirect('/login/?next=%s' % 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, 'books/login_error.html')
    # ...
```

THE LOGIN_REQUIRED DECORATOR

```
django.contrib.auth.decorators.login_required([redirect_field_name=REDIRECT_FIELD_NAME, 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 `LOGIN_URL`, passing the current absolute path in the query string. Example: `/accounts/login/?next=/reviews/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 `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 `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.

Note

The `login_required` decorator does NOT check the `is_active` flag on a user.

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:

```
def my_view(request):
    if not request.user.email.endswith('@example.com'):
        return HttpResponseRedirect("You can't leave a review for this book.")

    #
    django.contrib.auth.decorators.user_passes_test(func[,           login_url=None,
redirect_field_name=REDIRECT_FIELD_NAME])
```

As a shortcut, you can use the convenient `user_passes_test` decorator:

```
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 `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):
    ...
```

THE PERMISSION_REQUIRED DECORATOR

```
django.contrib.auth.decorators.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('reviews.can_vote')
```

```
def my_view(request):
    ...
```

Just like the `has_perm()` method, permission names take the form "`<app label>.<permission codename>`" (i.e. `reviews.can_vote` for a permission on a model in the `reviews` application).

The decorator may also take a list of permissions.

Note that `permission_required()` also takes an optional `login_url` parameter. Example:

```
from django.contrib.auth.decorators import permission_required
```

```
@permission_required('reviews.can_vote', login_url='/loginpage/')

def my_view(request):
    ...
```

As in the `login_required()` decorator, `login_url` defaults to `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.

APPLYING PERMISSIONS TO GENERIC VIEWS

To apply a permission to a class-based generic view , decorate the `View.dispatch` method on the class. Another approach is to write a mixin that wraps `as_view()`.

SESSION INVALIDATION ON PASSWORD CHANGE

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. If the `SessionAuthenticationMiddleware` is enabled, Django verifies that the hash sent along with each request matches the one that's computed server-side. This allows a user to log out all of their sessions by changing their password.

The default password change views included with Django, `django.contrib.auth.views.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 this function:

```
django.contrib.auth.decorators.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:

        ...

```

Note

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 built-in auth forms but you can pass in your own forms as well.

Django provides no default template for the authentication views – however the template context is documented for each view below.

The built-in views all return a `TemplateResponse` instance, which allows you to easily customize the response data before rendering. Most built-in authentication views provide a URL name for easier reference.

LOGIN

```
django.contrib.auth.views.login(request[, template_name, redirect_field_name,  
authentication_form, current_app, extra_context])
```

Logs a user in.

Default URL: /login/

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.

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 `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']`.

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 `books/login.html` instead:

```
url(r'^accounts/login/$', auth_views.login, {'template_name': 'books/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 %}

<form method="post" action="{% url 'django.contrib.auth.views.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>
```

```
{% endblock %}
```

If you have customized 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

```
django.contrib.auth.views.logout([request],      next_page,      template_name,  
redirect_field_name, current_app, extra_context])
```

Logs a user out.

Default URL: /logout/

Optional arguments:

- `next_page`: The URL to redirect to after logout.
- `template_name`: The full name of a template to display after logging the user out. Defaults to `registration/logout.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.

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']`.
- `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

```
django.contrib.auth.views.logout_then_login(request[, login_url, current_app,  
extra_context])
```

Logs a user out, then redirects to the login page.

Default URL: None provided.

Optional arguments:

- `login_url`: The URL of the login page to redirect to. Defaults to `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.

PASSWORD_CHANGE

```
django.contrib.auth.views.password_change(request[, template_name,  
post_change_redirect, password_change_form, current_app, extra_context])
```

Allows a user to change their password.

Default URL: /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.

Template context:

- `form`: The password change form (see `password_change_form` above).

PASSWORD_CHANGE_DONE

```
django.contrib.auth.views.password_change_done(request[, template_name,  
current_app, extra_context])
```

The page shown after a user has changed their password.

Default URL: /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.

PASSWORD_RESET

```
django.contrib.auth.views.password_reset(request[, template_name,  
email_template_name, password_reset_form, token_generator, post_reset_redirect,  
from_email, current_app, extra_context, html_email_template_name])
```

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.

Default URL: /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.

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']`.
- `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

```
django.contrib.auth.views.password_reset_done(request[, template_name,  
current_app, extra_context])
```

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.

Default URL: /password_reset_done/

Note

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.

PASSWORD_RESET_CONFIRM

```
django.contrib.auth.views.password_reset_confirm(request[, uidb64, token,  
template_name, token_generator, set_password_form, post_reset_redirect, current_app,  
extra_context])
```

Presents a form for entering a new password.

Default URL: /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.

PASSWORD_RESET_COMPLETE

`django.contrib.auth.views.password_reset_complete(request[, template_name, current_app, extra_context])`

Presents a view which informs the user that the password has been successfully changed.

Default URL: /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.

THE REDIRECT_TO_LOGIN HELPER FUNCTION

```
django.contrib.auth.views.redirect_to_login(next[, login_url, redirect_field_name])
```

Django provides a convenient function, `redirect_to_login` that can be used in a view for implementing custom access control. It 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 `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`:

Note

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.

ADMINPASSWORDCHANGEFORM

A form used in the admin interface to change a user's password.

Takes the `user` as the first positional argument.

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.

```
django.contrib.auth.views.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
```

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',
            )
```

PASSWORDCHANGEFORM

A form for allowing a user to change their password.

PASSWORDRESETFORM

A form for generating and emailing a one-time use link to reset a user's password.

```
django.contrib.auth.views.send_email(subject_template_name,  
email_template_name, context, from_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.

SETPASSWORDFORM

A form that lets a user change their password without entering the old password.

USERCHANGEFORM

A form used in the admin interface to change a user's information and permissions.

USERCREATIONFORM

A form for creating a new user.

AUTHENTICATION DATA IN TEMPLATES

The currently logged-in user and their permissions are made available in the template context when you use `RequestContext`.

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.

