

Mastering Django: Core

The Complete Guide to Django 1.8 LTS

Delivers absolutely everything you will ever need to know to become a master Django programmer

Nigel George

Table of Contents

[Mastering Django: Core](#)

[Credits](#)

[About the Author](#)

[www.PacktPub.com](#)

[Why subscribe?](#)

[Preface](#)

[What you need for this book](#)

[Who this book is for](#)

[Conventions](#)

[Reader feedback](#)

[Customer support](#)

[Errata](#)

[Piracy](#)

[Questions](#)

[1. Introduction to Django and Getting Started](#)

[Introducing Django](#)

[Django's history](#)

[Installing Django](#)

[Installing Python](#)

[Python versions](#)

[Installation](#)

[Installing a Python Virtual Environment](#)

[Installing Django](#)

[Setting up a database](#)

[Starting a project](#)

[Django settings](#)

[The development server](#)

[The Model-View-Controller \(MVC\) design pattern](#)

[What's next?](#)

[2. Views and URLconfs](#)

[Your first Django-powered page: Hello World](#)

[Your first view](#)

[Your first URLconf](#)

[Regular expressions](#)

[A quick note about 404 errors](#)

[A quick note about the site root](#)

[How Django processes a request](#)

[Your second view: dynamic content](#)

[URLconfs and loose coupling](#)

[Your third view: dynamic URLs](#)

[Django's pretty error pages](#)

[What's next?](#)

[3. Templates](#)

Template system basics
Using the template system
 Creating template objects
 Rendering a template
Dictionaries and contexts
 Multiple contexts, same template
 Context variable lookup
 Method call behavior
 How invalid variables are handled
Basic templatetags and filters
 Tags
 if/else
 for
 ifequal/ifnotequal
 Comments
 Filters
Philosophies and limitations
Using templates in views
Template loading
 Template directories
render()
Template subdirectories
The include template tag
Template inheritance
What's next?

4. Models
 The "dumb" way to do database queries in views
 Configuring the database
 Your first app
 Defining Models in Python
 Your first model
 Installing the Model
 Basic data access
 Adding model string representations
 Inserting and updating data
 Selecting objects
 Filtering data
 Retrieving single objects
 Ordering data
 Chaining lookups
 Slicing data
 Updating multiple objects in one statement
 Deleting objects
 What's next?
5. The Django Admin Site
 Using the admin site
 Start the development server
 Enter the admin site
 Adding your models to the admin site

Making fields optional

Making date and numeric fields optional

Customizing field labels

Custom model admin classes

Customizing change lists

Customizing edit forms

Users, groups, and permissions

When and why to use the admin interface-and when not to

What's next?

6. Forms

Getting data from the Request Object

Information about the URL

Other information about the Request

Information about submitted data

A simple form-handling example

Query string parameters

Improving our simple form-handling example

Simple validation

Making a contact form

Your first form class

Tying form objects into views

Changing how fields are rendered

Setting a maximum length

Setting initial values

Custom validation rules

Specifying labels

Customizing form design

What's next?

7. Advanced Views and URLconfs

URLconf Tips and Tricks

Streamlining function imports

Special-Casing URLs in debug mode

Named groupsPreview

The matching/grouping algorithm

What the URLconf searches against

Captured arguments are always strings

Specifying defaults for view arguments

Performance

Error handling

Including other URLconfs

Captured parameters

Passing extra options to view functions

Passing extra options to include()

Reverse resolution of URLs

Examples

Naming URL patterns

URL namespaces

Reversing namespaced URLs

URL namespaces and included URLconfs

What's next?

8. Advanced Templates

[Template language review](#)

[Requestcontext and context processors](#)

[auth](#)

[DEBUG](#)

[i18n](#)

[MEDIA](#)

[static](#)

[csrf](#)

[Request](#)

[messages](#)

[Guidelines for writing our own context processors](#)

[Automatic HTML escaping](#)

[How to turn it off](#)

[For individual variables](#)

[For template blocks](#)

[Automatic escaping of string literals in filter arguments](#)

[Inside Template loading](#)

[The DIRS option](#)

[Loader types](#)

[Filesystem loader](#)

[App directories loader](#)

[Other loaders](#)

[Extending the template system](#)

[Code layout](#)

[Creating a template library](#)

[Custom template tags and filters](#)

[Writing custom template filters](#)

[Registering custom filters](#)

[Template filters that expect strings](#)

[Filters and autoescaping](#)

[Filters and time zones](#)

[Writing custom template tags](#)

[Simple tags](#)

[Inclusion tags](#)

[Assignment tags](#)

[Advanced custom template tags](#)

[A quick overview](#)

[Writing the compilation function](#)

[Writing the renderer](#)

[Autoescaping Considerations](#)

[Thread-safety Considerations](#)

[Registering the tag](#)

[Passing template variables to The Tag](#)

[Setting a variable in the context](#)

[Variable scope in context](#)

[Parsing until another block tag](#)

[Parsing until another block tag, and saving contents](#)

[What's next](#)

[9. Advanced Models](#)

[Related objects](#)

[Accessing ForeignKey values](#)

[Accessing many-to-many values](#)

[Managers](#)

[Adding extra manager methods](#)

[Modifying initial manager QuerySets](#)

[Model methods](#)

[Overriding predefined model methods](#)

[Executing raw SQL queries](#)

[Performing raw queries](#)

[Model table names](#)

[Mapping query fields to model fields](#)

[Index lookups](#)

[Deferring model fields](#)

[Adding annotations](#)

[Passing parameters into raw\(\)](#)

[Executing custom SQL directly](#)

[Connections and cursors](#)

[Adding extra Manager methods](#)

[What's next?](#)

[10. Generic Views](#)

[Generic views of objects](#)

[Making "friendly" template contexts](#)

[Adding extra context](#)

[Viewing subsets of objects](#)

[Dynamic filtering](#)

[Performing extra work](#)

[What's next?](#)

[11. User Authentication in Django](#)

[Overview](#)

[Using the Django authentication system](#)

[User objects](#)

[Creating superusers](#)

[Creating users](#)

[Changing passwords](#)

[Permissions and authorization](#)

[Default permissions](#)

[Groups](#)

[Programmatically creating permissions](#)

[Permission caching](#)

[Authentication in web requests](#)

[How to log a user in](#)

[How to log a user out](#)

[Limiting access to logged-in users](#)

[The raw way](#)

[The login_required decorator](#)

[Limiting access to logged-in users that pass a test](#)

	The permission_required() decorator
	Session invalidation on password change
Authentication views	
	Login
	Logout
	Logout_then_login
	Password_change
	Password_change_done
	Password_reset
	Password_reset_done
	Password_reset_confirm
	Password_reset_complete
	The redirect_to_login helper function
	Built-in forms
Authenticating data in templates	
	Users
	Permissions
Managing users in the admin	
	Creating users
	Changing passwords
Password management in Django	
	How Django stores passwords
	Using Bcrypt with Django
	Password truncation with BCryptPasswordHasher
	Other Bcrypt implementations
	Increasing the work factor
	Password upgrading
	Manually managing a user's password
Customizing authentication in Django	
	Other authentication sources
	Specifying authentication backends
	Writing an authentication backend
	Handling authorization in custom backends
	Authorization for anonymous users
	Authorization for inactive users
	Handling object permissions
Custom permissions	
Extending the existing user model	
Substituting a custom user model	
What's next?	
12. Testing in Django	
	Introduction to testing
	Introducing automated testing
	What are automated tests?
	So why create tests?
	Basic testing strategies
	Writing a test
	Creating a test
	Running tests

Testing tools

The test client

Provided TestCase classes

Simple TestCase

Transaction TestCase

TestCase

LiveServerTestCase

Test cases features

Default test client

Fixture loading

Overriding settings

 settings()

 modify_settings()

 override_settings()

 modify_settings()

Assertions

Email services

Management commands

Skipping tests

The test database

Using different testing frameworks

What's next?

13. Deploying Django

Preparing your codebase for production

Deployment checklist

Critical settings

 SECRET_KEY

 DEBUG

Environment-specific settings

 ALLOWED_HOSTS

 CACHES

 DATABASES

 EMAIL_BACKEND and Related Settings

 STATIC_ROOT and STATIC_URL

 MEDIA_ROOT and MEDIA_URL

HTTPS

 CSRF_COOKIE_SECURE

 SESSION_COOKIE_SECURE

Performance optimizations

 CONN_MAX_AGE

 TEMPLATES

Error reporting

 LOGGING

 ADMINS and MANAGERS

 Customize the default error views

Using a virtualenv

Using different settings for production

Deploying Django to a production server

Deploying Django with Apache and mod_wsgi

[Basic configuration](#)
[Using mod_wsgi daemon Mode](#)
[Serving files](#)
[Serving the admin files](#)
[If you get a UnicodeEncodeError](#)
[Serving static files in production](#)
[Serving the site and your static files from the same server](#)
[Serving static files from a dedicated server](#)
[Serving static files from a cloud service or CDN](#)

[Scaling](#)
[Running on a single server](#)
[Separating out the database server](#)
[Running a separate media server](#)
[Implementing load balancing and redundancy](#)
[Going big](#)

[Performance tuning](#)
[There's no such thing as Too Much RAM](#)
[Turn off Keep-Alive](#)
[Use Memcached](#)
[Use Memcached often](#)
[Join the conversation](#)

[What's next?](#)

14. Generating Non-HTML Content

[The basics: views and MIME types](#)

[Producing CSV](#)
[Streaming large CSV files](#)

[Using the template system](#)

[Other text-based formats](#)

[Generating PDF](#)
[Install ReportLab](#)
[Write your view](#)
[Complex PDF's](#)
[Further resources](#)
[Other possibilities](#)

[The syndication feed framework](#)

[The high-level framework](#)
[Overview](#)
[Feed classes](#)
[A simple example](#)
[A complex example](#)
[Specifying the type of feed](#)
[Enclosures](#)
[Language](#)
[URLs](#)
[Publishing Atom and RSS Feeds in tandem](#)

[The low-level framework](#)
[SyndicationFeed classes](#)
[SyndicationFeed.__init__\(\)](#)
[SyndicationFeed.add_item\(\)](#)

SyndicationFeed.write()
SyndicationFeed.writeString()
Custom feed generators
SyndicationFeed.root_attributes(self,)
SyndicationFeed.add_root_elements(self, handler)
SyndicationFeed.item_attributes(self, item)
SyndicationFeed.add_item_elements(self, handler, item)

The Sitemap framework

Installation
Initialization
Sitemap classes
A simple example
Sitemap class reference
items
location
lastmod
changefreq
priority
protocol
i18n
Shortcuts
Example
Sitemap for static views
Creating a sitemap index
Template customization
Context variables
Index
Sitemap
Pinging google
django.contrib.syndication.ping_google()
Pinging Google via manage.py

What's next?

15. Django Sessions

Enabling sessions

Configuring the session engine

Using database-backed sessions
Using cached sessions
Using file-based sessions
Using cookie-based sessions

Using Sessions in Views

flush()
set_test_cookie()
test_cookie_worked()
delete_test_cookie()
set_expiry(value)
get_expiry_age()
get_expiry_date()
get_expire_at_browser_close()
clear_expired()

[cycle_key\(\)](#)
[Session object guidelines](#)
[Session serialization](#)
 [Bundled serializers](#)
 [serializers.JSONSerializer](#)
 [serializers.PickleSerializer](#)
 [Write your own serializer](#)
[Setting test cookies](#)
[Using sessions out of views](#)
[When sessions are saved](#)
[Browser-length sessions vs. persistent sessions](#)
[Clearing the session store](#)
[What's next](#)

[16. Djangos Cache Framework](#)

[Setting up the cache](#)
 [Memcached](#)
 [Database caching](#)
 [Creating the cache table](#)
 [Multiple databases](#)
 [Filesystem caching](#)
 [Local-memory caching](#)
 [Dummy caching \(for development\)](#)
 [Using a custom cache backend](#)
 [Cache arguments](#)
[The per-site cache](#)
[The per-view cache](#)
 [Specifying per-view Cache in the URLconf](#)
[Template fragment caching](#)
[The low-level cache API](#)
 [Accessing the cache](#)
 [Basic usage](#)
 [Cache key prefixing](#)
 [Cache versioning](#)
 [Cache key transformation](#)
 [Cache key warnings](#)
[Downstream caches](#)
[Using vary headers](#)
[Controlling cache: using other headers](#)
[What's next?](#)

[17. Django Middleware](#)

[Activating middleware](#)
[Hooks and application order](#)
[Writing your own middleware](#)
 [process_request](#)
 [process_view](#)
 [process_template_response](#)
 [process_response](#)
 [Dealing with streaming responses](#)
 [process_exception](#)

init
 Marking middleware as unused
 Additional guidelines
Available middleware
 Cache middleware
 Common middleware
 GZip middleware
 Conditional GET middleware
 Locale middleware
 Message middleware
 Security middleware
 HTTP strict transport security
 X-content-type-options: nosniff
 X-XSS-protection
 SSL redirect
 Session middleware
 Site middleware
 Authentication middleware
 CSRF protection middleware
 X-Frame-options middleware

Middleware ordering

What's next?

18. Internationalization

Definitions

 Internationalization
 Localization
 locale name
 language code
 message file
 translation string
 format file

Translation

 Internationalization: in Python code

 Standard translation

 Comments for Translators

 Marking strings as No-Op

 Pluralization

 Contextual markers

 Lazy translation

 Model fields and relationships

 Model verbose names values

 Model methods short_description attribute values

 Working with lazy translation objects

 Lazy translations and plural

 Joining strings: string_concat()

 Other uses of lazy in delayed translations

 Localized names of languages

 Internationalization: In template code

 trans template tag

blocktrans template tag
String literals passed to tags and filters
Comments for translators in templates
Switching language in templates
Other tags
Internationalization: In Javascript code
The javascript_catalog view
Using the JavaScript translation catalog
Note on performance
Internationalization: In URL patterns
Language prefix in URL patterns
Translating URL patterns
Reversing in templates
Localization: How to create language files
Message files
Compiling message files
Creating message files from JavaScript source code
gettext on windows
Customizing the makemessages command
Explicitly setting the active language
Using translations outside views and templates
Implementation notes
Specialties of Django translation
How Django discovers language preference
How Django discovers translations
What's next?
19. Security in Django
Django's built in security features
Cross Site Scripting (XSS) protection
Cross Site Request Forgery (CSRF) protection
How to use it
AJAX
Other template engines
The decorator method
Rejected requests
How it works
Caching
Testing
Limitations
Edge cases
Utilities
`django.views.decorators.csrf.csrf_exempt(view)`
`django.views.decorators.csrf.requires_csrf_token(view)`
`django.views.decorators.csrf.ensure_csrf_cookie(view)`
Contrib and reusable apps
CSRF settings
SQL injection protection
Clickjacking protection
An example of clickjacking

Preventing clickjacking
How to use it
 Setting X-Frame-Options for all responses
 Setting X-Frame-Options per view
Limitations
Browsers that support X-Frame-Options
SSL/HTTPS
 HTTP strict transport security
Host header validation
Session security
 User-Uploaded content
Additional security tips
 Archive of security issues
 Cryptographic signing
 Protecting the SECRET_KEY
 Using the low-level API
 Using the salt argument
 Verifying timestamped values
 Protecting complex data structures
 Security middleware
What's next?
20. More on Installing Django
 Running other databases
 Installing Django manually
 Upgrading Django
 Remove any old versions of Django
 Installing a Distribution-specific package
 Installing the development version
What's next?
21. Advanced Database Management
 General notes
 Persistent connections
 Connection management
 Caveats
 Encoding
 postgreSQL notes
 Optimizing PostgreSQL's configuration
 Isolation level
 Indexes for varchar and text columns
 MySQL notes
 Version support
 Storage engines
 MySQL DB API drivers
 mySQLdb
 mySQLclient
 mySQL connector/python
 Timezone definitions
 Creating your database
 Collation settings

[Connecting to the database](#)
[Creating your tables](#)
[Table names](#)
[Savepoints](#)
[Notes on specific fields](#)
[Character fields](#)
[Fractional seconds support for time and datetime fields](#)
[TIMESTAMP columns](#)
[Row locking with Queryset.Select_Force_Update\(\)](#)
[Automatic typecasting can cause unexpected results](#)

[SQLite notes](#)

[Substring matching and case sensitivity](#)
[Old SQLite and CASE expressions](#)
[Using newer versions of the SQLite DB-API 2.0 driver](#)
[Database is locked errors](#)
[queryset.Select_Force_Update\(\) not Supported](#)
[pyformat parameter style in raw queries not supported](#)
[Parameters not quoted in connection.queries](#)

[Oracle notes](#)

[Connecting to the database](#)
[Threaded option](#)
[INSERT ... RETURNING INTO](#)
[Naming issues](#)
[NULL and empty strings](#)
[Textfield limitations](#)
[Using a 3rd-Party database backend](#)
[Integrating Django with a legacy database](#)
[Give Django your database parameters](#)
[Auto-generate the models](#)
[Install the core Django tables](#)
[Cleaning up generated models](#)
[Test and tweak](#)

[What's next?](#)

[A. Model Definition Reference](#)

[Fields](#)

[Field name restrictions](#)
[FileField notes](#)
[FileField FileField.upload_to](#)
[FileField.storage](#)
[FileField and FieldFile](#)
[FieldFile.url](#)
[FieldFile.open\(mode='rb'\)](#)
[FieldFile.close\(\)](#)
[FieldFile.save\(name, content, save=True\)](#)
[FieldFile.delete\(save=True\)](#)

[Universal field options](#)

[Field attribute reference](#)

[Attributes for fields](#)
[Field.auto_created](#)

```
Field.concrete
Field.hidden
Field.is_relation
Field.model
Attributes for fields with relations
Field.many_to_many
Field.many_to_one
Field.one_to_many
Field.one_to_one
Field.related_model
Relationships
ForeignKey
Database representation
Arguments
    limit_choices_to
    related_name
    related_query_name
    to_field
    db_constraint
    on_delete
    swappable
ManyToManyField
Database representation
Arguments
    related_name
    related_query_name
    limit_choices_to
    symmetrical
    through
    through_fields
    db_table
    db_constraint
    swappable
OneToOneField
    parent_link
Model metadata options
B. Database API Reference
Creating objects
Saving changes to objects
    Saving ForeignKey and ManyToManyField fields
Retrieving objects
    Retrieving all objects
    Retrieving specific objects with filters
        Chaining filters
    Filtered querysets are unique
        QuerySets are lazy
    Retrieving a single object with get
    Other queryset methods
    Limiting querysets
```

- Field lookups
- Lookups that span relationships
 - Spanning multi-valued relationships
- Filters can reference fields on the model
- The pk lookup shortcut
- Escaping percent signs and underscores in LIKE statements
- Caching and querysets
 - When querysets are not cached
- Complex lookups with Q objects
- Comparing objects
- Deleting objects
- Copying model instances
- Updating multiple objects at once
- Related objects
 - One-to-many relationships
 - Forward
 - Following relationships backward
 - Using a custom reverse manager
 - Additional methods to handle related objects
 - Many-to-many relationships
 - One-to-one relationships
 - Queries over related objects
- Falling back to raw SQL

C. Generic View Reference

- Common arguments to generic views
- Simple generic views
 - Rendering a template-TempalteView
 - Redirecting to another URL
 - Attributes
 - url
 - pattern_name
 - permanent
 - query_string
 - Methods
 - List/detail generic views
 - Lists of objects
 - Detail views
 - Date-Based Generic Views
 - ArchiveIndexView
 - YearArchiveView
 - MonthArchiveView
 - WeekArchiveView
 - DayArchiveView
 - TodayArchiveView
 - DateDetailView
 - Form handling with class-based views
 - Basic forms
 - Model forms
 - Models and request.user

AJAX example

D. Settings

[What's a settings file?](#)

[Default settings](#)

[Seeing which settings you've changed](#)

[Using settings in Python code](#)

[Altering settings at runtime](#)

[Security](#)

[Creating your own settings](#)

[DJANGO_SETTINGS_MODULE](#)

[The django-admin utility](#)

[On the server \(mod_wsgi\)](#)

[Using settings without setting DJANGO_SETTINGS_MODULE](#)

[Custom default settings](#)

[Either configure\(\) or DJANGO_SETTINGS_MODULE is required](#)

[Available settings](#)

[Core settings](#)

[Auth](#)

[Messages](#)

[Sessions](#)

[Sites](#)

[Static files](#)

E. Built-in Template Tags and Filters

[Built-in tags](#)

[autoescape](#)

[block](#)

[comment](#)

[csrf_token](#)

[cycle](#)

[debug](#)

[extends](#)

[filter](#)

[firstof](#)

[for](#)

[for... empty](#)

[if](#)

[Boolean operators](#)

[Complex expressions](#)

[Filters](#)

[ifchanged](#)

[ifequal](#)

[ifnotequal](#)

[include](#)

[load](#)

[lorem](#)

[now](#)

[regroup](#)

[spaceless](#)

[templatetag](#)

url
verbatim
widthratio
with

Built-in filters

add
addslashes
capfirst
center
cut
date
default
default_if_none
dictsort
dictsortreversed
divisibleby
escape
escapejs
filesizeformat
first
floatformat
get_digit
iriencode
join
last
length
length_is
linebreaks
linebreaksbr
linenumbers
ljust
lower
make_list
phone2numeric
pluralize
pprint
random
rjust
safe
safeseq
slice
slugify
stringformat
striptags
time
timesince
timeuntil
title
truncatechars

[truncatechars_html](#)
[truncatewords](#)
[truncatewords_html](#)
[unordered_list](#)
[upper](#)
[urlencode](#)
[urlize](#)
[urlizetrunc](#)
[wordcount](#)
[wordwrap](#)
[yesno](#)

[Internationalization tags and filters](#)

[i18n](#)
[l10n](#)
[tz](#)

[Other tags and filters libraries](#)

[static](#)
[get_static_prefix](#)
[get_media_prefix](#)

F. Request and Response Objects

[HttpRequest objects](#)

[Attributes](#)
[Methods](#)

[QueryDict objects](#)

[Methods](#)

[HttpResponse objects](#)

[Usage](#)
[Attributes](#)
[Methods](#)
[HttpResponse subclasses](#)

[JsonResponse Objects](#)

[Usage](#)

[StreamingHttpResponse objects](#)

[Performance considerations](#)
[Attributes](#)

[FileResponse objects](#)

[Error views](#)

[The 404 \(page not found\) view](#)
[The 500 \(server error\) view](#)
[The 403 \(HTTP Forbidden\) view](#)
[The 400 \(bad request\) view](#)

[Customizing error views](#)

G. Developing Django with Visual Studio

[Installing Visual Studio](#)

[Install PTVS and Web Essentials](#)

[Creating A Django project](#)

[Start a Django project](#)

[Django development in Visual Studio](#)

[Integration of Django management commands](#)

Easy installation of Python packages
Easy installation of new Django apps

Mastering Django: Core

Mastering Django: Core

Copyright © 2016 Nigel George All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: December 2016

Production reference: 1291116

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street Birmingham

B3 2PB, UK.

ISBN 978-1-78728-114-1

www.packtpub.com

Credits

Author Nigel George	Indexer Tejal Daruwale Soni
Acquisition Editor Reshma Raman	Production Coordinator Aparna Bhagat
Technical Editor Rashil Shah	

About the Author

Nigel George is a business systems developer who specializes 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 back end 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 and has written applications in C, C#, C++, VB, VBA, HTML, JavaScript, Python and PHP.

He has another book on Django—*Beginning Django CMS*—published by Apress in December 2015.

Nigel lives in Newcastle, NSW, Australia.

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 many years ago when I first discovered the “new kid on the webframework block”. It’s your support that makes Django so great. Thank you.

www.PacktPub.com

For support files and downloads related to your book,
please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www.packtpub.com/mapt>

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Preface

What you need for this book

Required programming knowledge

Readers of this book should understand the basics of procedural and object-oriented programming: control structures (such as if, while, or for), data structures (lists, hashes/dictionaries), variables, classes, and objects. Experience in web development is, as you may expect, very helpful, but it is not required to understand this book. Throughout the book, I 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* (that is, 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, I recommend you read the official Python tutorial (for more information visit <http://docs.python.org/tut/>). I also recommend Mark Pilgrim's free book *Dive Into Python*, available online 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 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 1.8 LTS. At the time of printing (July 2016), the most current production version of Django 1.8 LTS is 1.8.13.

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 at
<https://docs.djangoproject.com/en/dev/releases/>.

For any queries visit: <http://masteringdjango.com>.

Who this book is for

This book assumes you have a basic understanding of the Internet and programming. Experience with Python or Django would be an advantage, but is not necessary. It is ideal for beginner to intermediate programmers looking for a fast, secure, scalable, and maintainable alternative web development platform to those based on PHP, Java, and dotNET.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Type `python` at a command prompt (or in [Applications/Utilities/Terminal](#), in OS X)."

A block of code is set as follows:

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

Any command-line input or output is written as follows:

```
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.
>>>
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "You

should see the text **Hello world**-the output of your Django view (Figure 2-1)."

NOTE

Warnings or important notes appear in a box like this.

TIP

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Chapter 1. Introduction to Django and Getting Started

Introducing 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).

By Adrian Holovaty and Jacob Kaplan-Moss-December 2009

In the early days, web developers wrote every page by hand. Updating a website meant editing HTML; a *redesign* involved redoing every single page, one at a time. As websites 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 **National Center for Supercomputing Applications** (the NCSA 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 (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 websites. 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 is now the most popular tool used to create dynamic websites, and dozens of similar languages (ASP, JSP, and so on.) 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; it is very easy 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 day's or hour's 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 5](#), *The Django Admin Site*) that are particularly well suited for "content" sites such as [Amazon.com](#), [craigslist.org](#), and [washingtonpost.com](#) that offer dynamic and database-driven information.

Don't let that turn you off, though 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 website. (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.

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

Before you can start learning how to use Django, you must first install some software on your computer. Fortunately, this is a simple three step process:

1. Install Python.
2. Install a Python Virtual Environment.
3. Install Django.

If this does not sound familiar to you don't worry, in this chapter, lets assume that you have never installed software from the command line before and will lead you through it step by step.

I have written this section for those of you running Windows. While there is a strong *nix and OSX user base for Django, most new users are on Windows. If you are using Mac or Linux, there are a large number of

resources on the Internet; with the best place to start being Django's own installation instructions. For more information visit

[https://docs.djangoproject.com/en/1.8/topics/install/.](https://docs.djangoproject.com/en/1.8/topics/install/)

For Windows users, your computer can be running any recent version of Windows (Vista, 7, 8.1, or 10). This chapter also assumes you're installing Django on a desktop or laptop computer 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 the 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 20](#), [More on Installing Django](#), and [Chapter 21](#), [Advanced Database Management](#).

NOTE

If you are using Windows, I recommend that you try out Visual Studio for all your Django development. Microsoft has made a significant investment in providing support for Python and Django programmers. This includes full IntelliSense support for Python/Django and incorporation of all of Django's command line tools into the VS IDE.

Best of all it's entirely free. I know, who would have expected that from M\$??, but it's true!

See [Appendix G, Developing Django with Visual Studio](#) for a complete installation guide for Visual Studio Community 2015, as well as a few tips on developing Django in Windows.

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, 3.4 and 3.5. For each version of Python, only the latest micro release (A.B.C) is supported.

If you are just trialling Django, it doesn't really matter whether you use Python 2 or 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 (for more information visit <https://wiki.python.org/moin/Python2orPython3>, 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 (for example, legacy libraries), Python 3 is the way to go.

TIP

NOTE: All of the code samples in this book are written in Python 3

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.
```

NOTE

You can see that, in the preceding 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.

Assuming Python is not installed on your system, we first need to get the installer. Go to <https://www.python.org/downloads/>, and click the big yellow button that says **Download Python 3.x.x**.

At the time of writing, the latest version of Python is 3.5.1, but it may have been updated by the time you read this, so the numbers may be slightly different.

DO NOT download version 2.7.x as this is the old version of Python. All of the code in this book is written in Python 3, so you will get compilation errors if you try to run the code on Python 2.

Once you have downloaded the Python installer, go to your **Downloads** folder and double-click the file `python3.x.x.msi` to run the installer. The installation process is the same as any other Windows program, so if you have installed software before, there should be no problem here, however, there is one extremely important customization you must make.

NOTE

Do not forget this next step as it will solve most problems that arise from an incorrect mapping of [pythonpath](#) (an important variable for Python installations) in Windows.

By default, the Python executable is not added to the Windows PATH statement. For Django to work properly, Python must be listed in the PATH statement.

Fortunately, this is easy to rectify:

- In Python 3.4.x, When the installer opens the customization window, the option **Add python.exe to Path** is not selected, you must change this to **Will be installed on a local hard drive** as shown in *Figure 1.1.*

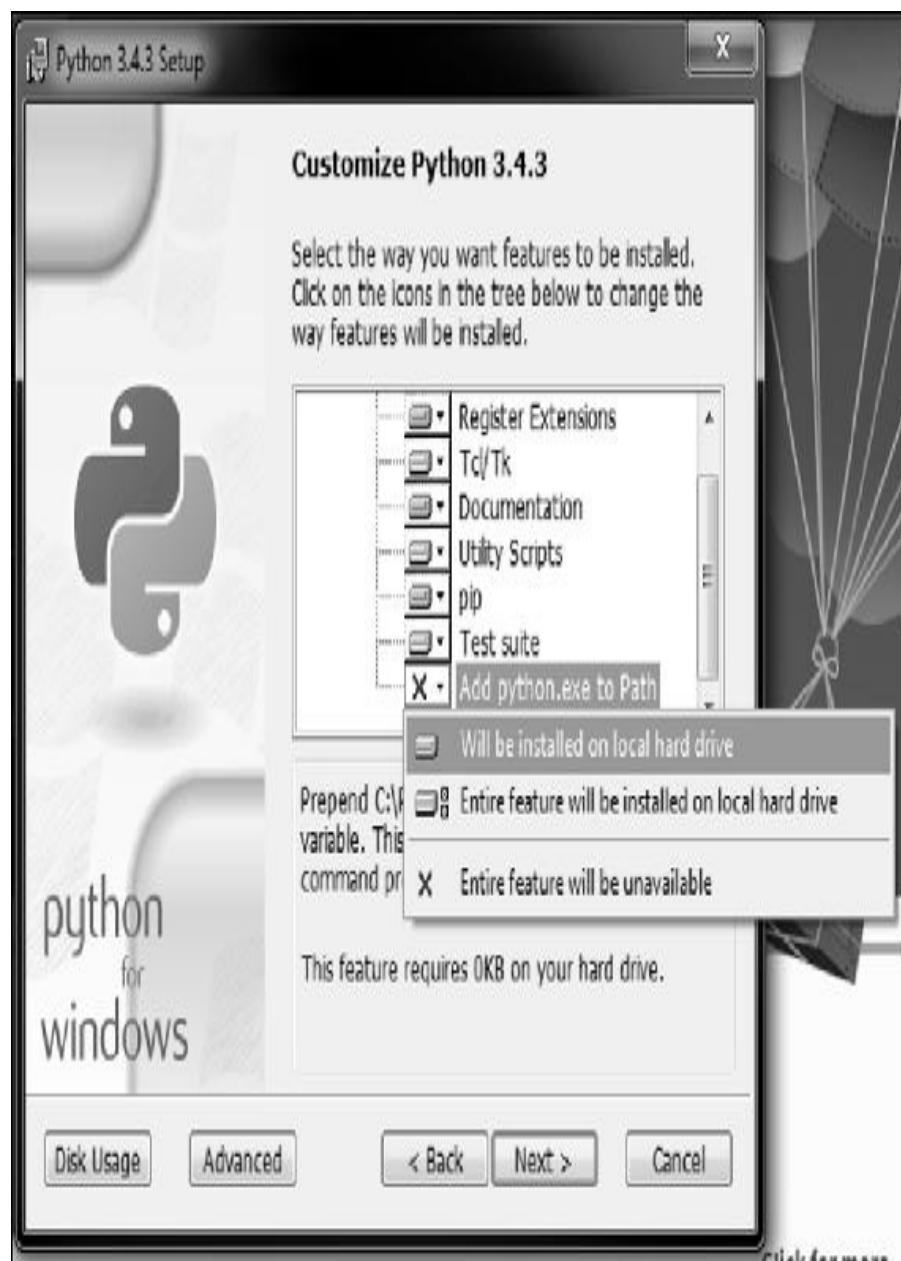


Figure 1.1: Add Python to PATH (Version 3.4.x).

- In Python 3.5.x you make sure **Add Python 3.5 to PATH** is checked before installing (*Figure 1.2*).



Figure 1.2: Add Python to PATH (Version 3.5.x).

Once Python is installed, you should be able to re-open the command window and type python at the command prompt and get something like this:

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6
2015, 01:38:48)
[MSC v.1900 32 bit (Intel)] on win32
```

```
Type "help", "copyright", "credits" or
"license" for more
information.

>>>
```

While you are at it, there is one more important thing to do. Exit out of Python with *CTRL+C*. At the command prompt type, the following and hit enter:

```
python -m pip install -U pip
```

The output will be something similar to this:

```
C:\Users\nigel>python -m pip install -U
pip
Collecting pip
  Downloading pip-8.1.2-py2.py3-none-
any.whl (1.2MB)
    100%
|#####| 1.2MB
198kB/s
Installing collected packages: pip
  Found existing installation: pip 7.1.2
  Uninstalling pip-7.1.2:
  Successfully uninstalled pip-7.1.2
  Successfully installed pip-8.1.2
```

You don't need to understand exactly what this command does right now; put briefly **pip** is the Python package manager. It's used to install Python packages: **pip** is actually a recursive acronym for Pip Installs Packages. Pip is important for the next stage of our install process, but first, we need to make sure we are running the latest version of pip (8.1.2 at the time of writing), which is exactly what this command does.

Installing a Python Virtual Environment

NOTE

If you are going to use Microsoft Visual Studio (VS), you can stop here and jump to [Appendix G, Developing Django with Visual Studio](#). VS only requires that you install Python, everything else VS does for you from inside the Integrated Development Environment (IDE).

All of the software on your computer operates interdependently—each program has other bits of software that it depends on (called **dependencies**) and settings that it needs to find the files and other software it needs to run (called **environment variables**).

When you are writing new software programs, it is possible (and common!) to modify dependencies and environment variables that your other software depends on. This can cause numerous problems, so should be avoided.

A Python virtual environment solves this problem by wrapping all the dependencies and environment variables that your new software needs into a file system separate from the rest of the software on your computer.

NOTE

Some of you who have looked at other tutorials will note that this step is often described as optional. This is not a view I support, nor is it supported by a number of Django's core developers.

NOTE

The advantages of developing Python applications (of which Django is one) within a virtual environment are manifest and not worth going through here. As a beginner, you just need

to take my word for it—running a virtual environment for Django development is not optional.

The virtual environment tool in Python is called `virtualenv` and we install it from the command line using `pip`:

```
pip install virtualenv
```

The output from your command window should look something like this:

```
C:\Users\nigel>pip install virtualenv
Collecting virtualenv
  Downloading virtualenv-15.0.2-py2.py3-
none-any.whl (1.8MB)
100% |#####
1.8MB 323kB/s
Installing collected packages: virtualenv
Successfully installed virtualenv-15.0.2
```

Once `virtualenv` is installed, you need to create a virtual environment for your project by typing:

```
virtualenv env_mysite
```

NOTE

Most examples on the Internet use `env` as your environment name. This is bad; principally because it's common to have several virtual environments installed to test different configurations, and `env` is not very descriptive. For example, you may be developing an application that must run on Python 2.7 and Python 3.4. Environments named `env_someapp_python27` and `env_someapp_python34` are going to be a lot easier to distinguish than if you had named them `env` and `env1`.

In this example, I have kept it simple as we will only be using one virtual environment for our project, so I have

used `env_mysite`. The output from your command should look something like this:

```
C:\Users\nigel>virtualenv env_mysite
Using base prefix

'c:\\users\\nigel\\appdata\\local\\program
s\\python\\python35-32'
New python executable in

C:\Users\nigel\env_mysite\Scripts\python.e
xe
Installing setuptools, pip, wheel...done.
```

Once `virtualenv` has finished setting up your new virtual environment, open Windows Explorer and have a look at what `virtualenv` created for you. In your home directory, you will now see a folder called `\env_mysite` (or whatever name you gave the virtual environment). If you open the folder, you will see the following:

```
\Include
\Lib
\Scripts
\src
```

`virtualenv` has created a complete Python installation for you, separate from your other software, so you can work on your project without affecting any of the other software on your system.

To use this new Python virtual environment, we have to activate it, so let's go back to the command prompt and

type the following:

```
env_mysite\scripts\activate
```

This will run the activate script inside your virtual environment's `\scripts` folder. You will notice your command prompt has now changed:

```
(env_mysite) C:\Users\nigel>
```

The `(env_mysite)` at the beginning of the command prompt lets you know that you are running in the virtual environment. Our next step is to install Django.

Installing Django

Now that we have Python and are running a virtual environment, installing Django is super easy, just type the command:

```
pip install django==1.8.13
```

This will instruct pip to install Django into your virtual environment. Your command output should look like this:

```
(env_mysite) C:\Users\nigel>pip
install django==1.8.13
Collecting django==1.8.13
  Downloading Django-1.8.13-py2.py3-
none-any.whl (6.2MB)
    100%
|#####| 6.2MB
107kB/s
```