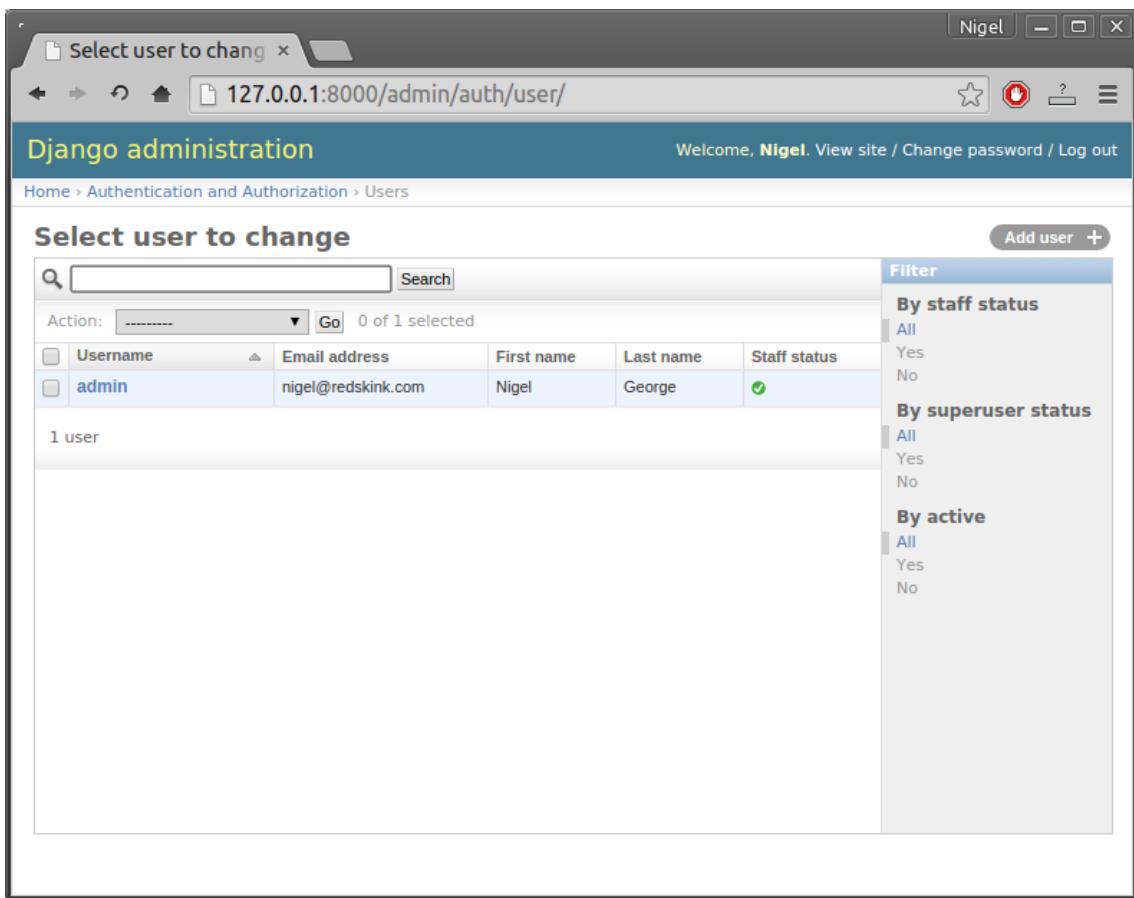


Figure 11-1: Django admin user management screen

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.

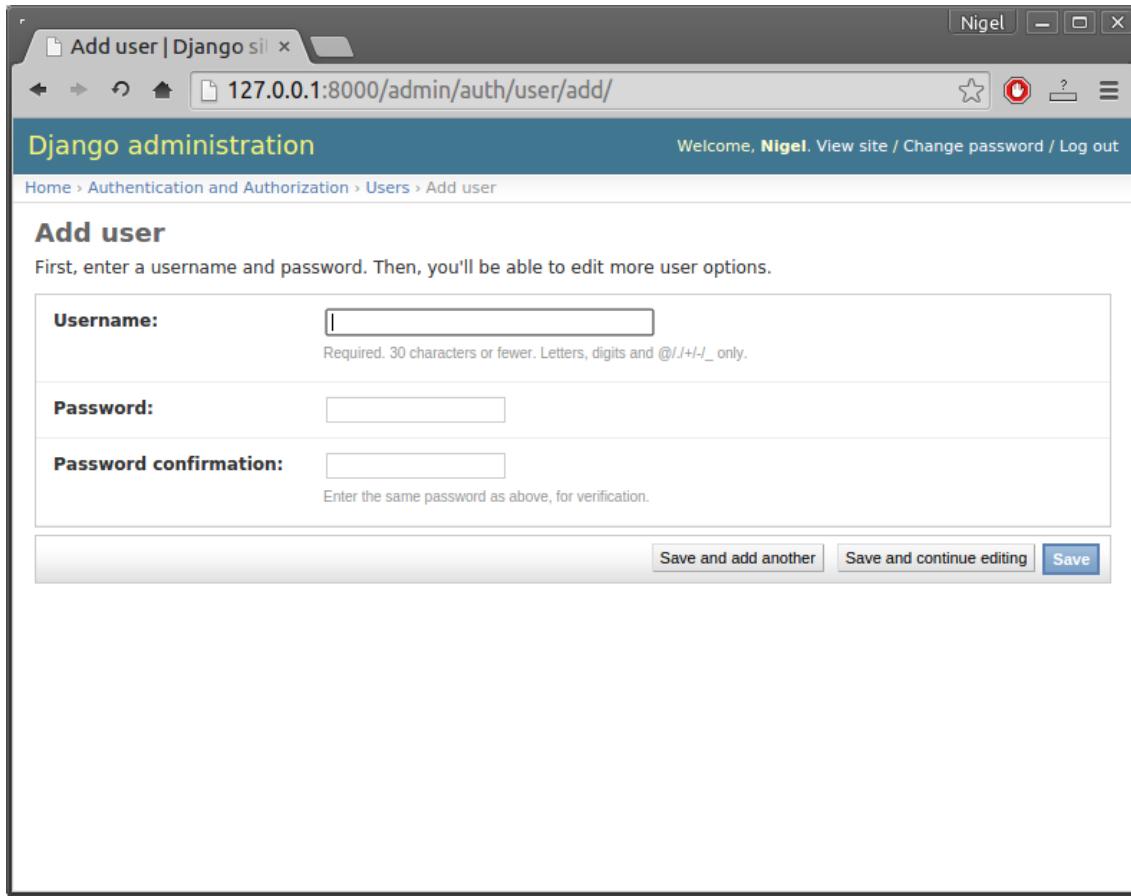


Figure 11-2: Django admin add user screen

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.