```
Installing collected packages: django
Successfully installed django-1.8.13
```

In this case, we are explicitly telling pip to install Django 1.8.13, which is the latest version of Django 1.8 LTS at the time of writing. If you are installing Django, it's good practice to check the Django Project website for the latest version of Django 1.8 LTS.

NOTE

In case you were wondering, typing in `pip install django` will install the latest stable release of Django. If you want information on installing the latest development release of Django, see [Chapter 20, More On Installing Django](#).

For some post-installation positive feedback, take a moment to test whether the installation worked. At your virtual environment command prompt, start the Python interactive interpreter by typing `python` and hitting enter. If the installation was successful, you should be able to import the module `django`:

```
(env_mysite) C:\Users\nigel>python
Python 3.5.1 (v3.5.1:37a07cee5969, Dec
6 2015, 01:38:48)

[MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or
"license" for more
information.
>>> import django
>>> django.get_version()
1.8.13'
```

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 engines like PostgreSQL, MySQL, or Oracle, see [Chapter 21, Advanced Database Management](#).

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. 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.

I am assuming at this stage you are still running the virtual environment from the previous installation step. If not, you will have to start it again with:

```
env_mysite\scripts\activate\
```

From your virtual environment command line, run the following command:

```
django-admin startproject mysite
```

This will create a `mysite` directory in your current directory (in this case `\env_mysite\`). If you want to create your project in a directory other than the root, you can create a new directory, change into that directory and run the `startproject` command from there.

NOTE

Warning!

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

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. It's 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 your Django project in various ways. You can read all the details about `manage.py` on the Django Project website (for more information visit <https://docs.djangoproject.com/en/1.8/ref/django-admin/>).
- The inner `mysite/` directory. It's the Python package for your project. It's the name you'll use to import anything inside it (for

example, `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 at <https://docs.python.org/tutorial/modules.html#packages>, if you're a Python beginner.)
- `mysite/settings.py`, `settings/configuration` for this Django project. [Appendix D, Settings](#) will tell you all about how settings work.
- `mysite/urls.py`, the URL declarations for this Django project; a table of contents of your Django-powered site. You can read more about URLs in [Chapter 2, Views and Urlconfs](#) and [Chapter 7, Advanced Views and Urlconfs](#).
- `mysite/wsgi.py`, an entry-point for WSGI-compatible web servers to serve your project. See [Chapter 13, Deploying Django](#), 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 `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 `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 12, 2016-08:48:58
Django version 1.8.13, using settings
'mysite.settings'
Starting development server at
http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

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 (*Figure 1.3*). It worked!

NOTE

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 such as adding files don't trigger a restart, so you'll have to restart the server in these cases.



Django's welcome page

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

- The **Model(M)** is a model or representation of your data. It's not the actual data, but an interface to the data. The model allows you to pull data from your database without knowing 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's 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 interpretations 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 another might

vehemently defend the need for it to be on 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 use 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, Models](#).
- **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, Templates](#).
- **V stands for "View,"** the business logic layer. This layer contains the logic that accesses 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's views and learning the basics of serving web pages with Django.

Chapter 2. Views and URLconfs

In the previous chapter, I 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 (for example, <http://www.example.com/hello.html>). 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 we created 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 is an instance of the class `djangohttp.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, Advanced Views and Urlconfs](#).

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, Advanced Views and URLconfs](#).
- 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 `hellofoo` and `hellobar`, not just `hello`.

Similarly, if we had left off the initial caret character (that is, `hello/$`), Django would match any URL that ends with `hello/`, such as `foobarhello`.

If we had simply used `hello/`, without a caret or dollar sign, then any URL containing `hello/` would match, such as `foohello/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, Settings](#)).

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, *Introduction to Django and Getting Started*](#), 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://127.0.0.1:8000hello`. You should see the text **Hello World**-the output of your Django view (*Figure 2.1*).

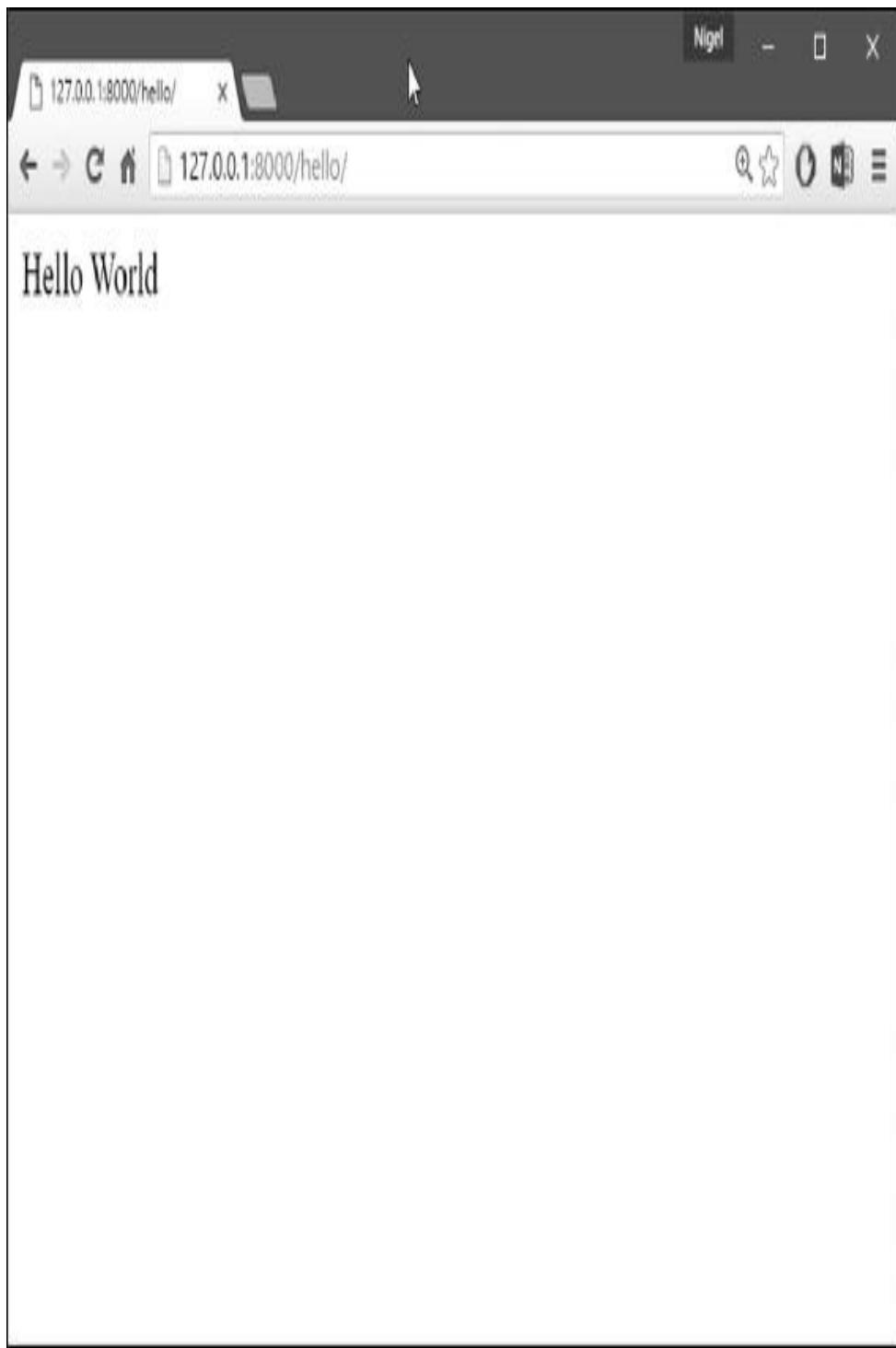


Figure 2.1: Hooray! Your first Django view

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. *Table 2.1* lists a selection of common symbols.

Table 2.1: Common regex 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)

<code>+</code>	One or more of the previous expression (for example, <code>\d+</code> matches one or more digits)
<code>[^/] +</code>	One or more characters until (and not including) a forward slash
<code>?</code>	Zero or one of the previous expression (for example, <code>\d?</code> matches zero or one digits)
<code>*</code>	Zero or more of the previous expression (for example, <code>\d*</code> matches zero, one or more than one digit)
<code>{1,3}</code>	Between one and three (inclusive) of the previous expression (for example, <code>\d{1,3}</code> matches one, two or three digits)

For more on regular expressions, see the Python regex documentation, visit

<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/>.

You should see a **Page not found** message (*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**.

I'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://127.0.0.1:8000hello` 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`, and so on. 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 auto-generated `settings.py` contains a `ROOT_URLCONF` setting that points to the auto-generated

`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 (that is, the content of the web page). In summary:

- A request comes in to `hello`.
- Django determines the root URLconf by looking at the `ROOT_URLCONF` setting.
- Django looks at all of the URLpatterns in the URLconf for the first one that matches `hello`.
- If it finds a match, it calls the associated view function.

- The view function returns an `HttpResponse`.
- 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 is 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, 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 HttpResponse(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 following 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 `"2015-07-15 18:12:39.002731"`. This will result in an HTML string such as `"<html><body>It is now 2015-07-15 18:12:39.002731.</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 likely because

the default time zone in your `settings.py` is set to **UTC**.

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.

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 the 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 `books243/` and `books81196/`. 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 `timeplus/1/` displays the date/time one hour into the future, the page `timeplus/2/` displays the date/time two hours into the future, the page `timeplus/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),
]
```

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 or three hours.

If we decided to create a page that displayed the time four hours into the future, we'd have to create a separate view and URLconf line for that, furthering the duplication.

How, then do we design our application to handle arbitrary hour offsets? The key is to use wildcard URLpatterns. As I 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),
    # ...
]
```

(I'm using the `# ...` to imply there might be other URLpatterns that have been trimmed from this example.) This new URLpattern will match any URL such as `timeplus/2/`, `timeplus/25/`, or even `timeplus/100000000000/`. 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),  
]
```

NOTE

If you're experienced in another web development platform, you may be thinking, "Hey, let's use a query string parameter!"—something like `timeplus?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 I'll tell you how in [Chapter 7, Advanced Views and URLconfs](#)), but one of Django's core philosophies is that URLs should be beautiful. The URL `timeplus/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.

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,
HttpResponse
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 take a closer look at this code.

The view function, `hours_ahead`, takes two

parameters: `request` and `offset`:

- `request` is an `HttpRequest` object, just as in `hello` and `current_datetime`. I'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 `timeplus/3/`, then `offset` would be the string '`3`'. If the requested URL were `timeplus/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`'.

I 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 a URLconf. I cover that in [Chapter 7, Advanced Views and URLconfs.](#))

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 `djongo.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://127.0.0.1:8000timeplus/3/` to verify it works.

Then try `http://127.0.0.1:8000timeplus/5/`.

Then `http://127.0.0.1:8000timeplus/24/`.

Finally, visit

`http://127.0.0.1:8000timeplus/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://127.0.0.1:8000timeplus/` (with no hour designation) should also throw a 404.

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 [`timeplus/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*).

Nigel

TypeError at /time/plus/3/ X

127.0.0.1:8000/time/plus/3/

TypeError at /time/plus/3/

unsupported type for timedelta hours component: str

Request Method: GET

Request URL: http://127.0.0.1:8000/time/plus/3/

Django Version: 1.9.7

Exception Type: TypeError

Exception Value: unsupported type for timedelta hours component: str

Exception Location: C:\Users\Nigel\OneDrive\Documents\Visual Studio 2015\Projects\mysite\mysite\mysite\views.py in hours_ahead, line 22

Python Executable: C:\Users\Nigel\OneDrive\Documents\Visual Studio 2015\Projects\mysite\mysite\env\Scripts\python.exe

Python Version: 3.4.3

Python Path: ['C:\\Users\\Nigel\\OneDrive\\Documents\\Visual Studio 2015\\Projects\\mysite\\mysite', 'C:\\WINDOWS\\SYSTEM32\\python34.zip', 'C:\\Users\\Nigel\\OneDrive\\Documents\\Visual Studio 2015\\Projects\\mysite\\mysite\\env\\DLLs', 'C:\\Users\\Nigel\\OneDrive\\Documents\\Visual Studio 2015\\Projects\\mysite\\mysite\\env\\lib', 'C:\\Users\\Nigel\\OneDrive\\Documents\\Visual Studio 2015\\Projects\\mysite\\mysite\\env\\Scripts', 'C:\\Python34\\Lib', 'C:\\Python34\\DLLs', 'C:\\Users\\Nigel\\OneDrive\\Documents\\Visual Studio 2015\\Projects\\mysite\\mysite\\env', 'C:\\Users\\Nigel\\OneDrive\\Documents\\Visual Studio 2015\\Projects\\mysite\\mysite\\env\\lib\\site-packages']

Server time: Tue, 7 Jun 2016 15:45:31 +1000

Traceback [Switch to copy-and-paste view](#)

```
[REDACTED]
```

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 an 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 dparse (for more information visit <http://www.dparse.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 F, Request and Response Objects](#), has a complete reference of all the information a request object contains.
- Following to the **Request information** section, the **Settings** section lists all of the settings for this particular Django installation. All the available settings are covered in detail in [Appendix D, Settings](#).

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.

The Django error page is also really useful if you are the type of programmer who likes to debug with the help of carefully placed `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" % dt.strftime("%A %B %d %Y %I:%M %p")
```

```
%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. I'll explain how to deactivate debug mode in [Chapter 13, *Deploying Django*](#). For now, just know that every Django project is in debug mode automatically when you start it. (Sounds 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. I've done that to keep things simple while I 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 into 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.

NOTE

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:

```
<html>
<head><title>Ordering
notice</title></head>
<body>
```

```

<h1>Ordering notice</h1>

<p>Dear {{ person_name }},</p>

<p>Thanks for placing an order from {{ company }}. It's scheduled to ship on {{ ship_date|date:"F j, Y" }}.</p>
<p>Here are the items you've ordered:</p>
<ul>
  {% for item in item_list %}<li>{{ item }}</li>{% endfor %}
</ul>

{% if ordered_warranty %}
  <p>Your warranty information will be included in the packaging.</p>
{% else %}
  <p>You didn't order a warranty, so you're on your own when the products inevitably stop working.</p>
{% endif %}

<p>Sincerely,<br >{{ company }}<p>

</body>
</html>

```

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 (for example, {{ 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 (for example, {% 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, Built-in Template Tags and Filters](#), 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, Advanced Templates](#).

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` (for more information visit <http://jinja.pocoo.org/>). 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, Advanced Templates](#). 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 you created in [Chapter 1, *Introduction to Django and Getting Started*](#), type `python manage.py shell` to start the interactive interpreter.

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:

```
<django.template.base.Template object at  
0x030396B0>
```

That `0x030396B0` 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.'
```

NOTE

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.

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, Advanced Templates](#).

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 = """<p>Dear {{  
person_name }}</p>  
...  
... <p>Thanks for placing an order from {{  
company }}. It's scheduled to  
... ship on {{ ship_date|date:"F j, Y"  
}}.</p>  
...  
... {% if ordered_warranty %}  
... <p>Your warranty information will be  
included in the packaging.</p>  
... {% else %}  
... <p>You didn't order a warranty, so  
you're on your own when  
... the products inevitably stop  
working.</p>  
... {% endif %}
```

```
...
... <p>Sincerely,<br >{{ company }}</p>"""
>>> 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)
u"<p>Dear John Smith,</p>\n\n<p>Thanks for
placing an order from Outdoor
Equipment. It's scheduled to\ncship on July
2, 2015.</p>\n\n<p>You
didn't order a warranty, so you're on your
own when\ncthe products
inevitably stop
working.</p>\n\n<p>Sincerely,<br
>Outdoor Equipment
<p>"
```

- 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—that is, 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 to 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
```

```
>>> 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 of 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`.

NOTE

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 (for example, `foo["bar"]`)
- Attribute lookup (for example, `foo.bar`)
- Method call (for example, `foo.bar()`)
- List-index lookup (for example, `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's not possible to pass arguments to method calls accessed from within templates. Data should be calculated in views and then passed 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 preceding 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 behavior 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` (that is, 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 %}
    <p>Welcome to the weekend!</p>
{% endif %}
```

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

```
{% if today_is_weekend %}
    <p>Welcome to the weekend!</p>
{% else %}
    <p>Get back to work.</p>
{% 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 %}
    <p>Athletes should be out of the locker room soon! </p>
{% elif ... %}
    ...
{% else %}
    <p>No athletes. </p>
{% endif %}
```

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 %}
    <p>Both athletes and coaches are available. </p>
{% endif %}

{% if not athlete_list %}
    <p>There are no athletes. </p>
{% endif %}

{% if athlete_list or coach_list %}
    <p>There are some athletes or some coaches. </p>
{% endif %}

{% if not athlete_list or coach_list %}
    <p>There are no athletes or there are some coaches. </p>
{% endif %}

{% if athlete_list and not coach_list %}
    <p>There are some athletes and
```

```
>perhaps are some athletes and  
absolutely no coaches. </p>  
{% endif %}
```

Use of both **and** and **or** clauses within the same tag is allowed, with **and** having higher precedence than **or** for example:

```
{% if athlete_list and coach_list or  
cheerleader_list %}
```

will be interpreted like:

```
if (athlete_list and coach_list) or  
cheerleader_list
```

TIP

NOTE: Use of actual parentheses in the if tag is invalid syntax.

If you need parentheses 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  
%}  
        <p>We have athletes, and either  
coaches or cheerleaders! </p>  
    {% 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`:

```
<ul>  
  {% for athlete in athlete_list %}  
    <li>{{ athlete.name }}</li>  
  {% endfor %}  
</ul>
```

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

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

```
...</li></ul>
```

```
{% endfor %}
```

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

```
{% for athlete in athlete_list %}
    <h1>{{ athlete.name }}</h1>
    <ul>
        {% for sport in athlete.sports_played
    %}
        <li>{{ sport }}</li>
        {% endfor %}
    </ul>
{% endfor %}
```

If you need to loop over a list of lists, you can unpack the values in each sub list 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 %}
    <p>There is a point at {{ x }},{{ y }}
</p>
{% 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 %}
        <p>{{ athlete.name }}</p>
    {% endfor %}

    {% else %}
        <p>There are no athletes. Only
        computer programmers.</p>
    {% 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 %}
    <p>{{ athlete.name }}</p>
{% empty %}
    <p>There are no athletes. Only
    computer programmers.</p>
{% 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 %}
    <p>{{ forloop.counter }}: {{ item }}</p>
{% 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 %}<li
        class="first">
    {% else %}<li>{% endif %}
        . . .
```

```
    {{ object }}
  </li>
  {% 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 preceding 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.