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.

Django administration

Welcome, **Nigel**. View site / Change password / Log out

Home > Authentication and Authorization > Users > admin

Change user

History

Username:	admin
Required. 30 characters or fewer. Letters, digits and @/_/+/-/_ only.	
Password:	algorithm: pbkdf2_sha256 iterations: 20000 salt: cZYq6v***** hash: eFNqwY***** Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using this form .
Personal info	
First name:	Nigel
Last name:	George

Figure 11-3: Link to change password (circled)

Change password: a x Nigel

127.0.0.1:8000/admin/auth/user/1/password/

Django administration

Welcome, **Nigel**. View site / Change password / Log out

Home > Authentication and Authorization > Users > admin > Change password

Change password: admin

Enter a new password for the user **admin**.

Password:	<input type="text"/>
Password (again):	<input type="text"/>
Enter the same password as above, for verification.	
<input type="button" value="Change password"/>	

Figure 11-4: Django admin change password form

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.

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.BCryptSHA256PasswordHasher',
'django.contrib.auth.hashers.BCryptPasswordHasher',
'django.contrib.auth.hashers.SHA1PasswordHasher',
'django.contrib.auth.hashers.MD5PasswordHasher',
'django.contrib.auth.hashers.CryptPasswordHasher',
]

```

This means that Django will use [PBKDF2](#) to store all passwords, but will support checking passwords stored with PBKDF2SHA1, [bcrypt](#), [SHA1](#), etc. The next few sections describe a couple of common ways advanced users may want to modify this setting.

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]`, or by downloading the library and installing it with `python setup.py install`.
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.SHA1PasswordHasher',
    'django.contrib.auth.hashers.MD5PasswordHasher',
    'django.contrib.auth.hashers.CryptPasswordHasher',
]

```

]

(You need to keep the other entries in this list, or else Django won't be able to upgrade passwords; see below).

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$$2a12NT0I31Sa7ihGEWpka9ASYrEFkhutTNeBQ2xfZskIiiJeyFXhRgS.Sy.`

INCREASING THE WORK FACTOR

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.BCryptSHA256PasswordHasher',
    'django.contrib.auth.hashers.BCryptPasswordHasher',
    'django.contrib.auth.hashers.SHA1PasswordHasher',
    'django.contrib.auth.hashers.MD5PasswordHasher',
    'django.contrib.auth.hashers.CryptPasswordHasher',
]

```

That's it – now your Django install will use more iterations when it stores passwords using PBKDF2.

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. Passwords will be upgraded when changing the PBKDF2 iteration count.

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.

`django.contrib.auth.hashers.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.

`django.contrib.auth.hashers.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). Currently supported algorithms are: `'pbkdf2_sha256'`, `'pbkdf2_sha1'`, `'bcrypt_sha256'`, `'bcrypt'`, `'sha1'`, `'md5'`, `'unsalted_md5'` (only for backward compatibility) and `'crypt'` if you have the `crypt` library installed. If the `password` argument is `None`, an unusable password is returned (a one that will be never accepted by `check_password()`).

`django.contrib.auth.hashers.is_password_usable(encoded_password)` Checks if the given string is a hashed password that has a chance of being verified against `check_password()`.

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.

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 the previous section on logging 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.

Note

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 system is tightly coupled to the Django `User` object described at the beginning of this document. For now, 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.models import User, check_password  
  
  
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 = 'sha1$4e987$afbcf42e21bd417fb71db8c66b321e9fc33051de'  
    """
```

```
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. Note that we can set password
        # to anything, because it won't be checked; the password
        # from settings.py will.

        user = User(username=username, password='get' from
settings.py')

        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):  
  
        if user_obj.username == settings.ADMIN_LOGIN:  
  
            return True  
  
        else:  
  
            return False
```

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 Web sites 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 one that is authenticated but has its attribute `is_active` set to `False`. However this does not mean they are not authorized to do anything. For example they are allowed to activate their account.

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.

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`. 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 one-to-one relationship 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)
    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
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(UserAdmin):
    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 = 'books.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.

Warning

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.

Warning

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)

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.

`django.contrib.auth.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)
```

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(signal, sender, instance, \*\*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. Your model must have an integer primary key.
2. 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.
3. 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).

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` supports `ForeignKeys`. 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']
```

Note

`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` supports `ForeignKeys`. 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()`.

The following methods are available on any subclass of `AbstractBaseUser`:

`class models.AbstractBaseUser get_username()` Returns the value of the field nominated by `USERNAME_FIELD`.

`is_anonymous()` Always returns `False`. This is a way of differentiating from `AnonymousUser` objects. Generally, you should prefer using `is_authenticated()` to this method.

`is_authenticated()` Always returns `True`. 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 – it only indicates that the user has provided a valid username and password.

`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 normalize_email(email)` A classmethod that 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='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ234  
56789') 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. `AbstractUser` provides the full implementation of the default `User` as an abstract model.

CUSTOM USERS AND THE BUILT-IN AUTH FORMS

As you may expect, Django's built in forms and views make certain assumptions about the user model that they are working with.

If your user model doesn't follow the same assumptions, it may be necessary to define a replacement form, and pass that form in as part of the configuration of the auth views.

- `UserCreationForm`

Depends on the `User` model. Must be re-written for any custom user model.

- `UserChangeForm`

Depends on the `User` model. Must be re-written for any custom user model.

- `AuthenticationForm`

Works with any subclass of `AbstractBaseUser`, and will adapt to use the field defined
in `USERNAME_FIELD`.

- `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.

- `SetPasswordForm`

Works with any subclass of `AbstractBaseUser`

- `PasswordChangeForm`

Works with any subclass of `AbstractBaseUser`

- `AdminPasswordChangeForm`

Works with any subclass of `AbstractBaseUser`

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 django.contrib.auth.is_staff` Returns True if the user is allowed to have access to the admin site.

`django.contrib.auth.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)`". 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`.

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.

CUSTOM USERS AND SIGNALS

Another limitation of custom User models is that you can't use `django.contrib.auth.get_user_model()` as the sender or target of a signal handler. Instead, you must register the handler with the resulting User model.

CUSTOM USERS AND TESTING/FIXTURES

If you are writing an application that interacts with the User model, you must take some precautions to ensure that your test suite will run regardless of the User model that is being used by a project. Any test that instantiates an instance of User will fail if the User model has been swapped out. This includes any attempt to create an instance of User with a fixture.

To ensure that your test suite will pass in any project configuration, `django.contrib.auth.tests.utils` defines a `@skipIfCustomUser` decorator. This decorator will cause a test case to be skipped if any User model other than the default Django user is in use. This decorator can be applied to a single test, or to an entire test class.

Depending on your application, tests may also be needed to be added to ensure that the application works with *any* user model, not just the default User model. To assist with this, Django provides two substitute user models that can be used in test suites:

`class tests.custom_user.CustomUser` A custom user model that uses an `email` field as the username, and has a basic admin-compliant permissions setup

`class tests.custom_user.ExtensionUser` A custom user model that extends `django.contrib.auth.models.AbstractUser`, adding a `date_of_birth` field.

You can then use the `@override_settings` decorator to make that test run with the custom User model. For example, here is a skeleton for a test that would test three possible User models – the default, plus the two User models provided by `auth` app:

```
from django.contrib.auth.tests.utils import skipIfCustomUser
from django.contrib.auth.tests.custom_user import CustomUser, ExtensionUser
from django.test import TestCase, override_settings

class ApplicationTestCase(TestCase):
    @skipIfCustomUser
    def test_normal_user(self):
        """Run tests for the normal user model"""
        self.assertSomething()

    @override_settings(AUTH_USER_MODEL='auth.CustomUser')
    def test_custom_user(self):
        """Run tests for a custom user model with email-based authentication"""
        self.assertSomething()

    @override_settings(AUTH_USER_MODEL='auth.ExtensionUser')
    def test_extension_user(self):
        """Run tests for a simple extension of the built-in User."""
        self.assertSomething()
```

A FULL EXAMPLE

A full example of an admin-compliant custom user app can be found on the [Django Project website](#).

CHAPTER 12 - TESTING IN DJANGO

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 we did earlier in the book, 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!

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.

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.

Warning

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 adding the `--keepdb` flag to the test command. 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.

By default the test databases get their names by prepending `test_` to the value of the `NAME` settings for the databases defined in `DATABASES`. When using the SQLite database engine the tests will by default use an in-memory database (i.e., the database will be created in memory, bypassing the filesystem entirely!). If you want to use a different database name, specify `NAME <TEST_NAME>` in the `TEST <DATABASE-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 <DATABASE-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_CHARSET>` `TEST` option. If you're using MySQL, you can also use the `COLLATION <TEST_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 a 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()`.

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.

Note

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 by passing `--reverse` to the `test` command. 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`.

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.assertEqual(future_poll.was_published_recently(), False)
AssertionError: True != 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

In recent versions of Django, the default password hasher is rather slow by design. If during your tests you are authenticating many users, 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.

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

'<!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('http://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 an 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 2616](#), which mandates that TRACE requests should not have an entity-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.

`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 string. 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'
```

`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. See [RFC 2616#section-10](#) for a full list of HTTP status codes.

`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'`.)

`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()
```

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)
```


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:

SIMPLETESTCASE

```
class SimpleTestCase
```

A thin subclass of `unittest.TestCase`, it extends it with some basic functionality like:

- Saving and restoring the Python warning machinery state.
- 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`.
- Custom test-time URL maps.

If you need any of the other more complex and heavyweight Django-specific features like:

- Testing or using the ORM.
- Database fixtures.
- Test skipping based on database backend features.

- The remaining specialized `assert*` methods.

then you should use `TransactionTestCase` or `TestCase` instead.

`SimpleTestCase` inherits from `unittest.TestCase`.

Warning

`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(cls, MyTestCase).setUpClass()          # Call parent first
        ...

    @classmethod
    def tearDownClass(cls):
        ...
        super(cls, MyTestCase).tearDownClass()      # Call parent last
```

TRANSACTIONTESTCASE

`class TransactionTestCase`

Django's `TestCase` class (described below) 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`.

In older versions of Django, the effects of transaction commit and rollback could not be tested within a `TestCase`. With the completion of the deprecation cycle of the old-style transaction

management in Django 1.8, transaction management commands (e.g. `transaction.commit()`) are no longer disabled within `TestCase`.

`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.

Warning

`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.

`TransactionTestCase` inherits from `SimpleTestCase`.

TESTCASE

`class TestCase`

This class provides some additional capabilities that can be useful for testing Web sites.

Converting a normal `unittest.TestCase` to a Django `TestCase` is easy: Just change the base class of your test from '`unittest.TestCase`' to '`django.test.TestCase`'. All of the standard Python unit test functionality will continue to be available, but it will be augmented with some useful additions, including:

- Automatic loading of fixtures.

- Wraps the tests within two nested `atomic` blocks: one for the whole class and one for each test.
- Creates a `TestClient` instance.
- Django-specific assertions for testing for things like redirection and form errors.

`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.

Warning

If you want to test some specific database transaction behavior, you should use `TransactionTestCase`, as `TestCase` wraps test execution within an `atomic()` block.

`TestCase` inherits from `TransactionTestCase`.