NOTE

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 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 %}  
    <h1>Welcome!</h1>  
{% endifequal %}
```

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

```
{% ifequal section 'sitenews' %}  
    <h1>Site News</h1>  
{% endifequal %}  
  
{% ifequal section "community" %}  
    <h1>Community</h1>  
{% endifequal %}
```

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

```
{% ifequal section 'sitenews' %}  
    <h1>Site News</h1>  
{% else %}  
    <h1>No News Here</h1>  
{% 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 (that is, 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, Built-in Template Tags and Filters](#) covers the rest.

- `addslashes`: Adds a backslash before any backslash, single quote, or double quote. This is useful for escaping strings. For example:

```
{% value|addslashes %}
```

- `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, Built-in Template Tags and Filters](#).
- `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:

1. Separate logic from presentation
2. Discourage redundancy
3. Be decoupled from HTML
4. XML is bad
5. Assume designer competence
6. Treat whitespace obviously
7. Don't invent a programming language
8. Ensure safety and security
9. Extensible

Following is the explanation for this:

1. **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.

2. **Discourage redundancy**

The majority of dynamic websites use some sort of common site-wide design—a common header, footer, navigation bar, and so on. 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.

3. 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.

4. 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.

5. 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.

6. 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.

7. 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.

8. 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.

9. 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 whole-heartedly 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 of 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 programmer's 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 HttpResponseRedirect  
import datetime  
  
def current_datetime(request):  
  
    now = datetime.datetime.now()  
    html = "<html><body>It is now %s.  
</body></html>" % now  
    return HttpResponseRedirect(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 HttpResponseRedirect  
import datetime  
  
def current_datetime(request):  
  
    now = datetime.datetime.now()  
    t = Template("<html><body>It is now {{  
current_date }}.  
</body></html>")  
    html =
```

```
t.render(Context({'current_date': now}))  
return HttpResponseRedirect(reverse('home'))
```

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

homedjangouser/templates/mytemplate.html:

```
from django.template import Template,  
Context  
from django.http import HttpResponseRedirect  
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('homedjangouser/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 I mentioned last chapter, when I 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.DjangoTemp`

`lates` and
`django.template.backends.jinja2.Jinja2`.

Since most engines load templates from files, the top-level configuration for each engine contains three 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': [  
    'homehtml/example.com',  
    'homehtml/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.
- If you want to have a set of master templates at project root, for example, `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 preceding 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 HttpResponseRedirect
```

```
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    t =
    get_template('current_datetime.html')
    html =
    t.render(Context({'current_date': now}))
    return HttpResponseRedirect(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 '`homedjango/mysite/templates`', the preceding `get_template()` call would look for the template `homedjango/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 (for example, `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*).



Figure 3.1: Missing template error page.

This error page is similar to the one I explained in [Chapter 2, Views and Urlconfs](#) 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 (for example, **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. I 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.

Template subdirectories

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, I recommend doing so; some more advanced Django features (such as the generic views system, which we cover in [Chapter 10, Generic Views](#)) 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.htm
l')
```

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

<html><body>

{% include "includes/nav.html" %}

<h1>{{ title }}</h1>
</body></html>

# includes/nav.html

<div id="nav">
    You are in: {{ current_section }}
</div>
```

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 site wide 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:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML  
4.01//EN">  
<html lang="en">  
<head>
```

```
<title>The current time</title>
</head>
<body>
    <h1>My helpful timestamp site</h1>
    <p>It is now {{ current_date }}.</p>

    <hr>
    <p>Thanks for visiting my site.</p>
</body>
</html>
```

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, Views and Urlconfs](#)? If we want again to make a nice, valid, full HTML template, we'd create something like:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML
4.01//EN">
<html lang="en">

    <head>
        <title>Future time</title>
    </head>

    <body>
        <h1>My helpful timestamp site</h1>
        <p>In {{ hour_offset }} hour(s), it
will be {{ next_time }}.</p>

        <hr>
        <p>Thanks for visiting my site.</p>
    </body>
</html>
```

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`:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML  
4.01//EN">  
<html lang="en">  
<head>
```

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

```
<hr>  
    <p>Thanks for visiting my site.</p>  
</body>  
</html>
```

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.

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:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML  
4.01//EN">  
<html lang="en">  
  
<head>  
    <title>{% block title %}{% endblock %}</title>  
</head>  
  
<body>  
    <h1>My helpful timestamp site</h1>  
    {% block content %}{% endblock %}  
    {% block footer %}  
    <hr>  
    <p>Thanks for visiting my site.</p>  
    {% endblock %}  
</body>  
</html>
```

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 %}
<p>It is now {{ current_date }}.</p>
{% endblock %}
```

While we're at it, let's create a template for the `hours_ahead` view from this chapter. (If you're following along with code, I'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 %}

<p>In {{ hour_offset }} hour(s), it will
be {{ next_time }}.</p>
{% 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 fall-back.

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 (for example, `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 websites are database-driven: the content of the website 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, Views and Urlconfs](#), we covered the fundamentals of building dynamic websites 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 website 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 websites provide some combination of the two. www.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 websites 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, Views and Urlconfs](#), 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, let's explore the initial configuration that was added to `settings.py` when we created the application:

```
# Database
# 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`: It 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`: It 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 21, Advanced Database Management](#).

Your first app

Now that you've verified that the connection is working, it's time to create a **Django app**--a bundle of Django code, including models and views, that live together in a single Python package and represent 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, *Introduction to Django and Getting Started*](#), 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 live 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 website, you may use only a single app. If you're building a complex website 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 (this is the directory where your `manage.py` file is, not the `mysite` app 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 [Chapter 1, *Introduction to Django and Getting Started*](#), 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 somehow, 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, you 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 21](#), *Advanced Database Management*.

Your first model

As an ongoing example in this chapter and the next chapter, I'll focus on a basic book/author/publisher data layout. I 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!

I'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 website.
- 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 =
```

```
class Address(models.Model):
    street = 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 (example, `CharField`) corresponds to the database column type (example, `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, Database API Reference](#). 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 `INSTALLED_APPS`, so the setting ends up looking like this ('`books`' refers to the "books" app we're working on):

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

```
'django.contrib.messages',
'django.contrib.staticfiles',
'books',
)
```

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. Anytime 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 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, Database API Reference](#).
- 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 (example, 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, and so on.) into your database schema. They're designed to be mostly automatic, however, there are some caveats. For more information on migrations, see [Chapter 21, Advanced Database Management](#).

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

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`, and so on.

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

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 auto-incrementing 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 (that is, 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 9, Advanced Models](#). 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 an `objects` manager; you'll use it anytime 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, Generic View Reference](#), 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="p
ress")
[<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
```

```
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, Generic View Reference](#), 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 `class Meta`, which is a class that's embedded within the `Publisher` class definition (that is, 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, Database API Reference](#), 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='US
A')
2
```

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').de
lete()
>>> 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').de
lete()
```

What's next?

Having read this chapter, you have enough knowledge of Django models to be able to write basic database applications. Chapter 9, *Advanced Models*, 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 21, *Advanced Database Management*, will give you advice about integrating with legacy databases. You might rely on site users to supply your data, in which case Chapter 6, *Forms*, 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 websites, 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 website 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, *Introduction to Django and Getting Started*, 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.

NOTE

If you are using Visual Studio, you don't need to complete this next step at the command line, you can just add a superuser from the **Project** menu tab within Visual Studio.

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

In Django 1.8, the django admin site is activated by default. Let's start the development server and explore it. Recall from previous chapters 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-for example, `http://127.0.0.1:8000admin`. You should see the admin's login screen (*Figure 5.1*).

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 administrator** index page (*Figure 5.2*).

You should see two 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.

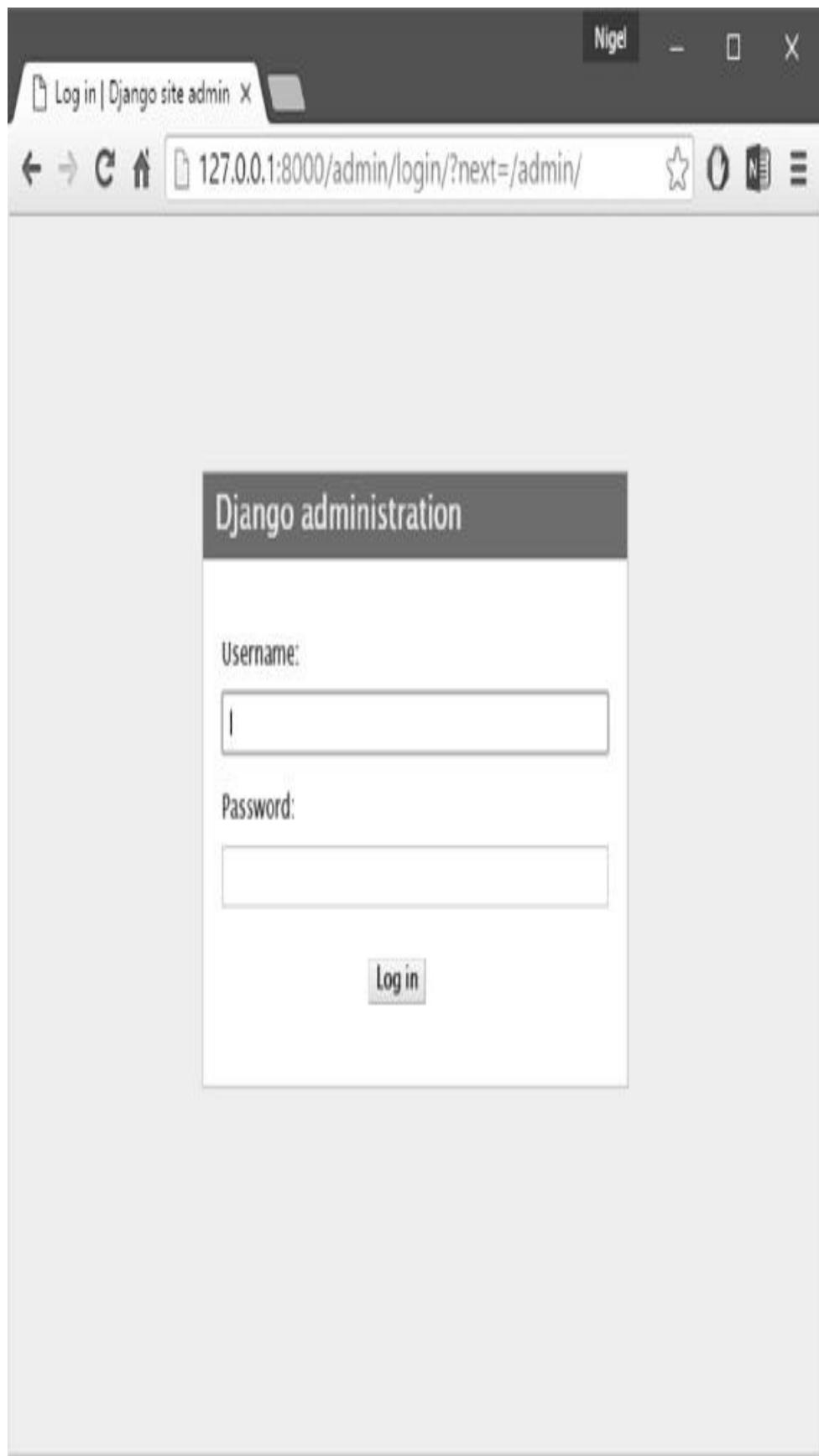


Figure 5.1: **Django administrator** login screen



Figure 5.2: **Django administrator** home page

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 screenshot shows the Django administration interface for the user model. At the top, there's a navigation bar with tabs for 'Select user to change' (selected), 'Create user', and 'List users'. Below that is a browser header showing the URL '127.0.0.1:8000/admin/auth/user/'. The main title is 'Django administration' with a welcome message 'Welcome, admin. View site / Change password / Log out'. A breadcrumb trail shows 'Home > Authentication and Authorization > Users'. The main content area is titled 'Select user to change' with a search bar and a 'Search' button. To the right is a 'FILTER' sidebar with three sections: 'By staff status' (All, Yes, No), 'By superuser status' (All, Yes, No), and 'By active' (All, Yes, No). A table lists one user: 'admin' with email 'nigel@masteringdjango.com' and staff status '0'. There are also buttons for 'Add user +' and 'Go'.

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 (*Figure 5.4*).

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.

The screenshot shows the Django administration interface for changing a user. The URL in the browser is `127.0.0.1:8000/admin/auth/user/1/`. The page title is "Change user".

Username: admin
Required: 30 characters or fewer. Letters, digits and `!@#$%^&*`, only.

Password: algorithm: pbkdf2_sha256 iterations: 20000 salt: h7AwXP***** hash: XsXXJR*****
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: [empty input field]

Last name: [empty input field]

Email address: nigel@masterngdjango.com

Permissions

Figure 5.4: The user edit form

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*.

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*).

The screenshot shows a browser window displaying the Django administration interface. The title bar reads "Add user | Django site admin". The address bar shows the URL "127.0.0.1:8000/admin/auth/user/add/". The main content area is titled "Django administration" with a sub-header "Welcome, admin. View site / Change password / Logout". Below this, the breadcrumb navigation shows "Home : Authentication and Authorization : Users : Add user". The main form is titled "Add user" and contains instructions: "First, enter a username and password. Then, you'll be able to edit more user options.". A red error message box at the top left says "Please correct the errors below." Three fields are highlighted with red error bars: "Username", "Password", and "Password confirmation". Each field has a "This field is required." message above its input field. Below the "Password confirmation" field is a note: "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 large "SAVE" button.

Figure 5.5: An edit form displaying errors



Figure 5.6: An object history page

NOTE

How the Admin Site Works

Behind the scenes, how does the admin site work? It's pretty straightforward. When Django loads at server startup, it runs the `admin.autodiscover()` function. In earlier versions of Django, you used to call this function from `urls.py`, but now Django runs it automatically. 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.

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, Models](#), 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 development server 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 4, Models](#), 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.

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, Models](#), 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 author's 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 (for example 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, Models](#)) add an explicit `NOT NULL` to each column definition. For example, here's the generated statement for our `Author` model, from [Chapter 4, Models](#):

```
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 (for example `DateField`, `TimeField`, `DateTimeField`) or numeric field (for example `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 (for example `"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 model admin 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, Models, 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=30)
```

```
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*.)

The screenshot shows a Django administration interface titled "Select author to change". The URL in the browser is `127.0.0.1:8000/admin/books/author/`. The page displays a list of three authors: Peter, Smith, peter@example.com; Barry, Jones, barry@example.com; and Nigel, George, nigel@example.com. The columns are labeled "First name", "Last name", and "Email". An "Add author" button is visible at the top right. The top navigation bar includes links for "View site", "Change password", and "Log out".

	First name	Last name	Email
<input type="checkbox"/>	Peter	Smith	peter@example.com
<input type="checkbox"/>	Barry	Jones	barry@example.com
<input type="checkbox"/>	Nigel	George	nigel@example.com

Figure 5.8: The author change list page after `list_display` added

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` added

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 two 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.

The screenshot shows the Django admin interface for the 'books' model. The title bar says 'Select book to change'. The address bar shows the URL '127.0.0.1:8000/admin/books/book/'. The main header says 'Django administration' and 'Welcome, Nigel. View site / Change password / Log out'. Below it, the breadcrumb navigation shows 'Home > Books > Books'. The main content area is titled 'Select book to change' with a 'Add book +' button. On the left, there's a table with two rows: 'A Really Cool Book' and 'Mastering Django: Core'. On the right, there's a 'Filter' sidebar with a dropdown menu set to 'By publication date' and options like 'Any date', 'Today', 'Past 7 days', 'This month', and 'This year'. At the bottom left, it says '2 books'.

Title	Publisher	Publication date
A Really Cool Book	Book Publishing Inc	June 7, 2016
Mastering Django: Core	GNW Independent Publishing	Sept. 9, 2015

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, Models](#))-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 model's `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.*)



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 some 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 it still 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 display 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.

Add book | Django site X

127.0.0.1:8000/admin/books/book/add/

Welcome, Nigel. View site / Change password / Log out

Home > Books > Books > Add book

Add book

Title:

Authors:

Available authors:

Nigel George
Barry Jones
Peter Smith

Hold down "Control", or "Command" on a Mac, to select more than one.

Publisher:

Publication date: Today

This screenshot shows the Django admin interface for adding a new book. At the top, it displays the URL 127.0.0.1:8000/admin/books/book/add/. The main content area is titled 'Add book'. It has a 'Title' field with an empty input box. Below that is a 'Authors' section with two panes: 'Available authors' on the left containing three names (Nigel George, Barry Jones, Peter Smith) and 'Chosen authors' on the right which is currently empty. There are 'Choose all' and 'Remove all' buttons below these panes. A note at the bottom of this section says 'Hold down "Control", or "Command" on a Mac, to select more than one.' Below the authors section is a 'Publisher' field with a dropdown menu and a small icon. At the bottom, there are three save buttons: 'Save and add another', 'Save and continue editing', and a highlighted 'Save' button.

Figure 5.13: The book edit form after adding
`filter_horizontal`

I'd highly recommend using `filter_horizontal` for any `ManyToManyField` that has more than ten 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`:

```
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',)
```

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*.

Add book | Django site X

127.0.0.1:8000/admin/books/book/add/

Django administration

Welcome, Nigel. View site / Change password / Log out

Home > Books > Books > Add book

Add book

Title:

Authors:

Available authors: Filter

Nigel George
Barry Jones
Peter Smith

Chosen authors:

Hold down "Control", or "Command" on a Mac, to select more than one.

Publisher:

Publication date: Today

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, *User Authentication in Django*](#), we'll cover how to manage users site-wide (that is, 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 (that is, whether that user is considered a staff member in your organization). Since this same user system can be used to control access to public (that is, non-admin) sites (see [Chapter 11, User Authentication in Django](#)), 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 (for example 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

Warning!

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 I 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-modelling mistakes or other problems with your models.
- **Managing acquired data:** For applications that rely on data coming from external sources (for example 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 websites, 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

I introduced [HttpRequest](#) objects in [Chapter 2, Views and URLconfs](#), when we first covered view functions, but I 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 (that is, 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 (*Table 6.1*).

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 (that is, the "domain," in common parlance).	" <code>127.0.0.1:8000</code> " or " <code>www.example.com</code> "
<code>request.get_full_path()</code>	The <code>path</code> , plus a query string (if available).	" <code>hello?</code> <code>print=true</code> "

<code>request.is_secure()</code>	<code>True</code> if the request was made via HTTPS. Otherwise, <code>False</code> .	<code>True</code> or <code>False</code>
----------------------------------	--	---

Table 6.1: HttpRequest methods and attributes

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, for example, "12.345.67.89". (If the request has passed through any proxies, then this might be a comma-separated list of IP addresses, for example, "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)
```

I 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.

`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.

NOTE

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.

A simple form-handling example

Continuing the 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 you learned in [Chapter 3, *Templates*](#), 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, *Templates*](#), 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, *Advanced Views and Urlconfs*](#)). Now, if you run the development server 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 HttpResponse

# ...

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:

- 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`.
- 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 I pointed out in the `request.META` section preceding, 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 HttpResponseRedirect(message)
```

Query string parameters

Because `GET` data is passed in the query string (for example, `search?q=django`), you can use `request.GET` to access query string variables. In [Chapter 2, Views and Urlconfs](#), introduction of Django's URLconf system, I compared Django's pretty URLs to more traditional PHP/Java URLs such as `timeplus?hours=3` and said I'd show you how to do the latter in [Chapter 6, Forms](#). 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 the

<http://www.w3.org/2001/tag/doc/whenToUseGet.html> site) 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, Models, and Appendix B, Database API Reference](#)), 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, I've shown you the simplest thing that could possibly work. Now I'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 preceding to 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 I'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, they'll see the search form with no error message. If a user submits the form with an empty value for '`q`', they'll see the search form with an error message. And, finally, if a user submits the form with a non-empty value for '`q`', they'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 websites:

- *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.*

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']
    else:
        errors.append('Enter a search term.')
    if len(q) > 20:
        errors.append('Please enter at most 20 characters.')
    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 %}</body> </html>

```
<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 preceding 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><inp
ut 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('contactthanks/')
    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 Forgeries. Thankfully, you don't have to worry too hard, because Django comes with a very easy-to-use system for protecting against it. In short, all `POST` forms that are targeted at internal URLs should use the `{% csrf_token %}` template tag. More details about

`{% csrf_token %}` can be found in [Chapter 19, *Security in Django*](#).

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.)

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'](mailto:'noreply%40example.com')),
                [['siteowner@example.com'](mailto:'siteowner%40example.com')],
                )
            return
        HttpResponseRedirect('contactthanks/')
    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():

        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.

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, *Models*](#)). 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>`, and so on.) 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.message }}
        </div>
    </form>

```

```
    {{ form.email.errors }}  
    <label for="id_email">Your e-  
mail address:</label>  
    {{ form.email }}  
    </div>  
    <div class="field">  
        {{ form.message.errors }}  
        <label  
for="id_message">Message:</label>  
        {{ form.message }}  
        </div>  
        {% csrf_token %}  
        <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, Advanced Views and URLconfs](#), to [Chapter 13, Deploying Django](#), goes into more detail about advanced Django usage, including how to deploy a Django application ([Chapter 13, Deploying Django](#)).

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 [Chapters 7, Advanced Views and URLconfs](#), by doubling back and taking a closer look at views and URLconfs (introduced first in [Chapter 2, Views and URLconfs](#)).

Chapter 7. Advanced Views and URLconfs

In Chapter 2, Views and URLconfs, we explained the basics of Django's view functions and URLconfs. This chapter goes into more detail about advanced functionality in those two pieces of the framework.

URLconf Tips and 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, Views and URLconfs:

```
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)
```

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

As explained in [Chapter 2, Views and URLconfs](#), 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 groupsPreview

The above example used simple, non-named regular-expression groups (via parenthesis) to capture bits of the URL and pass them as positional arguments to a view.

In more advanced usage, it's possible to use named regular-expression groups to capture URL bits and pass them as keyword arguments to a view.

In Python regular expressions, the syntax for named regular-expression groups is `(?P<name>pattern)`, where `name` is the name of the group and `pattern` is some pattern to match.

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),
]
```

TIP

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 `reviews2005/03/` would match the third entry in the

list. Django would call the function
`views.month_archive(request, '2005', '03').`

- `reviews2005/3/` would not match any URL patterns, because the third entry in the list requires two digits for the month.
- `reviews2003/` 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.
- `reviews2003` would not match any of these patterns, because each pattern requires that the URL end with a slash.
- `reviews2003/03/03/` 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 `reviews2005/03/` would call the function
`views.month_archive(request, year='2005', month='03')`,
instead of `views.month_archive(request, '2005', '03')`.
- A request to `reviews2003/03/03/` 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 view's 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`, and so on—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 view's 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.

NOTE

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')`

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 website 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()`, 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 `creditreports/` 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 `reviews2005/`, 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 14, Generating Non-HTML Content](#)).

NOTE

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, and so on) or for handling of the navigation flow on the server side (redirections, and so on)

It is strongly desirable to avoid hard-coding 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 (for example 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 [reviewsnnnn/](#). 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 }} Archive</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

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:

1. **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`.
2. **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 (for example "`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.
3. The current application can also be specified manually as an argument to the `reverse()` function.
4. 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`").
5. If there is no default application instance, Django will pick the last deployed instance of the application, whatever its instance name may be.

6. 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, Advanced Templates](#), 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, *Templates*](#):

- 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 `}`):
- 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, *Templates*](#). 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, *Templates*](#), 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's passed a different context instance explicitly. 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, *Templates*, I 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 {
        'ann': 'My ann'
```

```

        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 19, Security in Django](#)).

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.
- `DEFAULT_MESSAGE_LEVELS`: A mapping of the message level names to their numeric value.

Guidelines for writing our 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 19, Security in Django](#)). 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 unharmed 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 autoescaping 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 autoescaped 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 autoescaping 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 autoescaping 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 autoescaping when it would otherwise be disabled. Here is an example template:

```
Autoescaping is on by default. Hello {{ name }}

{% autoescape off %}
    This will not be autoescaped: {{ data }}.

Nor this: {{ other_data }}
{% autoescape on %}
    Autoescaping applies again: {{ name }}
    {% endautoescape %}
    {% endautoescape %}
```

The autoescaping 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 autoescaping 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 autoescaping 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 autoescaping is enabled, then add an `escape` filter to any variable that needs escaping. When autoescaping is on, there's no danger of the `escape` filter double-escaping data—the `escape` filter does not affect autoescaped 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 &lt; 2" }}
```

... rather than

```
 {{ data|default:"3 < 2" }} <== Bad! Don't  
do this.
```

This doesn't affect what happens to data coming from the variable itself. The variable's contents are still automatically escaped, if necessary, because they're beyond the control of the template author.

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.djangoprojectTemplates',
        'DIRS': [
            'homehtml/templates/lawrence.com',
            'homehtml/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

`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

`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']
```

`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.reviews`.

You must then make sure that your `myproject.reviews` comes before `dango.contrib.admin` in `INSTALLED_APPS`, otherwise `dango.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.DjangoTem  
        plates',  
    '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`, and so on. 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 %}` tag to work.

NOTE

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. I'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()
```

Custom template tags and filters

Django's template language comes with a wide variety of built-in tags and filters designed to address the presentation logic needs of your application. Nevertheless, you may find yourself needing functionality that is not covered by the core set of template primitives.

You can extend the template engine by defining custom tags and filters using Python, and then make them available to your templates using the `{% load %}` tag.

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

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.

1. A Python function.
2. The compilation functiona 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 autoescaping and filters and time zones below.

TEMPLATE FILTERS THAT EXPECT STRINGS

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 AUTOESCAPING

When writing a custom filter, give some thought to how the filter will interact with Django's autoescaping 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 autoescaping 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

autoescaping 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; or
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.

In this first case, you can let Django take care of all the autoescaping 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 autoescaping 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 `&`, 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`).

In the second case, you want to mark the output as safe from further escaping so that your HTML mark-up 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 autoescaping is in effect.

The idea is to write filters that can operate in templates where autoescaping is either on or off in order to make things easier for your template authors.

In order for your filter to know the current autoescaping 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 autoescaping 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 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 autoescaping issues and return a safe string, the `is_safe` flag won't change anything either way.

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.

NOTE

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 }}
```

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

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_st
ring)
```

A few things to note about the `simple_tag` helper function:

- Checking for the required number of arguments, and so on, 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

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 when first creating the function:

```
from django.template.loader import
get_template
t = get_template('book_snippet.html')
```

```
register.inclusion_tag(t)(show_reviews)
```

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. Like `simple_tag`, `inclusion_tag` functions may also accept any number of positional or keyword arguments.

ASSIGNMENT TAGS

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_st
ring)
```

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 %T" as
```

```
    {{> gec_current_time }}  
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:

        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.form
at_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.

Autoescaping Considerations

The output from template tags is **not** automatically run through the autoescaping 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 `autoescape` 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 `autoescape` 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('smal
    l_fragment.html')
    return t.render(Context({'var': obj}),
    autoescape=context.autoescape)
```

If we had neglected to pass in the current `context.autoescape` value to our new `Context` in this example, the results would have *always* been automatically escaped, which may not be the desired behavior if the template tag is used inside a

`{% autoescape off %}` block.

Thread-safety Considerations

Once a node is parsed, its `render` method may be called any number of times. Since Django is sometimes run in multi-threaded environments, a single node may be simultaneously rendering with different contexts in response to two separate requests.

Therefore, it's important to make sure your template tags are thread safe. To make sure your template tags are thread safe, you should never store state information on the node itself. For example, Django provides a built-in `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, cycler):
        self.cycle_iter =
itertools.cycle(cycler)

    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:

- Thread 1 performs its first loop iteration, `CycleNode.render()` returns 'row1'
- Thread 2 performs its first loop iteration, `CycleNode.render()` returns 'row2'
- Thread 1 performs its second loop iteration, `CycleNode.render()` returns 'row1'
- Thread 2 performs its second loop iteration, `CycleNode.render()` returns 'row2'

The `CycleNode` is iterating, but it's iterating globally. As far as Thread 1 and Thread 2 are concerned, it's always returning the same value. This is obviously not what we want!

To address this problem, Django provides a `render_context` that's associated with the `context` of the template that is currently being rendered. The `render_context` behaves like a Python dictionary, and should be used to store `Node` state between invocations of the `render` method. Let's refactor our `CycleNode` implementation to use the `render_context`:

```
class CycleNode(template.Node):
    def __init__(self, cyclevars):
        self.cyclevars = cyclevars

    def render(self, context):
        if self not in
context.render_context:
            context.render_context[self] =
itertools.cycle(self.cyclevars)
            cycle_iter =
context.render_context[self]
        return next(cycle_iter)
```

Note that it's perfectly safe to store global information that will not change throughout the life of the `Node` as an attribute.

In the case of `CycleNode`, the `cyclevars` argument doesn't change after the `Node` is instantiated, so we don't need to put it in the `render_context`. But state information that is specific to the template that is currently being rendered, like the current iteration of the `CycleNode`, should be stored in the `render_context`.

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:

- The name of the template tag-a string. If this is left out, the name of the compilation function will be used.
- 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 datetime:

```
<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("%%")
```

```
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:

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.form
at_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
            ("Expected either a double or single
quote at the start of the argument")
```

```
        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 %}` 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 %}` is slightly different in that it allows broken template tags to appear between `{% comment %}` and `{% endcomment %}`. It does so by calling `parser.skip_past('endcomment')` instead of `parser.parse(('endcomment',))` followed by `parser.delete_first_token()`, thus avoiding the generation of a node list.

`parser.parse()` takes a tuple of names of block tags "to parse until". It returns an instance of `django.template.NodeList`, which is a list of all `Node` objects that the parser encountered "before" it encountered any of the tags named in the tuple. In `"nodelist = parser.parse(('endcomment',))"` in the above example, `nodelist` is a list of all nodes between the `{% comment %}` and `{% endcomment %}`, not counting `{% comment %}` and `{% endcomment %}` themselves.

After `parser.parse()` is called, the parser hasn't yet "consumed" the

`{% endcomment %}` tag, so the code needs to explicitly call `parser.delete_first_token()`. `CommentNode.render()` simply returns an empty string. Anything between `{% comment %}` and `{% endcomment %}` is ignored.

Parsing until another block tag, and saving contents

In the previous example, `do_comment()` discarded

everything between

`{% comment %}` and `{% endcomment %}`. Instead of doing that, it's possible to do something with the code between block tags. For example, here's a custom template tag, `{% upper %}`, that capitalizes everything between itself and

`{% endupper %}`. Usage:

```
{% upper %}This will appear in uppercase,  
{{ your_name }}.{% endupper %}
```

As in the previous example, we'll use `parser.parse()`. But this time, we pass the resulting `nodelist` to the `Node`:

```
def do_upper(parser, token):  
    nodelist = parser.parse(('endupper',))  
    parser.delete_first_token()  
    return UpperNode(nodelist)  
  
class UpperNode(template.Node):  
    def __init__(self, nodelist):  
        self.nodelist = nodelist  
    def render(self, context):  
        output =  
        self.nodelist.render(context)  
        return output.upper()
```

The only new concept here is the `self.nodelist.render(context)` in `UpperNode.render()`. For more examples of complex rendering, see the source code of `{% for %}` in

`django/template/defaulttags.py` and `{% if %}`
in `django/template/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, *Models*](#), 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, *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 '%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, *Models*](#), 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 ForeignKey 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, *Models*), 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, *Models*, 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—that is, 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
```

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.

With this manager in place, we can now do this:

```
>>> Book.objects.title_count('django')
4
>>> Book.objects.title_count('python')
18
```

Obviously, this is just an example—if you typed this in at your interactive prompt, you will likely get different return values.

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 getfull_name(self):
    # Returns the person's full name.
    return '%s %s' % (self.first_name,
self.last_name)
full_name = property(getfull_name)
```

The model instance reference in [Appendix A, Model Definition Reference](#), 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.

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.

NOTE

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 the preceding 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.

NOTE

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.

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 preceding query could also be written:

```
>>> name_map = {'first': 'first_name',
    'last': 'last_name', 'bd': 'birth_date',
    'pk': 'id'}
>>> Person.objects.raw('SELECT * FROM
    some_other_table', translations=name_map)
```

Index lookups

`raw()` supports indexing, so if you need only the first result you can write:

```
>>> first_person =
Person.objects.raw('SELECT * FROM
myapp_person')[0]
```

However, the indexing and slicing are not performed at the database level. If you have a large number of `Person` objects in your database, it is more efficient to limit the query at the SQL level:

```
>>> first_person =
Person.objects.raw('SELECT * FROM
myapp_person LIMIT 1')[0]
```

Deferring model fields

Fields may also be left out:

```
>>> people = Person.objects.raw('SELECT
id, first_name FROM myapp_person')
```

The `Person` objects returned by this query will be deferred model instances (see `defer()`). This means that the fields that are omitted from the query will be loaded on demand. For example:

```
>>> for p in Person.objects.raw('SELECT
id, first_name FROM myapp_person'):
...     print(p.first_name, # This will be
```

```
retrieved by the original query
...           p.last_name) # This will be
retrieved on demand
...
John Smith
Jane Jones
```

From outward appearances, this looks like the query has retrieved both the first name and last name. However, this example actually issued 3 queries. Only the first names were retrieved by the raw() query—the last names were both retrieved on demand when they were printed.

There is only one field that you can't leave out—the primary key field. Django uses the primary key to identify model instances, so it must always be included in a raw query. An [InvalidQuery](#) exception will be raised if you forget to include the primary key.

Adding annotations

You can also execute queries containing fields that aren't defined on the model. For example, we could use PostgreSQL's `age()` function to get a list of people with their ages calculated by the database:

```
>>> people = Person.objects.raw('SELECT *,
age(birth_date) AS age FROM myapp_person')
>>> for p in people:
...     print("%s is %s." % (p.first_name,
p.age))
John is 37.
Jane is 42.
...
```

Passing parameters into raw()

If you need to perform parameterized queries, you can pass 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

Do not use string formatting on raw queries!

It's tempting to write the preceding 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 (for more information visit, <https://www.python.org/dev/peps/pep-0249>), 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—that is, 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(*)
        """)
        results = [
            {'id': r[0], 'question': r[1], 'poll_date': r[2], 'num_responses': r[3]}
            for r in cursor.fetchall()
        ]
        return results
```

```

        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 websites that follow common pattern

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 detailed description of the other class-based views Django provides, see [Appendix C, Generic View Reference](#).

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.

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, Generic View Reference](#), documents all the generic views and their options in detail; the rest of this document will consider some of the common ways you might customize and extend generic views.

Making "friendly" template contexts

You might have noticed that our sample publisher list template stores all the publishers in a variable named `object_list`. While this works just fine, it isn't all that "friendly" to template authors: they have to "just know" that they're dealing with publishers here.

In Django, if you're dealing with a model object, this is already done for you. When you are dealing with an object or queryset, Django populates 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, that is `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 co-workers 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 [`booksacme/`](#), 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.publish
er)
```

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<id>\d+)/$', AuthorDetailView.as_view())
]
```

```
    url(r'^authors/(?P<pk>[0-9]+)/$',  
        AuthorDetailView.as_view(), name='author-  
        detail'),  
]
```

Then we'd write our new view-`get_object` is the method that retrieves the object-so we simply override it and wrap the call:

```
from django.views.generic import  
DetailView  
from django.utils import timezone  
from books.models import Author  
  
class AuthorDetailView(DetailView):  
  
    queryset = Author.objects.all()  
  
    def get_object(self):  
        # Call the superclass  
        object = super(AuthorDetailView,  
self).get_object()  
  
        # Record the last accessed date  
        object.last_accessed =  
timezone.now()  
        object.save()  
        # Return the object  
        return object
```

The URLconf here uses the named group `pk`-this name is the default name that `DetailView` uses to find the value of the primary key used to filter the queryset.

If you want to call the group something else, you can set `pk_url_kwarg` on the view. More details can be found in the reference for `DetailView`.

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 apply pretty closely to any generic view. [Appendix C, Generic View Reference](#), 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 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 authentication 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 for managing user authentication and authorization.
- 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 and so on. Only one class of user exists in Django's authentication framework, that is, `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 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.

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, let's 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()
```

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` 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.

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's 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 for all new models each time you run `manage.py migrate`.

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_ba
r')
    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

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
```

```
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:
        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.
```

NOTE

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

To log out a user who has been logged in via `login()`, use `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 `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

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_re
direct_field')
def mv_view(request):
```

```
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 HttpResponse("You can't
                            leave a review for this book.")
        # ...
```

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

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)` (that is `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.