LIVE SERVER TEST CASE

```
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's address is '`localhost:8081`' and the full URL can be accessed during the tests with `self.live_server_url`. If you'd like to change the default address (in the case, for example, where the 8081 port is already taken) then you may pass a different one to the test command via the `--liveserver` option, for example:

```
./manage.py test --liveserver=localhost:8082
```

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`). The code for this test may look as follows:

```
from django.test import LiveServerTestCase

from selenium.webdriver.firefox.webdriver import WebDriver


class MySeleniumTests(LiveServerTestCase):

    fixtures = ['user-data.json']

    @classmethod
    def setUpClass(cls):
        super(MySeleniumTests, cls).setUpClass()
        cls.selenium = WebDriver()

    @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.

Tip

If you use the `staticfiles` app in your project and need to perform live testing, then you might want to use the `StaticLiveServerTestCase` subclass which transparently serves all the assets during execution of its 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`.

Note

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 Web site isn't much use if there isn't any data in the database. To make it easy to put test data into the database, Django's custom `TransactionTestCase` class provides a way of loading **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 case, 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.

This flush/load procedure is repeated for each test in the test case, so 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.

MULTI-DATABASE SUPPORT

`TestCase.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 testIndexPageView(self):
        call_some_test_code()
```

This test case will flush *all* the test databases before running `testIndexPageView`.

The `multi_db` flag also affects into which databases the attr:`TestCase.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

Warning

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_CLASSES={
            '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_CLASSES={
        '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_CLASSES={
```

```

    'append': 'django.middleware.cache.FetchFromCacheMiddleware',
    'prepend': 'django.middleware.cache.UpdateCacheMiddleware',
)

```

```

class MiddlewareTestCase(TestCase):

```

```

    def test_cache_middleware(self):
        response = self.client.get('/')
        # ...

```

Note

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()`.

Warning

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 use 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 an failure in your test suite.

`SimpleTestCase.assertRaisesMessage(expected_exception, expected_message, callable_obj=None, *args, **kwargs)` Asserts that execution of callable `callable_obj` raised the `expected_exception` exception and that such exception has an `expected_message` representation. Any other outcome is reported as a failure. Similar to unittest's `assertRaisesRegex()` with the difference that `expected_message` isn't a regular expression.

`SimpleTestCase.assertFieldOutput(fieldclass, valid, invalid, field_args=None, field_kwargs=None, empty_value="")` Asserts that a form field behaves correctly with various inputs.

Parameters:

- **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, host=None, 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.

The `host` argument sets a default host if `expected_url` doesn't include one (e.g. "/bar/"). If `expected_url` is an absolute URL that includes a host (e.g. "http://testhost/bar/"), the `host` parameter will be ignored. Note that the test client doesn't support fetching external URLs, but the parameter may be useful if you are testing with a custom HTTP host (for example, initializing the test client with `Client(HTTP_HOST="testhost")`).

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.

`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")
```

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.

```
django.test.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

``skipIfDBFeature`` can accept multiple feature strings.
```

```
django.test.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

``skipUnlessDBFeature`` can accept multiple feature strings.
```

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 django.test.runner.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.

`django.test.utils.setup_test_environment()` Performs any global pre-test setup, such as the installing the instrumentation of the template rendering system and setting up the dummy email outbox.

`django.test.utils.teardown_test_environment()` Performs any global post-test teardown, such as removing the black magic hooks into 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.

`django.db.connection.creation.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 <TEST_SERIALIZE>` entry in the `TEST`

`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.

`django.db.connection.creation.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.

CHAPTER 13: DEPLOYING DJANGO

This chapter covers the last essential step of building a Django application: deploying it to a production server.

If you've been following along with our ongoing examples, you've likely been using the `runserver`, which makes things very easy – with `runserver`, you don't have to worry about Web server setup. But `runserver` is intended only for development on your local machine, not for exposure on the public Web. To deploy your Django application, you'll need to hook it into an industrial-strength Web server such as Apache. In this chapter, we'll show you how to do that – but, first, we'll give you a checklist of things to do in your codebase before you go live.

PREPARING YOUR CODEBASE FOR PRODUCTION

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; and
- 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.

Some of the checks described below can be automated using the `--deploy` option of the `check` command. 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.

When we created a project in Chapter 1, the command `django-admin startproject` created a `settings.py` file with `DEBUG` set to `True`. Many internal parts of Django check this setting and change their behavior if `DEBUG` mode is on. For example, if `DEBUG` is set to `True`, then:

- All database queries will be saved in memory as the object `django.db.connection.queries`. As you can imagine, this eats up memory!

- Any 404 error will be rendered by Django's special 404 error page (covered in Chapter 3) rather than returning a proper 404 response. This page contains potentially sensitive information and should *not* be exposed to the public Internet.
- Any uncaught exception in your Django application – from basic Python syntax errors to database errors to template syntax errors – will be rendered by the Django pretty error page that you've likely come to know and love. This page contains even *more* sensitive information than the 404 page and should *never* be exposed to the public.

In short, setting `DEBUG` to `True` tells Django to assume only trusted developers are using your site. The Internet is full of untrustworthy hooligans, and the first thing you should do when you're preparing your application for deployment is set `DEBUG` to `False`.

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.

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.

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.

MEDIA_ROOT AND MEDIA_URL

Media files are uploaded by your users. They're untrusted! Make sure your web server never attempt 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.

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.

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:

- `http_not_found_view`
- `http_internal_server_error_view`
- `http_forbidden_view`
- `http_bad_request_view`

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 `WSGIPath` directive, with multiple paths separated by a colon (:) if using a UNIX-like system, or a semicolon (;) if using Windows. If any part of a directory path contains a space character, the complete argument string to `WSGIPath` must be quoted:

```
WSGIPath    /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 DIFFERENT SETTINGS FOR PRODUCTION

So far in this book, we've dealt with only a single settings file: the `settings.py` generated by `django-admin.py startproject`. But as you get ready to deploy, you'll likely find yourself needing multiple settings files to keep your development environment isolated from your production environment. (For example, you probably won't want to change `DEBUG` from `False` to `True` whenever you want to test code changes on your local machine.) Django makes this very easy by allowing you to use multiple settings files.