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 a **Hash Message Authentication Code (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 of 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:
        ...
```

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 following template context is documented for each view.

There are different methods to implement these views in your project, however, the easiest and most common way is to include the provided URLconf in `Django.contrib.auth.urls` in your own URLconf, for example:

```
urlpatterns = [url('^',
    include('Django.contrib.auth.urls'))]
```

This will make each of the views available at a default URL (detailed in next section).

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

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 `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.

Template Context

- `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.

Logout

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/logged_out.html` if no argument is supplied.
- `redirect_field_name`: The name of a `GET` field containing the URL to redirect to after log out. Defaults to `next`. Overrides the `next_page` URL if the given `GET` parameter is passed.
- `current_app`: A hint indicating which application contains the current view. See the namespaced URL resolution strategy for more information.
- `extra_context`: A dictionary of context data that will be added to the default context data passed to the template.

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

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

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` from the preceding list).

Password_change_done

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

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 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`) 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

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

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`) 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

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 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` (*Table 11-1*).

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.

Form Name	Description
<code>AdminPasswordChangeForm</code>	A form used in the admin interface to change a user's password. Takes the <code>user</code> as the first positional argument.
<code>AuthenticationForm</code>	A form for logging a user in. Takes <code>request</code> as its first positional argument, which is stored on the form instance for

<code>ActionForm</code>	use by subclasses.
<code>PasswordChangeForm</code>	A form for allowing a user to change their password.
<code>PasswordResetForm</code>	A form for generating and emailing a one-time use link to reset a user's password.
<code>SetPasswordForm</code>	A form that lets a user change their password without entering the old password.
<code>UserChangeForm</code>	A form used in the admin interface to change a user's information and permissions.
<code>UserCreationForm</code>	A form for creating a new user.

Table 11.1: Django's built-in authentication forms

Authenticating 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.PermsWrapper`, 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. If you click this link, you should see the user management screen (*Figure 11.1*).



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*).

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 (that is, 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.

The screenshot shows a web browser window with the Django admin interface. The title bar says 'Add user | Django site ad...'. The address bar shows the URL '127.0.0.1:8000/admin/auth/user/add/'. The main content area is titled 'Django administration' and 'Welcome, Nigel. View site / Change password / Log out'. Below that, it shows the breadcrumb navigation: 'Home > Authentication and Authorization > Users > Add user'. The form itself is titled 'Add user' and contains instructions: 'First, enter a username and password. Then, you'll be able to edit more user options.' It has three input fields: 'Username' (with placeholder '_'), 'Password', and 'Password confirmation'. Below the password confirmation field is a note: 'Enter the same password as above, for verification.' At the bottom right are three buttons: 'Save and add another', 'Save and continue editing', and a larger 'Save' button.

Figure 11.2: Django admin add user screen

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 (*Figure 11.3*).



Figure 11.3: Link to change password (circled)

Once you click the link, you will be taken to the change password form (*Figure 11.4*).



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** (for more information visit <http://en.wikipedia.org/wiki/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 the following code). 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(for more information visit <http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf>). 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, and so on. The next few sections describe a couple of common ways advanced users may want to modify this setting.

Using Bcrypt with Django

Bcrypt (for more information visit <http://en.wikipedia.org/wiki/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 `brypt` library. This can be done by running `pip install Django[brypt]`, 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.BCryptSHA256P  
asswordHasher',  
  
'Django.contrib.auth.hashers.BCryptPasswor  
dHasher',  
  
'Django.contrib.auth.hashers.PBKDF2Passwor  
dHasher',  
  
'Django.contrib.auth.hashers.PBKDF2SHA1Pas  
swordHasher',  
  
'Django.contrib.auth.hashers.SHA1PasswordH  
asher',  
  
'Django.contrib.auth.hashers.MD5PasswordHa  
sher',  
  
'Django.contrib.auth.hashers.CryptPassword  
Hasher',  
]
```

(You need to keep the other entries in this list, or else Django won't be able to upgrade passwords; see the following section).

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)`.

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.

The Django development team have 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(PBKDF2PasswordH  
asher):  
    iterations =  
    PBKDF2PasswordHasher.iterations * 100
```

2. Save this somewhere in your project. For example, you might put this in a file like `myproject/hashers.py`.
3. Add your new hasher as the first entry in `PASSWORD_HASHERS`:

```
PASSWORD_HASHERS = [  
  
'myproject.hashers.MyPBKDF2PasswordHasher'  
,  
  
'Django.contrib.auth.hashers.PBKDF2Passwor  
dHasher',  
  
# ... #  
]
```

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](#).

If you'd like to manually authenticate a user by comparing a plain-text password to the hashed password in the database, use the function `check_password()`. It takes two arguments: the plain-text password to check, and the full value of a user's `password` field in the database to check against, and returns `True` if they match, `False` otherwise.

`make_password()` 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 (one that will be never accepted by `check_password()`).

`is_password_usable()` 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.

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 `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'
```

d']

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 authorization backend, or use the mechanisms provided by most web servers. The order of `AUTHENTICATION_BACKENDS` matters, so if the same username and password is valid in multiple backends, Django will stop processing at the first positive match. If a backend raises a `PermissionDenied` exception, authentication will immediately fail. Django won't check the backends that follow.

Once a user has authenticated, Django stores which backend was used to authenticate the user in the user's session, and re-uses the same backend for the duration of that session whenever access to the currently authenticated user is needed. This effectively means that authentication sources are cached on a per-session basis, so if you change `AUTHENTICATION_BACKENDS`, you'll need to clear out session data if you need to force users to re-authenticate using different methods. A simple way to do that is simply to execute `Session.objects.all().delete()`.

Writing an authentication backend

An authentication backend is a class that implements two required methods: `get_user(user_id)` and `authenticate(**credentials)`, as well as a set of optional permission related authorization methods. The `get_user` method takes a `user_id`-which could be a username, database ID or whatever, but has to be the primary key of your `User` object-and returns a `User` object. The `authenticate` method takes credentials as keyword arguments. Most of the time, it'll just look like this:

```
class MyBackend(object):
    def authenticate(self, username=None,
password=None):
        # Check the username/password and
return a User.
    ...
```

But it could also authenticate a token, like so:

```
class MyBackend(object):
    def authenticate(self, token=None):
        # Check the token and return a
User.
    ...
```

Either way, `authenticate` should check the credentials it gets, and it should return a `User` object that matches those credentials, if the credentials are

valid. If they're not valid, it should return `None`. The Django admin system is tightly coupled to the Django `User` object described at the beginning of this chapter.

For now, the best way to deal with this is to create a Django `User` object for each user that exists for your backend (for example, in your LDAP directory, your external SQL database, and so on.) 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$4e987$afbcf42e21bd417fb71db8c66b321e
9fc33051de'
    """
```

```
        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='password')
                    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 authorization 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 mentioned before could implement permissions for the 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 preceding example. Notice that in addition to the same arguments given to the associated `User` functions, the backend authorization 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 it 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 that is they have provided no valid authentication details. However, that does not necessarily mean they are not authorized to do anything. At the most basic level, most websites authorize anonymous users to browse most of the site, and many allow anonymous posting of comments and so on.

Django's permission framework does not have a place to store permissions for anonymous users. However, the user object passed to an authentication backend may be an [Django.contrib.auth.models.AnonymousUser](#) object, allowing the backend to specify custom authorization behavior for anonymous users.

This is especially useful for the authors of re-usable apps, who can delegate all questions of authorization to the auth backend, rather than needing settings, for example, to control anonymous access.

Authorization for inactive users

An inactive user is a one that 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, that is, 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 preceding 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.

NOTE

Changing `AUTH_USER_MODEL` has a big effect on your Django project, particularly your database structure. For example, if you change your `AUTH_USER_MODEL` after you have run migrations, you will have to manually update your database because it affects the construction of many database table relationships. Unless there is a very good reason to do so, you should not change your `AUTH_USER_MODEL`.

Notwithstanding the preceding warning, Django does fully support custom user models, however a full

explanation is beyond the scope of this book. A full example of an admin-compliant custom user app, as well as comprehensive documentation on custom user models can be found on the Django Project website (<https://docs.djangoproject.com/en/1.8/topics/auth/customizing/>).

What's next?

In this chapter, we have learned about user authentication in Django, the built-in authentication tools, as well as the wide range of customizations available. In the next chapter, we will be covering arguably the most important tool for creating and maintaining robust applications—automated testing.

Chapter 12. Testing in Django

Introduction to testing

Like all mature programming languages, Django provides inbuilt *unit testing* capabilities. Unit testing is a software testing process where individual units of a software application are tested to ensure they do what they are expected to do.

Unit testing can be performed at multiple levels-from testing an individual method to see if it returns the right value and how it handles invalid data, up to testing a whole suite of methods to ensure a sequence of user inputs leads to the desired results.

Unit testing is based on four fundamental concepts:

1. A **test fixture** is the setup needed to perform tests. This could include databases, sample datasets and server setup. A test fixture may also include any clean-up actions required after tests have been performed.
2. A **test case** is the basic unit of testing. A test case checks whether a given set of inputs leads to an expected set of results.
3. A **test suite** is a number of test cases, or other test suites, that are executed as a group.
4. A **test runner** is the software program that controls the execution of tests and feeds the results of tests back to the user.

Software testing is a deep and detailed subject and this

chapter should be considered to be only a bare introduction to unit testing. There are a large number of resources on the Internet on software testing theory and methods and I encourage you to do your own research on this important topic. For a more detailed discussion on Django's approach to unit testing, see the Django Project website.

Introducing automated testing

What are automated tests?

You have been testing code right throughout this book; maybe without even realizing it. Each time you use the Django shell to see if a function works, or to see what output you get for a given input, you are testing your code. For example, back in [Chapter 2, Views and URLconfs](#), we passed a string to a view that expected an integer to generate a [TypeError](#) exception.

Testing is a normal part of application development, however 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.

So why create tests?

If creating simple applications like those in this book is the last bit of Django programming you do, then true, you don't need to know how to create automated tests. But, if you wish to become a professional programmer and/or work on more complex projects, you need to know how

to create automated tests.

Creating automated tests will:

- **Save you time:** Manually testing the myriad complex interactions between components of a big application is time-consuming and error prone. Automated tests save time and let you focus on programming.
- **Prevent problems:** Tests highlight the internal workings of your code, so you can see where things have gone wrong.
- **Look professional:** The pros write tests. Jacob Kaplan-Moss, one of Django's original developers, says "Code without tests is broken by design."
- **Improve teamwork:** Tests guarantee that colleagues don't inadvertently break your code (and that you don't break theirs without knowing).

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.

Writing a test

To create your first test, let's introduce a bug into your Book model.

Say you have decided to create a custom method on your Book model to indicate whether the book has been published recently. Your Book model may look something like this:

```
import datetime
from django.utils import timezone

from django.db import models

# ... #

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

    def recent_publication(self):
        return self.publication_date >=
            timezone.now().date() -
            datetime.timedelta(weeks=8)

# ... #
```

First we have imported two new modules: Python's `datetime` and `timezone` from `django.utils`. We need these modules to be able to do calculations with dates. Then we have added a custom method to the `Book` model called `recent_publication` that works out what date it was eight weeks ago and returns true if the publication date of the book is more recent.

So let's jump to the interactive shell and test our new method:

```
python manage.py shell

>>> from books.models import Book
>>> import datetime
>>> from django.utils import timezone
>>> book = Book.objects.get(id=1)
>>> book.title
'Mastering Django: Core'
>>> book.publication_date
datetime.date(2016, 5, 1)
>>> book.publication_date >=
timezone.now().date() -
datetime.timedelta(weeks=8)
True
```

So far so good, we have imported our book model and retrieved a book. Today is the 11th June, 2016 and I have entered the publication date of my book in the database as the 1st of May, which is less than eight weeks ago, so the function correctly returns `True`.

Obviously, you will have to modify the publication date in your data so this exercise still works for you based on

when you complete this exercise.

Now let's see what happens if we set the publication date to a time in the future to, say, 1st September:

```
>>> book.publication_date  
datetime.date(2016, 9, 1)  
>>>book.publication_date >=  
timezone.now().date()-  
datetime.timedelta(weeks=8)  
True
```

Oops! Something is clearly wrong here. You should be able to quickly see the error in the logic-any date after eight weeks ago is going to return true, including dates in the future.

So, ignoring the fact that this is a rather contrived example, lets now create a test that exposes our faulty logic.

Creating a test

When you created your books app with Django's `startapp` command, it created a file called `tests.py` in your app directory. This is where any tests for the books app should go. So let's get right to it and write a test:

```
import datetime from django.utils import timezone  
from django.test import TestCase from .models import Book  
class BookMethodTests(TestCase):  
    def test_recent_pub(self):  
        """ recent_publication() should return False for future publication dates.  
        """  
        futuredate = timezone.now().date() + datetime.timedelta(days=5)  
        future_pub = Book(publication_date=futuredate)  
        self.assertEqual(future_pub.recent_publication(), False)
```

This should all be pretty straight forward as it's nearly exactly what we did in the Django shell, the only real difference is that we now have encapsulated our test code in a class and created an assertion that tests our `recent_publication()` method against a future date.

We will be covering test classes and the `assertEqual` method in greater detail later in the chapter-for now, we just want to look at how tests work at a very basic level before getting onto more complicated topics.

Running tests

Now we have created our test, we need to run it. Fortunately, this is very easy to do, jump into your terminal and type: python manage.py test books

After a moment, Django should print out something like this:

```
Creating test database for alias
'default'...
F
=====
=====
FAIL: test_recent_pub
(books.tests.BookMethodTests)
-----
-----
Traceback (most recent call last):
  File "C:\Users\Nigel\ ...
mysite\books\tests.py", line 25, in
test_recent_pub

    self.assertEqual(future_pub.recent_publica
tion(), False)
AssertionError: True != False

-----
-----
Ran 1 test in 0.000s

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

What happened is this:

- Python `manage.py test books` looked for tests in the books application
- It found a subclass of the `django.test.TestCase` class
- It created a special database for the purpose of testing
- It looked for methods with names beginning with "test"
- In `test_recent_pub` it created a Book instance whose `publication_date` field is 5 days in the future; and
- Using the `assertEqual()` method, it discovered that its `recent_publication()` returns `True`, when it was supposed to return `False`.

The test informs us which test failed and even the line on which the failure occurred. Note also that if you are on a *nix system or a Mac, the file path will be different.

That's it for a very basic introduction to testing in Django. As I said at the beginning of the chapter, testing is a deep and detailed subject that is highly important to your career as a programmer. I can't possibly cover all the facets of testing in a single chapter, so I encourage you to dig deeper into some of the resources mentioned in this chapter as well as the Django documentation.

For the remainder of the chapter, I will be going over the various testing tools Django puts at your disposal.

Testing tools

Django provides a 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 (for more information visit <http://seleniumhq.org/>) 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.

For a more detailed look at the Django test client with examples, see the [Django Project website](#).

Provided TestCase classes

Normal Python unit test classes extend a base class of `unittest.TestCase`. Django provides a few extensions of this base class:

SIMPLE TESTCASE

Extends `unittest.TestCase` with some basic functionality like:

- Saving and restoring the Python warning machinery state.
- Adding a number of useful assertions including:
 - 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 test [Client](#).
- Custom test-time URL maps.

TRANSACTION TESTCASE

Django's [TestCase](#) class (described in following paragraph) 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.

In those cases, you should use [TransactionTestCase](#). [TransactionTestCase](#) and [TestCase](#) are identical except for the manner in which the database is reset to a known state and the ability for test code to test the effects of commit and rollback:

- A [TransactionTestCase](#) resets the database after the test runs by truncating all tables. A [TransactionTestCase](#) may call commit and rollback and observe the effects of these calls on the database.
- A [TestCase](#), on the other hand, does not truncate tables after a test. Instead, it encloses the test code in a database transaction that is rolled back at the end of the test. This guarantees that the rollback at the end of the test restores the database to its initial state.

[TransactionTestCase](#) inherits from [SimpleTestCase](#).

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.

`TestCase` inherits from `TransactionTestCase`.

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.

Test cases features

DEFAULT TEST CLIENT

Every test case in a `*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('customerdetails/')

self.assertEqual(response.status_code,
200)

    def test_index(self):
        client = Client()
        response =
client.get('customerindex/')

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('customerdetails/')

self.assertEqual(response.status_code,
```

```
200)

    def test_index(self):
        response =
self.client.get('customerindex/')

    self.assertEqual(response.status_code,
200)
```

FIXTURE LOADING

A test case for a database-backed website 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 [TestCase](#) 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']\n\n    def setUp(self):\n        # Test definitions as before.\n        call_setup_methods()\n\n    def testFluffyAnimals(self):\n        # A test that uses the fixtures.\n        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.

OVERRIDING SETTINGS

NOTE

Use the functions 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.

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 at <https://www.python.org/dev/peps/pep-0343>) 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,
        'accountslogin/?next=sekrit')

        # Then override the LOGIN_URL
        setting
        with
        self.settings(LOGIN_URL='otherlogin/'):
            response =
            self.client.get('sekrit')
            self.assertRedirects(response,
            'otherlogin/?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.

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',
                'django.middleware.cache.UpdateCacheMiddleware',
            ],
            'prepend': [
                '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 at <https://www.python.org/dev/peps/pep-0318>). It's used like this:

```
from django.test import TestCase,  
override_settings  
  
class LoginTestCase(TestCase):  
  
    @override_settings(LOGIN_URL='otherlogin/'  
    )  
    def test_login(self):  
        response =  
self.client.get('sekrit')  
        self.assertRedirects(response,  
'otherlogin/?next=sekrit')
```

The decorator can also be applied to `TestCase` classes:

```
from django.test import TestCase,  
override_settings  
  
@override_settings(LOGIN_URL='otherlogin/'  
)  
class LoginTestCase(TestCase):  
  
    def test_login(self):  
        response =  
self.client.get('sekrit')  
        self.assertRedirects(response,  
'otherlogin/?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('/')
        # ...
```

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 call-backs to clean up and otherwise reset state when settings are changed.

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:

- `assertRaisesMessage`: Asserts that execution of the callable object raised an exception with an `expected_message` representation.
- `assertFieldOutput`: Asserts that a form field behaves correctly with various inputs.
- `assertFormError`: Asserts that a field on a form raises the provided list of errors when rendered on the form.
- `assertFormsetError`: Asserts that the `formset` raises the provided list of errors when rendered.
- `assertContains`: Asserts that a `Response` instance produced the given `status_code` and that `text` appears in the content of the response.
- `assertNotContains`: Asserts that a `Response` instance produced the given `status_code` and that `text` does not appear in the content of the response.
- `assertTemplateUsed`: Asserts that the template with the given name was used in rendering the response. The name is a string such

as 'admin/index.html'.

- [assertTemplateNotUsed](#): Asserts that the template with the given name was not used in rendering the response.
- [assertRedirects](#): 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`.
 - [assertHTMLEqual](#): 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, for example, 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).
- [assertHTMLNotEqual](#): Asserts that the strings `html1` and `html2` are *not* equal. The comparison is based on HTML semantics. See [assertHTMLEqual\(\)](#) for details.
- [assertXMLEqual](#): 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.
- [assertXMLNotEqual](#): Asserts that the strings `xml1` and `xml2` are *not* equal. The comparison is based on XML semantics. See [assertXMLEqual\(\)](#) for details.

- `assertInHTML`: Asserts that the HTML fragment `needle` is contained in the `haystack` one.
- `assertJSONEqual`: Asserts that the JSON fragments `raw` and `expected_data` are equal.
- `assertJSONNotEqual`: Asserts that the JSON fragments `raw` and `expected_data` are not equal.
- `assertQuerysetEqual`: 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.
- `assertNumQueries`: Asserts that when `func` is called with `*args` and `**kwargs` that `num` database queries are executed.

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.)

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 following code 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.

The test database

Tests that require a database (namely, model tests) will not use your 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 (that is, the database will be created in memory, bypassing the filesystem entirely!).

If you want to use a different database name, specify `NAME` in the `TEST` dictionary for any given database in `DATABASES`. On PostgreSQL, `USER` will also need read access to the built-in `postgres` database. Aside from using a separate database, the test runner will otherwise use all of the same database settings you have in your settings file: `ENGINE`, `USER`, `HOST`, and so on. 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.

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 of the following file the in 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's 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.

See the Django Project website for more information on using different testing frameworks.

What's next?

Now that you know how to write tests for your Django projects, we will be moving on to a very important topic once you are ready to turn your project into a real live website-deploying Django to a web server.

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. Following checks described 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('etcsecret_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, *Introduction to Django and Getting Started*](#), 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, Templates](#)) 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 attack.

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 attempts to interpret them. For instance, if a user uploads a `.php` file, the web server shouldn't execute it. Now is a good time to check your backup strategy for these files.

HTTPS

Any website which allows users to log in should enforce site-wide HTTPS to avoid transmitting access tokens in clear. In Django, access tokens include the login/password, the session cookie, and password reset tokens. (You can't do much to protect password reset tokens if you're sending them by email.) Protecting sensitive areas such as the user account or the admin isn't sufficient, because the same session cookie is used for HTTP and HTTPS. Your web server must redirect all HTTP traffic to HTTPS, and only transmit HTTPS requests to Django. Once you've set up HTTPS, enable the following settings.

CSRF_COOKIE_SECURE

Set this to `True` to avoid transmitting the CSRF cookie over HTTP accidentally.

SESSION_COOKIE_SECURE

Set this to `True` to avoid transmitting the session cookie over HTTP accidentally.

Performance optimizations

Setting `DEBUG = False` disables several features that are only useful in development. In addition, you can tune the following settings.

CONN_MAX_AGE

Enabling persistent database connections can result in a nice speed-up when connecting to the database accounts for a significant part of the request processing time. This helps a lot on virtualized hosts with limited network performance.

TEMPLATES

Enabling the cached template loader often improves performance drastically, as it avoids compiling each template every time it needs to be rendered. See the template loaders docs for more information.

Error reporting

By the time you push your code to production, it's hopefully robust, but you can't rule out unexpected errors. Thankfully, Django can capture errors and notify you accordingly.

LOGGING

Review your logging configuration before putting your website in production, and check that it works as expected as soon as you have received some traffic.

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 (for more information visit

<http://sentry.readthedocs.org/en/latest/>) 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

(<https://docs.djangoproject.com/en/1.8/topics/http/views/#customizing-error-views>) 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 (for more information visit

<http://www.virtualenv.org/>), you'll need to add the path to this virtualenv's `site-packages` directory to your

Python path as well. To do this, add an additional path to your `WSGIPythonPath` directive, with multiple paths separated by a colon (`:`) if using a UNIX-like system, or a semicolon (`;`) if using Windows. If any part of a directory path contains a space character, the complete argument string to `WSGIPythonPath` must be quoted:

`WSGIPythonPath`

`path/to/mysite.com:path/to/your/venv/lib/python3.X/site-packages`

Make sure you give the correct path to your virtualenv, and replace `python3.X` with the correct Python version (for example `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 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`, and so on. 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 = False
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 = True
    TEMPLATE_DEBUG = True
else:
    DEBUG = False
    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, and so on. 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.

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 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` 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 (for example, '`mysite.settings`').

Deploying Django to a production server

NOTE

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 (<http://httpd.apache.org/>) and [mod_wsgi](http://code.google.com/p/modwsgi) (<http://code.google.com/p/modwsgi>) 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. Note, 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](#) preceding to 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 the following file—on your system, usually inside of your project package (`mysite` in this example). This tells Apache to serve any request following 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.

NOTE

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 `WSGIDaemonProcess` directives.

A further change required to the preceding 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—that is, one that's not also running Django—for serving media. Here are some good choices:

- Nginx (for more information visit <http://code.google.com/p/modwsgi>)
- 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` preceding to 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 recommend using `django.contrib.staticfiles` to handle the admin files (along strongly 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 in the preceding paragraph,

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.

Following 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`.

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(<http://fabfile.org/>).

Following, and in the following sections, we'll show off a few example **fabfiles** (that is 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 = 'homewww/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—that is 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:

1. When your static files change, run `collectstatic` locally.
2. Push your local `STATIC_ROOT` up to the static file server into the directory that's being served. `rsync` (<https://rsync.samba.org/>) 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 = 'tmpstatic'

# Where the static files should go
# remotely
env.remote_static_root =
```

```
'homewww/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 preceding paragraph, 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*. 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 and CPU) that they'd prefer to monopolize. This is solved easily by moving the database server to a second machine.

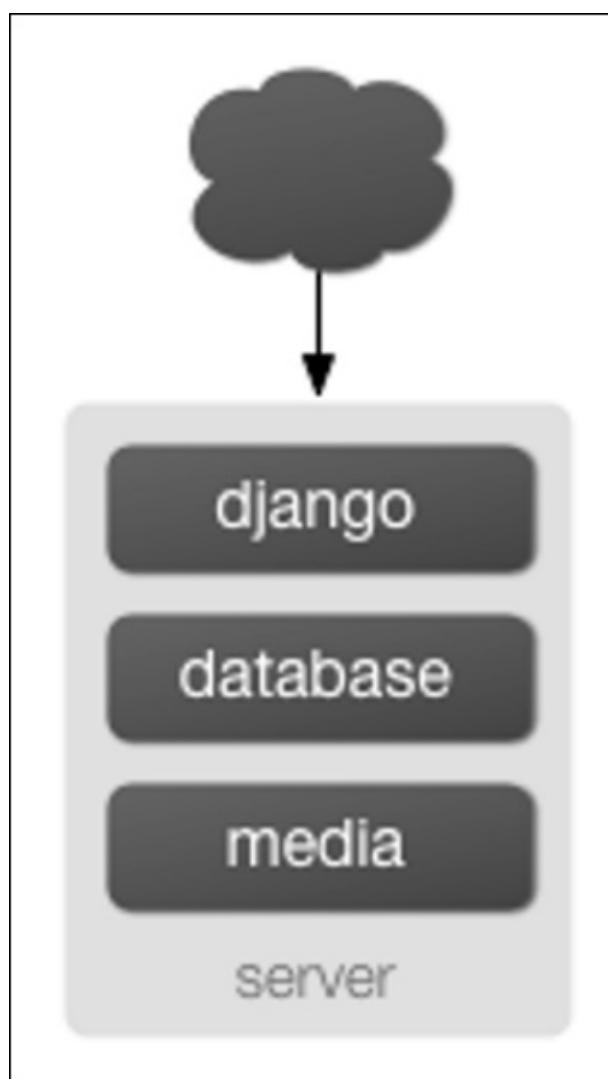


Figure 13.1: a single server Django setup.

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*.

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.

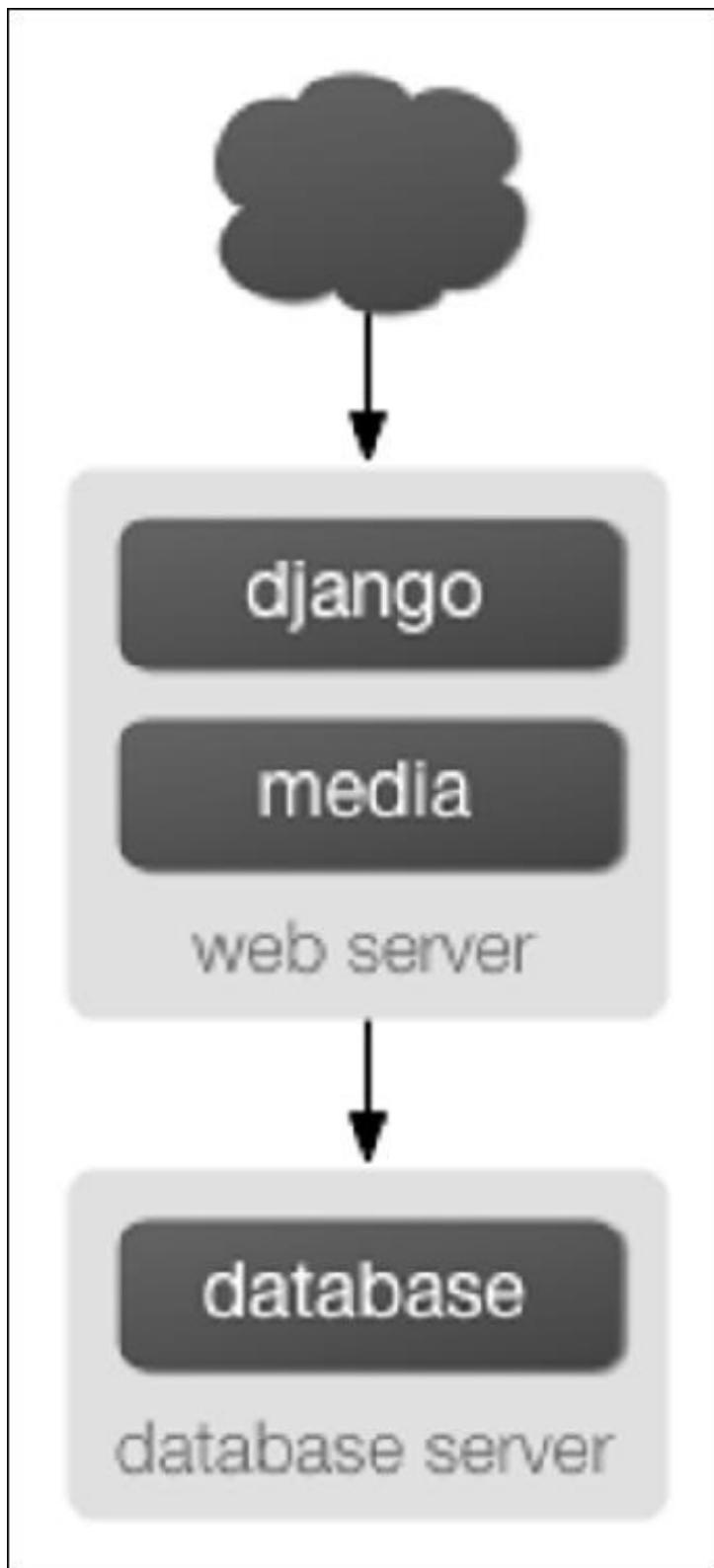


Figure 13.2: Moving the database onto a dedicated server.

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*).

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, and so on), 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.

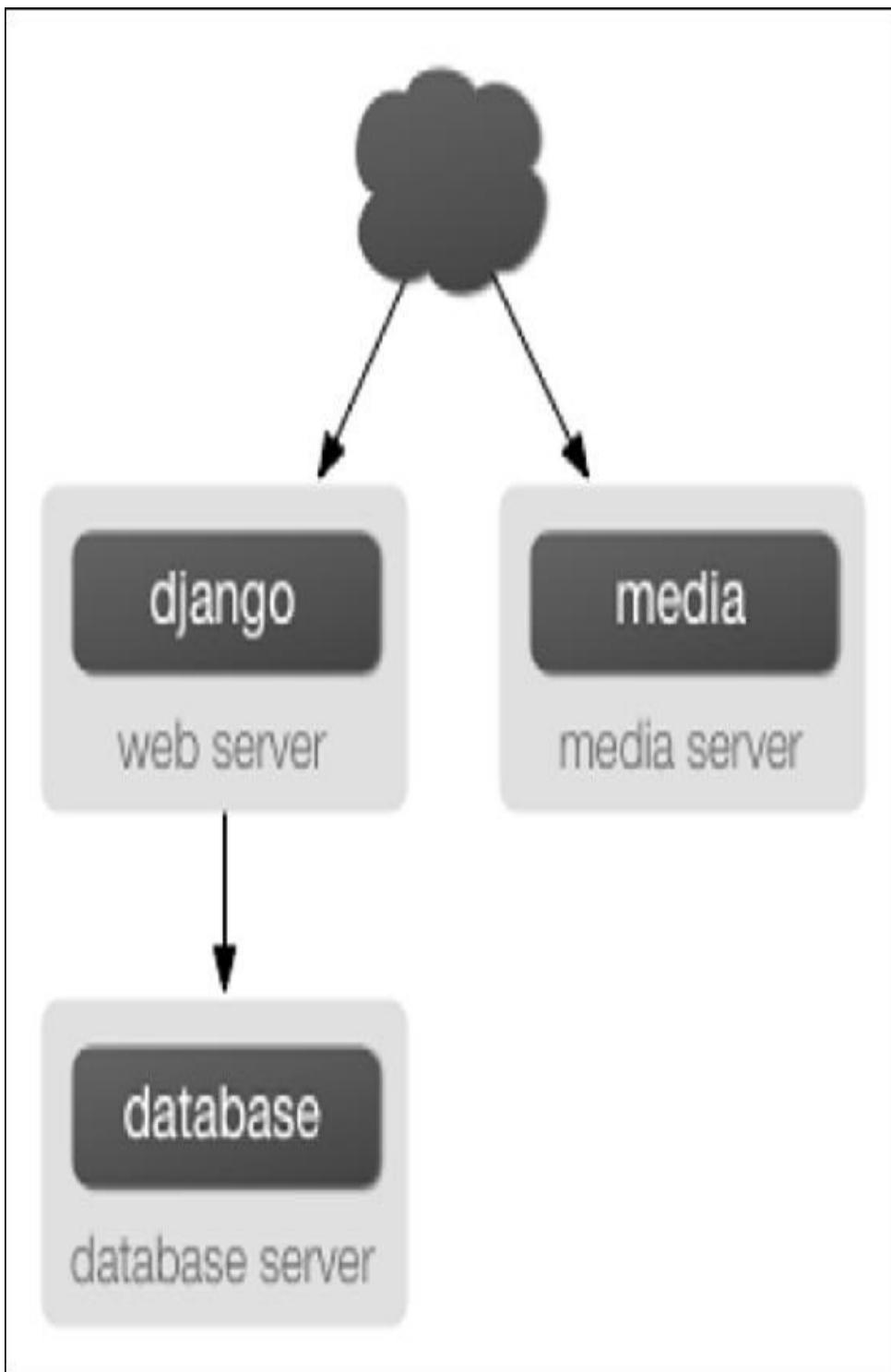


Figure 13.3: Separating out the media server.

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 all 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 16](#), *Django's Cache Framework*).

With the web servers now clustered, our evolving architecture starts to look more complex, as shown in *Figure 13.4*.

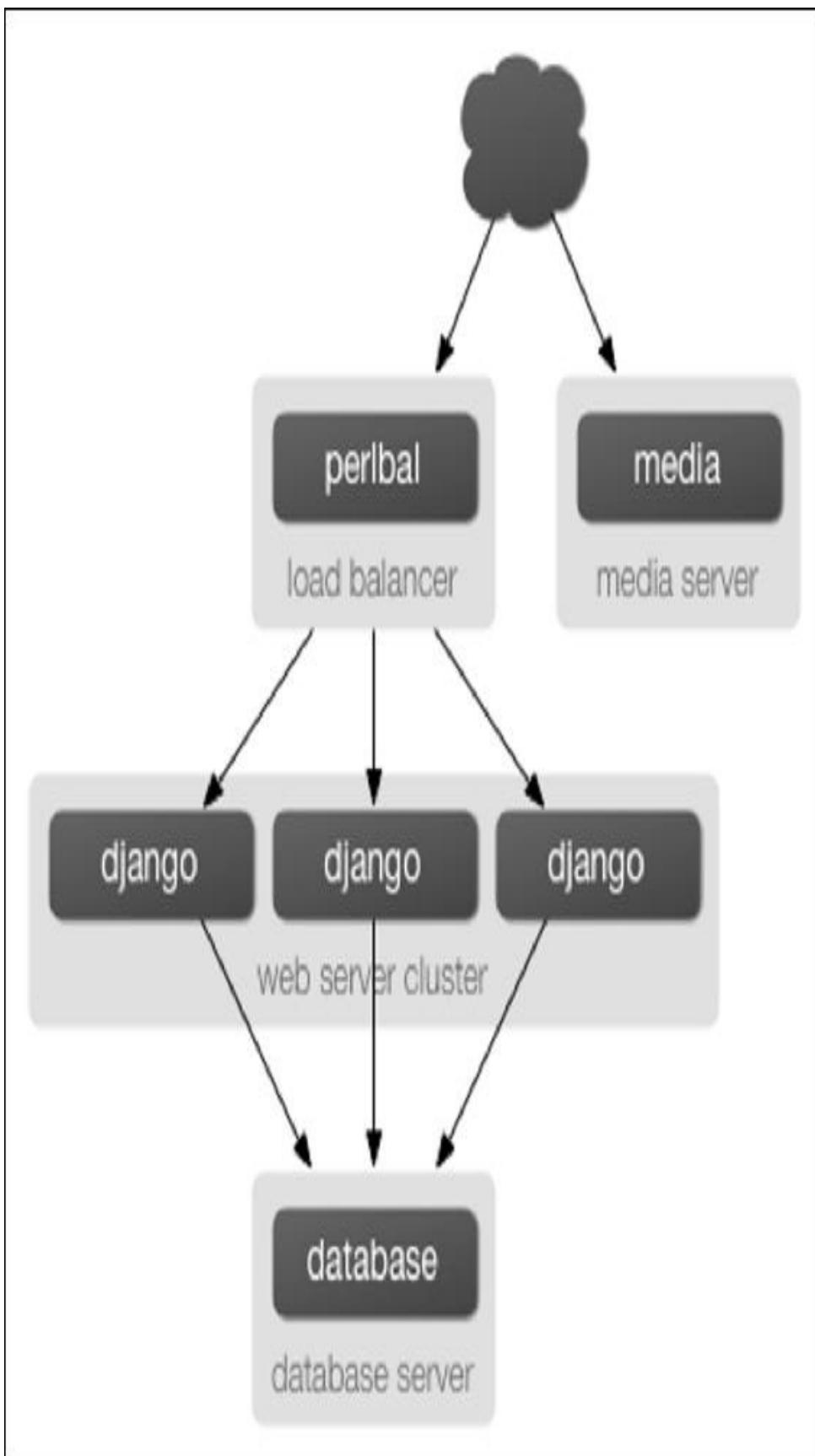


Figure 13.4: A load-balanced, redundant server setup.

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 may look like *Figure 13.5*.