If you'd like to organize your settings files into production and development settings, you can accomplish this in one of three ways:

- Set up two full-blown, independent settings files.
- Set up a base settings file (say, for development) and a second (say, production) settings file that merely imports from the first one and defines whatever overrides it needs to define.
- Use only a single settings file that has Python logic to change the settings based on context.

We'll take these one at a time.

First, the most basic approach is to define two separate settings files. If you're following along, you've already got `settings.py`. Now, just make a copy of it called `settings_production.py`. (We made this name up; you can call it whatever you want.) In this new file, change `DEBUG`, etc.

The second approach is similar but cuts down on redundancy. Instead of having two settings files whose contents are mostly similar, you can treat one as the base file and create another file that imports from it. For example:

```
# settings.py

DEBUG = True
TEMPLATE_DEBUG = DEBUG

DATABASE_ENGINE = 'postgresql_psycopg2'
DATABASE_NAME = 'devdb'
DATABASE_USER = ''
DATABASE_PASSWORD = ''
DATABASE_PORT = ''

# ...

# settings_production.py

from settings import *

DEBUG = TEMPLATE_DEBUG = False
DATABASE_NAME = 'production'
DATABASE_USER = 'app'
DATABASE_PASSWORD = 'letmein'
```

Here, `settings_production.py` imports everything from `settings.py` and just redefines the settings that are particular to production. In this case, `DEBUG` is set to `False`, but we've also set different database access parameters for the production setting. (The latter goes to show that you can redefine *any* setting, not just the basic ones like `DEBUG`.)

Finally, the most concise way of accomplishing two settings environments is to use a single settings file that branches based on the environment. One way to do this is to check the current hostname. For example:

```
# settings.py

import socket

if socket.gethostname() == 'my-laptop':
    DEBUG = TEMPLATE_DEBUG = True
else:
    DEBUG = TEMPLATE_DEBUG = False

# ...
```

Here, we import the `socket` module from Python's standard library and use it to check the current system's hostname. We can check the hostname to determine whether the code is being run on the production server.

A core lesson here is that settings files are *just Python code*. They can import from other files, they can execute arbitrary logic, etc. Just make sure that, if you go down this road, the Python code in your settings files is bulletproof. If it raises any exceptions, Django will likely crash badly.

Renaming settings.py

Feel free to rename your `settings.py` to `settings_dev.py` or `settings/dev.py` or `foobar.py` – Django doesn't care, as long as you tell it what settings file you're using.

But if you *do* rename the `settings.py` file that is generated by `django-admin.py startproject`, you'll find that `manage.py` will give you an error message saying that it can't find the settings. That's because it tries to import a module called `settings`. You can fix this either by editing `manage.py` to change `settings` to the name of your module, or by using `django-admin.py` instead of `manage.py`. In the latter case, you'll need to set the `DJANGO_SETTINGS_MODULE` environment variable to the Python path to your settings file (e.g., `'mysite.settings'`).

DEPLOYING DJANGO TO A PRODUCTION SERVER

Headache free deployment

If you are serious about deploying a live website, there is really only one sensible option – find a host that explicitly supports Django. Not only will you get a separate media server out of the box (usually Nginx), but they will also take care of the little things like setting up Apache correctly and setting a cron job that restarts the Python process periodically (to prevent your site hanging up). With the better hosts, you are also likely to get some form of ‘one-click’ deployment. Save yourself the headache and pay the few bucks a month for a host who knows Django.

DEPLOYING 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 for the default contents you should put in this file, and what else you can add to it.

Warning

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.

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 `WSGIProcessGroup` 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 `WSGIProcessGroup`, for example:

```
WSGIProcessGroup example.com python-
path=/path/to/mysite.com:/path/to/venv/lib/python2.7/site-packages

WSGIProcessGroup 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 explicitly serves `robots.txt`, `favicon.ico`, any CSS file, 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.

IF YOU GET A UNICODEENCODEERROR

If you're taking advantage of the internationalization features of Django and you intend to allow users to upload files, you must ensure that the environment used to start Apache is configured to accept non-ASCII file names. If your environment is not correctly configured, you will trigger `UnicodeEncodeError` exceptions when calling functions like the ones in `os.path` on filenames that contain non-ASCII characters.

To avoid these problems, the environment used to start Apache should contain settings analogous to the following:

```
export LANG='en_US.UTF-8'  
export LC_ALL='en_US.UTF-8'
```

Consult the documentation for your operating system for the appropriate syntax and location to put these configuration items; `/etc/apache2/envvars` is a common location on Unix platforms. Once you have added these statements to your environment, restart Apache.

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 = '/tmp/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 webpages (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.

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](#).

SCALING

Now that you know how to get Django running on a single server, let's look at how you can scale out a Django installation. This section walks through how a site might scale from a single server to a large-scale cluster that could serve millions of hits an hour.

It's important to note, however, that nearly every large site is large in different ways, so scaling is anything but a one-size-fits-all operation. The following coverage should suffice to show the general principle, and whenever possible we'll try to point out where different choices could be made.

First off, we'll make a pretty big assumption and exclusively talk about scaling under Apache and `mod_python`. Though we know of a number of successful medium- to large-scale FastCGI deployments, we're much more familiar with Apache.

RUNNING ON A SINGLE SERVER

Most sites start out running on a single server, with an architecture that looks something like Figure 13-1.

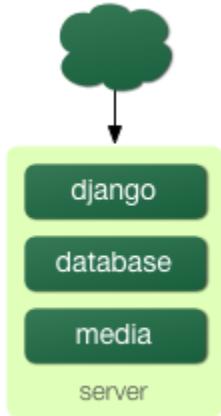


Figure 13-1: a single server Django setup.

However, as traffic increases you'll quickly run into *resource contention* between the different pieces of software. Database servers and Web servers *love* to have the entire server to themselves, so when run on the same server they often end up fighting over the same resources (RAM, CPU) that they'd prefer to monopolize.

This is solved easily by moving the database server to a second machine, as explained in the following section.

SEPARATING OUT THE DATABASE SERVER

As far as Django is concerned, the process of separating out the database server is extremely easy: you'll simply need to change the `DATABASE_HOST` setting to the IP or DNS name of your database server. It's probably a good idea to use the IP if at all possible, as relying on DNS for the connection between your Web server and database server isn't recommended.

With a separate database server, our architecture now looks like Figure 13-2.



Figure 13-2: Moving the database onto a dedicated server.

Here we're starting to move into what's usually called *n-tier* architecture. Don't be scared by the buzzword – it just refers to the fact that different tiers of the Web stack get separated out onto different physical machines.

At this point, if you anticipate ever needing to grow beyond a single database server, it's probably a good idea to start thinking about connection pooling and/or database replication. Unfortunately, there's not nearly enough space to do those topics justice in this book, so you'll need to consult your database's documentation and/or community for more information.

RUNNING A SEPARATE MEDIA SERVER

We still have a big problem left over from the single-server setup: the serving of media from the same box that handles dynamic content.

Those two activities perform best under different circumstances, and by smashing them together on the same box you end up with neither performing particularly well. So the next step is to separate out the media – that is, anything *not* generated by a Django view – onto a dedicated server (see Figure 13-3).

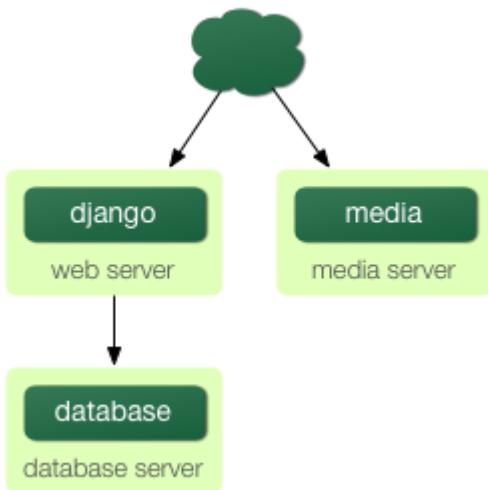


Figure 13-3: Separating out the media server.

Ideally, this media server should run a stripped-down Web server optimized for static media delivery. [Nginx](#) is the preferred option here, although lighttpd is another option, or a heavily stripped down Apache could work too.

For sites heavy in static content (photos, videos, etc.), moving to a separate media server is doubly important and should likely be the *first* step in scaling up.

This step can be slightly tricky, however. If your application involves file uploads, Django needs to be able to write uploaded media to the media server. If media lives on another server, you'll need to arrange a way for that write to happen across the network.

IMPLEMENTING LOAD BALANCING AND REDUNDANCY

At this point, we've broken things down as much as possible. This three-server setup should handle a very large amount of traffic – we served around 10 million hits a day from an architecture of this sort – so if you grow further, you'll need to start adding redundancy.

This is a good thing, actually. One glance at Figure 13-3 shows you that if even a single one of your three servers fails, you'll bring down your entire site. So as you add redundant servers, not only do you increase capacity, but you also increase reliability.

For the sake of this example, let's assume that the Web server hits capacity first. It's relatively easy to get multiple copies of a Django site running on different hardware – just copy all the code onto multiple machines, and start Apache on both of them.

However, you'll need another piece of software to distribute traffic over your multiple servers: a *load balancer*. You can buy expensive and proprietary hardware load balancers, but there are a few high-quality open source software load balancers out there.

Apache's `mod_proxy` is one option, but we've found Perlbal (<http://www.djangoproject.com/r/perlbal/>) to be fantastic. It's a load balancer and reverse proxy written by the same folks who wrote `memcached` (see Chapter 17).

With the Web servers now clustered, our evolving architecture starts to look more complex, as shown in Figure 13-4.

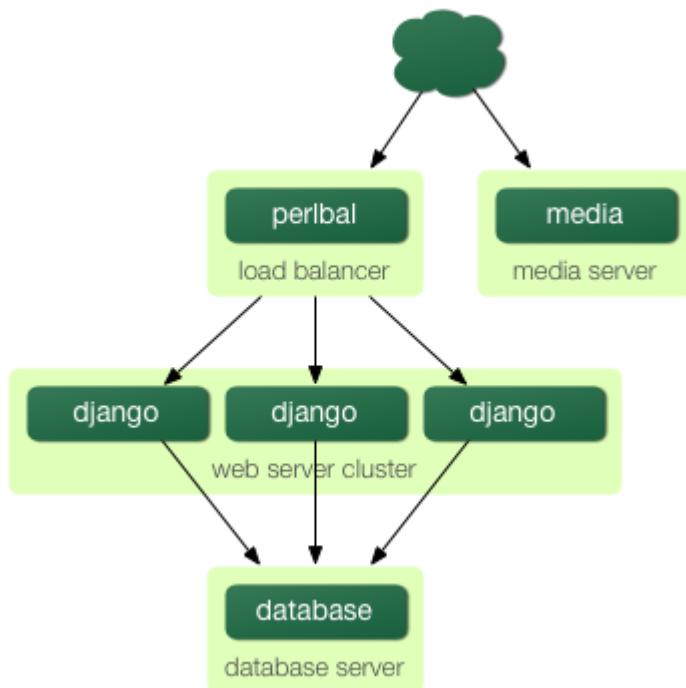


Figure 13-4: A load-balanced, redundant server setup.

Notice that in the diagram the Web servers are referred to as a cluster to indicate that the number of servers is basically variable. Once you have a load balancer out front, you can easily add and remove back-end Web servers without a second of downtime.

GOING BIG

At this point, the next few steps are pretty much derivatives of the last one:

- As you need more database performance, you might want to add replicated database servers. MySQL includes built-in replication; PostgreSQL users should look into Slony (<http://www.djangoproject.com/r/slony/>) and pgpool (<http://www.djangoproject.com/r/pgpool/>) for replication and connection pooling, respectively.
- If the single load balancer isn't enough, you can add more load balancer machines out front and distribute among them using round-robin DNS.
- If a single media server doesn't suffice, you can add more media servers and distribute the load with your load-balancing cluster.
- If you need more cache storage, you can add dedicated cache servers.
- At any stage, if a cluster isn't performing well, you can add more servers to the cluster.

After a few of these iterations, a large-scale architecture might look like Figure 13-5.

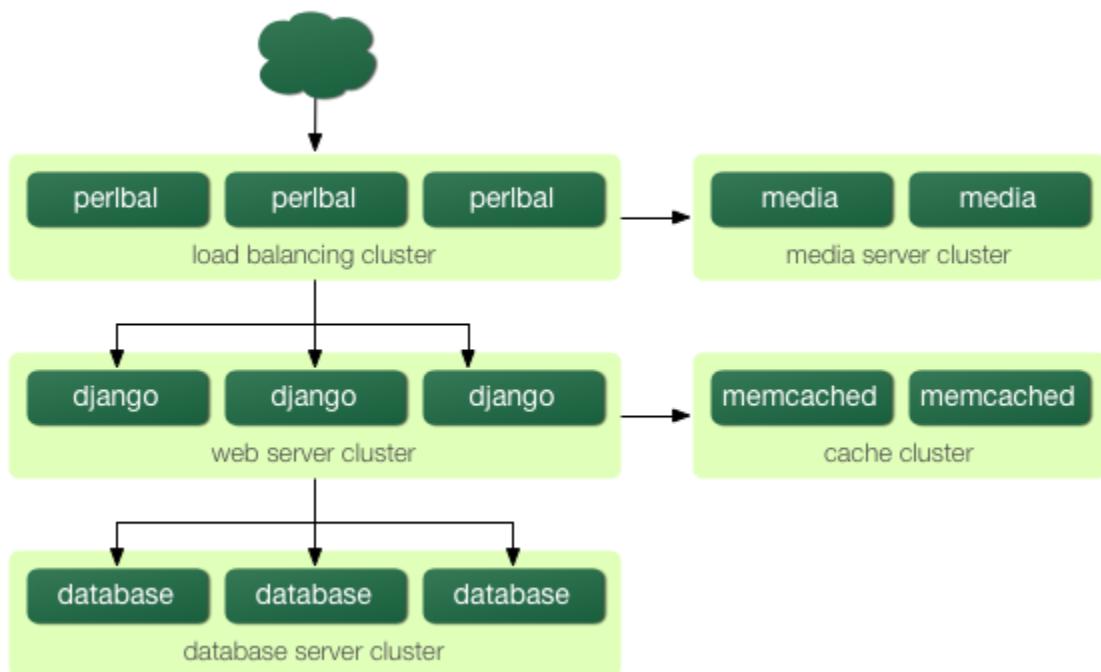


Figure 13-5. An example large-scale Django setup.

Though we've shown only two or three servers at each level, there's no fundamental limit to how many you can add.

PERFORMANCE TUNING

If you have huge amount of money, you can just keep throwing hardware at scaling problems. For the rest of us, though, performance tuning is a must.

Note

Incidentally, if anyone with monstrous gobs of cash is actually reading this book, please consider a substantial donation to the Django Foundation. We accept uncut diamonds and gold ingots, too.

Unfortunately, performance tuning is much more of an art than a science, and it is even more difficult to write about than scaling. If you're serious about deploying a large-scale Django application, you should spend a great deal of time learning how to tune each piece of your stack.

The following sections, though, present a few Django-specific tuning tips we've discovered over the years.

THERE'S NO SUCH THING As Too Much RAM

Even the really expensive RAM is relatively affordable these days. Buy as much RAM as you can possibly afford, and then buy a little bit more.

Faster processors won't improve performance all that much; most Web servers spend up to 90% of their time waiting on disk I/O. As soon as you start swapping, performance will just die. Faster disks might help slightly, but they're much more expensive than RAM, such that it doesn't really matter.

If you have multiple servers, the first place to put your RAM is in the database server. If you can afford it, get enough RAM to get fit your entire database into memory. This shouldn't be too hard; we've developed a site with more than half a million newspaper articles, and it took under 2GB of space.

Next, max out the RAM on your Web server. The ideal situation is one where neither server swaps – ever. If you get to that point, you should be able to withstand most normal traffic.

TURN OFF KEEP-ALIVE

Keep-Alive is a feature of HTTP that allows multiple HTTP requests to be served over a single TCP connection, avoiding the TCP setup/teardown overhead.

This looks good at first glance, but it can kill the performance of a Django site. If you're properly serving media from a separate server, each user browsing your site will only request a page from your Django server every ten seconds or so. This leaves HTTP servers waiting around for the next keep-alive request, and an idle HTTP server just consumes RAM that an active one should be using.

USE MEMCACHED

Although Django supports a number of different cache back-ends, none of them even come *close* to being as fast as memcached. If you have a high-traffic site, don't even bother with the other backends – go straight to memcached.

USE MEMCACHED OFTEN

Of course, selecting memcached does you no good if you don't actually use it. Chapter 17 is your best friend here: learn how to use Django's cache framework, and use it everywhere possible. Aggressive, preemptive caching is usually the only thing that will keep a site up under major traffic.

JOIN THE CONVERSATION

Each piece of the Django stack – from Linux to Apache to PostgreSQL or MySQL – has an awesome community behind it. If you really want to get that last 1% out of your servers, join the open source communities behind your software and ask for help. Most free-software community members will be happy to help.

And also be sure to join the Django community. Your humble authors are only two members of an incredibly active, growing group of Django developers. Our community has a huge amount of collective experience to offer.

LICENSE & COPYRIGHT

Mastering Django: Core is a Modified Version of “The Django Book” by Adrian Holovaty and Jacob Kaplan-Moss. New content for this Modified Version copyright Nigel George, the Django Software Foundation and individual contributors.

Original content Copyright Adrian Holovaty, Jacob Kaplan-Moss, and contributors.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section below entitled “GNU Free Documentation License”.

GNU FREE DOCUMENTATION LICENSE

GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright(c) 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
<http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a

printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise

Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The “publisher” means any person or entity that distributes copies of the Document to the public.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or

distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing

distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

4. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

8. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled “Acknowledgements” or “Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers. If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version,

but may differ in detail to address new problems or concerns. See
<http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

“Massive Multiauthor Collaboration Site” (or “MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A “Massive Multiauthor Collaboration” (or “MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.

“CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

“Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is “eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License". If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.