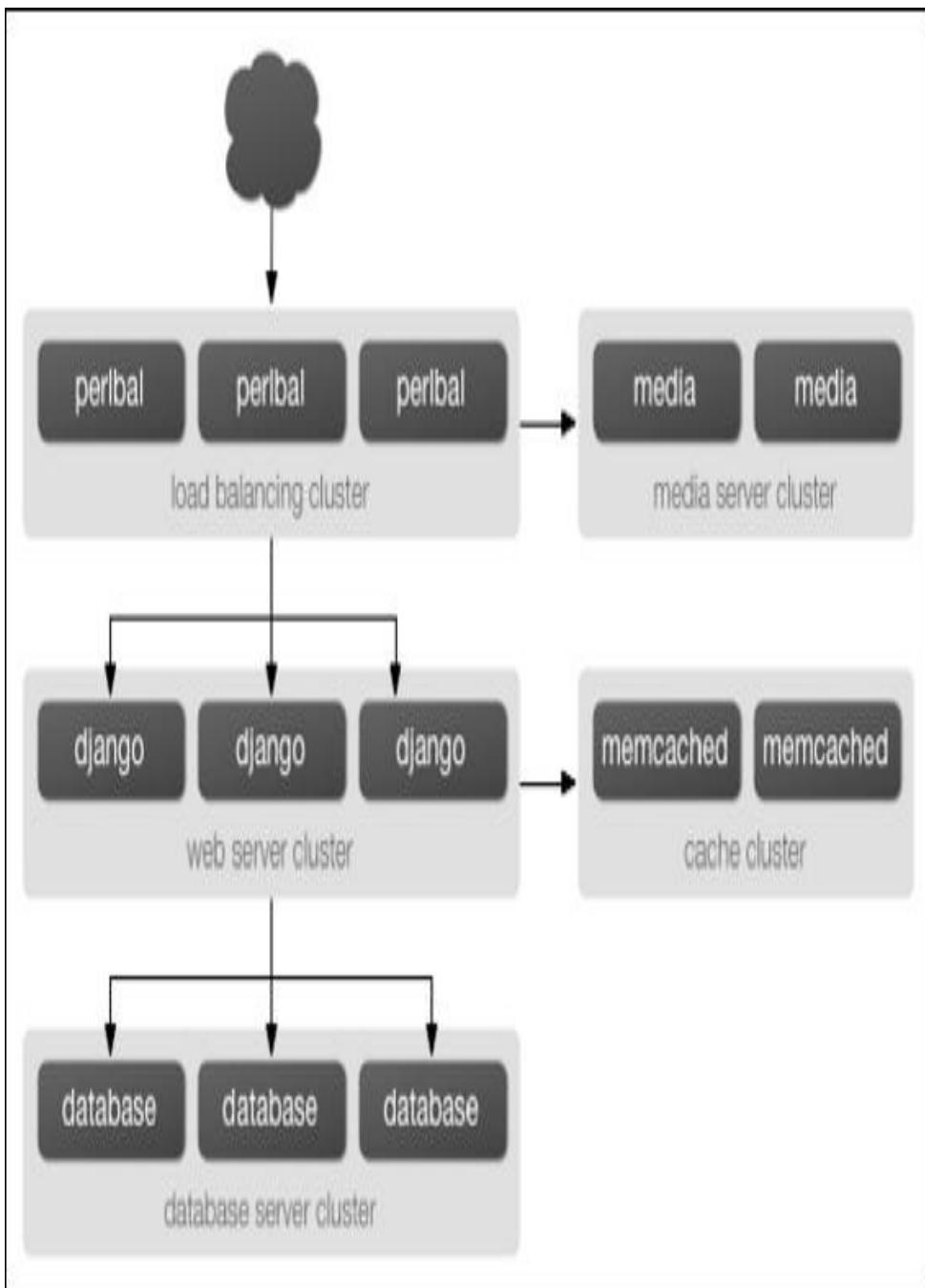


Figure 13.5: An example large-scale Django setup.

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. They 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 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 backends, 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 16, Django's Cache Framework](#), is your best friend here: learn how to use Django's cache framework, and use it everywhere possible. Aggressive, pre-emptive 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-an incredibly active, growing group of Django developers. Our community has a huge amount of collective experience to offer.

What's next?

The remaining chapters focus on other Django features that you may or may not need, depending on your application. Feel free to read them in any order you choose.

Chapter 14. Generating Non-HTML Content

Usually when we talk about developing websites, we're talking about producing HTML. Of course, there's a lot more to the web than HTML; we use the web to distribute data in all sorts of formats: RSS, PDFs, images, and so forth.

So far, we've focused on the common case of HTML production, but in this chapter we'll take a detour and look at using Django to produce other types of content. Django has convenient built-in tools that you can use to produce some common non-HTML content:

- Comma-delimited (CSV) files for importing into spreadsheet applications.
- PDF files.
- RSS/Atom syndication feeds.
- Sitemaps (an XML format originally developed by Google that gives hints to search engines).

We'll examine each of those tools a little later, but first we'll cover the basic principles.

The basics: views and MIME types

Recall from [Chapter 2, Views and URLconfs](#), that a view function is simply a Python function that takes a web request and returns a web response. This response can be the HTML contents of a web page, or a redirect, or a 404 error, or an XML document, or an image ...or anything, really. More formally, a Django view function must:

1. Accept an `HttpRequest` instance as its first argument; and
2. Return an `HttpResponse` instance.

The key to returning non-HTML content from a view lies in the `HttpResponse` class, specifically the `content_type` argument. By default, Django sets `content_type` to `text/html`. You can however, set `content_type` to any of the official Internet media types (MIME types) managed by IANA (for more information visit <http://www.iana.org/assignments/media-types/media-types.xhtml>).

By tweaking the MIME type, we can indicate to the browser that we've returned a response of a different format. For example, let's look at a view that returns a PNG image. To keep things simple, we'll just read the file off the disk:

```
from django.http import HttpResponse

def my_image(request):
    image_data =
        open("path/to/my/image.png", "rb").read()
    return HttpResponse(image_data,
                        content_type="image/png")
```

That's it! If you replace the image path in the `open()` call with a path to a real image, you can use this very simple view to serve an image, and the browser will display it correctly.

The other important thing to keep in mind is that `HttpResponse` objects implement Python's standard file-like object API. This means that you can use an `HttpResponse` instance in any place Python (or a third-party library) expects a file. For an example of how that works, let's take a look at producing CSV with Django.

Producing CSV

Python comes with a CSV library, [csv](#). The key to using it with Django is that the [csv](#) module's CSV-creation capability acts on file-like objects, and Django's [HttpResponse](#) objects are file-like objects. Here's an example:

```
import csv
from django.http import HttpResponse

def some_view(request):
    # Create the HttpResponse object with
    # the appropriate CSV header.
    response =
    HttpResponse(content_type='text/csv')
    response['Content-Disposition'] =
    'attachment;
        filename="somefilename.csv"'

    writer = csv.writer(response)
    writer.writerow(['First row', 'Foo',
    'Bar', 'Baz'])
    writer.writerow(['Second row', 'A',
    'B', 'C', '"Testing"'])

    return response
```

The code and comments should be self-explanatory, but a few things deserve a mention:

- The response gets a special MIME type, [text/csv](#). This tells browsers that the document is a CSV file, rather than an HTML file. If you leave this off, browsers will probably interpret the output as

HTML, which will result in ugly, scary gobbledegook in the browser window.

- The response gets an additional `Content-Disposition` header, which contains the name of the CSV file. This filename is arbitrary; call it whatever you want. It'll be used by browsers in the Save as... dialogue, and so on.
- Hooking into the CSV-generation API is easy: Just pass `response` as the first argument to `csv.writer`. The `csv.writer` function expects a file-like object, and `HttpResponse` objects fit the bill.
- For each row in your CSV file, call `writer.writerow`, passing it an iterable object such as a list or tuple.
- The CSV module takes care of quoting for you, so you don't have to worry about escaping strings with quotes or commas in them. Just pass `writerow()` your raw strings, and it'll do the right thing.

Streaming large CSV files

When dealing with views that generate very large responses, you might want to consider using Django's `StreamingHttpResponse` instead. For example, by streaming a file that takes a long time to generate you can avoid a load balancer dropping a connection that might have otherwise timed out while the server was generating the response. In this example, we make full use of Python generators to efficiently handle the assembly and transmission of a large CSV file:

```
import csv

from django.utils.six.moves import range
from django.http import
StreamingHttpResponse

class Echo(object):
    """An object that implements just the
```

```
    write method of the file-like
    interface.

    """
    def write(self, value):
        """Write the value by returning
        it, instead of storing in a buffer."""
        return value

    def some_streaming_csv_view(request):
        """A view that streams a large CSV
        file."""
        # Generate a sequence of rows. The
        range is based on the maximum number of
        # rows that can be handled by a single
        sheet in most spreadsheet
        # applications.
        rows = (["Row {}".format(idx),
        str(idx)] for idx in range(65536))
        pseudo_buffer = Echo()
        writer = csv.writer(pseudo_buffer)
        response =
StreamingHttpResponse((writer.writerow(row
)
        for row in rows),
content_type="text/csv")
        response['Content-Disposition'] =
'attachment;
        filename="somefilename.csv"'
        return response
```

Using the template system

Alternatively, you can use the Django template system to generate CSV. This is lower-level than using the convenient Python `csv` module, but the solution is presented here for completeness. The idea here is to pass a list of items to your template, and have the template output the commas in a `for` loop. Here's an example, which generates the same CSV file as above:

```
from django.http import HttpResponseRedirect
from django.template import loader,
Context

def some_view(request):
    # Create the HttpResponseRedirect object with
    the appropriate CSV header.
    response =
    HttpResponseRedirect(content_type='text/csv')
    response['Content-Disposition'] =
    'attachment;
        filename="somefilename.csv"'

    # The data is hard-coded here, but you
    could load it
    # from a database or some other
    source.
    csv_data = (
        ('First row', 'Foo', 'Bar',
    'Baz'),
        ('Second row', 'A', 'B', 'C',
    '"Testing"', "Here's a quote"),
    )

    t =
```

```
loader.get_template('my_template_name.txt')
)
c = Context({'data': csv_data,})
response.write(t.render(c))
return response
```

The only difference between this example and the previous example is that this one uses template loading instead of the CSV module. The rest of the code-such as the `content_type='text/csv'`-is the same. Then, create the template `my_template_name.txt`, with this template code:

```
{% for row in data %}
    "{{ row.0|addslashes }}",
    "{{ row.1|addslashes }}",
    "{{ row.2|addslashes }}",
    "{{ row.3|addslashes }}",
    "{{ row.4|addslashes }}"
{% endfor %}
```

This template is quite basic. It just iterates over the given data and displays a line of CSV for each row. It uses the `addslashes` template filter to ensure there aren't any problems with quotes.

Other text-based formats

Notice that there isn't very much specific to CSV here—just the specific output format. You can use either of these techniques to output any text-based format you can dream of. You can also use a similar technique to generate arbitrary binary data; For example, generating PDFs.

Generating PDF

Django is able to output PDF files dynamically using views. This is made possible by the excellent, opensource ReportLab (for more information visit <http://www.reportlab.comopensource/>) Python PDF library. The advantage of generating PDF files dynamically is that you can create customized PDFs for different purposes-say, for different users or different pieces of content.

Install ReportLab

The **ReportLab** library is available on PyPI. A user guide (not coincidentally, a PDF file) is also available for download. You can install ReportLab with `pip`: \$ pip install reportlab

Test your installation by importing it in the Python interactive interpreter:

```
>>> import reportlab
```

If that command doesn't raise any errors, the installation worked.

Write your view

The key to generating PDFs dynamically with Django is that the ReportLab API, like the `csv` library acts on file-like objects, like Django's `HttpResponse`. Here's a Hello World example:

```
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def some_view(request):
    # Create the HttpResponse object with
    # the appropriate PDF headers.
    response =
    HttpResponseRedirect(content_type='application/pdf')
    response['Content-Disposition'] =
    'attachment;
    filename="somefilename.pdf"'

    # Create the PDF object, using the
    # response object as its "file."
    p = canvas.Canvas(response)

    # Draw things on the PDF. Here's where
    # the PDF generation happens.
    # See the ReportLab documentation for
    # the full list of functionality.
    p.drawString(100, 100, "Hello world.")

    # Close the PDF object cleanly, and
    # we're done.
    p.showPage()
    p.save()
    return response
```

The code and comments should be self-explanatory, but a few things deserve a mention:

- The response gets a special MIME type, `application/pdf`. This tells browsers that the document is a PDF file, rather than an HTML file.
- The response gets an additional `Content-Disposition` header, which contains the name of the PDF file. This filename is arbitrary: Call it whatever you want. It'll be used by browsers in the Save as... dialogue, and so on.
- The `Content-Disposition` header starts with '`attachment;`' in this example. This forces web browsers to pop-up a dialog box prompting/confirming how to handle the document even if a default is set on the machine. If you leave off '`attachment;`', browsers will handle the PDF using whatever program/plugin they've been configured to use for PDFs. Here's what that code would look like:

```
response['Content-Disposition'] =  
'filename="somefilename.pdf"'
```

- Hooking into the ReportLab API is easy: Just pass `response` as the first argument to `canvas.Canvas`. The `Canvas` class expects a file-like object, and `HttpResponse` objects fit the bill.
- Note that all subsequent PDF-generation methods are called on the PDF object (in this case, `p`)-not on `response`.
- Finally, it's important to call `showPage()` and `save()` on the PDF file.

Complex PDF's

If you're creating a complex PDF document with ReportLab, consider using the `io` library as a temporary holding place for your PDF file. This library provides a file-like object interface that is particularly efficient.

Here's the above Hello World example re-written to use `io`:

```
from io import BytesIO from reportlab.pdfgen import canvas from django.http import HttpResponse def some_view(request): # Create the HttpResponse object with the appropriate PDF headers. response = HttpResponse(content_type='application/pdf') response['Content-Disposition'] = 'attachment; filename="somefilename.pdf"' buffer = BytesIO() # Create the PDF object, using the BytesIO object as its "file." p = canvas.Canvas(buffer) # Draw things on the PDF. Here's where the PDF generation happens. # See the ReportLab documentation for the full list of functionality. p.drawString(100, 100, "Hello world.") # Close the PDF object cleanly. p.showPage() p.save() # Get the value of the BytesIO buffer and write it to the response. pdf = buffer.getvalue() buffer.close() response.write(pdf) return response
```

Further resources

- PDFlib (<http://www.pdflib.org/>) is another PDF-generation library that has Python bindings. To use it with Django, just use the same concepts explained in this article.
- Pisa XHTML2PDF (<http://www.xhtml2pdf.com/>) is yet another PDF-generation library. Pisa ships with an example of how to integrate Pisa with Django.
- HTMLdoc (<http://www.htmldoc.org/>) is a command-line script that can convert HTML to PDF. It doesn't have a Python interface, but you can escape out to the shell using `system` or `popen` and retrieve the output in Python.

Other possibilities

There's a whole host of other types of content you can generate in Python. Here are a few more ideas and some pointers to libraries you could use to implement them:

- **ZIPfiles:** Python's standard library ships with the [zipfile](#) module, which can both read and write compressed ZIP files. You could use it to provide on-demand archives of a bunch of files, or perhaps compress large documents when requested. You could similarly produce TAR files using the standard library's [tarfile](#) module.
- **Dynamic images:** The [Python Imaging Library \(PIL\)](#) (<http://www.pythonware.com/products/pil/>) is a fantastic toolkit for producing images (PNG, JPEG, GIF, and a whole lot more). You could use it to automatically scale down images into thumbnails, composite multiple images into a single frame, or even do web-based image processing.
- **Plots and charts:** There are a number of powerful Python plotting and charting libraries you could use to produce on-demand maps, charts, plots, and graphs. We can't possibly list them all, so here are a couple of the highlights:
 - [matplotlib](#) (<http://matplotlib.sourceforge.net/>) can be used to produce the type of high-quality plots usually generated with MatLab or Mathematica.
 - [pygraphviz](#) (<http://networkx.lanl.gov/pygraphviz/>), an interface to the Graphviz graph layout toolkit, can be used for generating structured diagrams of graphs and networks.

In general, any Python library capable of writing to a file can be hooked into

Django. The possibilities are immense. Now that we've looked at the basics of generating non-HTML content, let's step up a level of abstraction. Django ships with some pretty nifty built-in tools for generating some common types of non-HTML content.

The syndication feed framework

Django comes with a high-level syndication-feed-generating framework that makes creating RSS and Atom feeds easy. RSS and Atom are both XML-based formats you can use to provide automatically updating feeds of your site's content. Read more about RSS here (<http://www.whatisrss.com/>), and get information on Atom here (<http://www.atomenabled.org/>).

To create any syndication feed, all you have to do is write a short Python class. You can create as many feeds as you want. Django also comes with a lower-level feed-generating API. Use this if you want to generate feeds outside of a web context, or in some other lower-level way.

The high-level framework

Overview

The high-level feedgenerating framework is supplied by the `Feed` class. To create a feed, write a `Feed` class and point to an instance of it in your URLconf.

Feed classes

A `Feed` class is a Python class that represents a syndication feed. A feed can be simple (for example, a site news feed, or a basic feed displaying the latest entries of a blog) or more complex (for example, a feed displaying all the blog entries in a particular category, where the category is variable). Feed classes subclass `django.contrib.syndication.views.Feed`. They can live anywhere in your codebase. Instances of `Feed` classes are views which can be used in your URLconf.

A simple example

This simple example, taken from a hypothetical police beat news site describes a feed of the latest five news items:

```
from django.contrib.syndication.views
import Feed
from django.core.urlresolvers import
```

```
reverse
from policebeat.models import NewsItem

class LatestEntriesFeed(Feed):
    title = "Police beat site news"
    link = "sitenews"
    description = "Updates on changes and
additions to police beat central."

    def items(self):
        return
NewsItem.objects.order_by('-pub_date')[:5]

    def item_title(self, item):
        return item.title

    def item_description(self, item):
        return item.description

    # item_link is only needed if NewsItem
has no get_absolute_url method.
    def item_link(self, item):
        return reverse('news-item',
args=[item.pk])
```

To connect a URL to this feed, put an instance of the Feed object in your URLconf. For example:

```
from django.conf.urls import url
from myproject.feeds import
LatestEntriesFeed

urlpatterns = [
    #
    url(r'^latest/feed/$',
LatestEntriesFeed()),
    #
]
```

Note:

- The Feed class subclasses `django.contrib.syndication.views.Feed`.
- `title`, `link` and `description` correspond to the standard RSS `<title>`, `<link>` and `<description>` elements, respectively.
- `items()` is, simply, a method that returns a list of objects that should be included in the feed as `<item>` elements. Although this example returns `NewsItem` objects using Django's object-relational mapper doesn't have to return model instances. Although you get a few bits of functionality for free by using Django models, `items()` can return any type of object you want.
- If you're creating an Atom feed, rather than an RSS feed, set the `subtitle` attribute instead of the `description` attribute. See Publishing Atom and RSS feeds in tandem later in this chapter for an example.

One thing is left to do. In an RSS feed, each `<item>` has a `<title>`, `<link>` and `<description>`. We need to tell the framework what data to put into those elements.

For the contents of `<title>` and `<description>`, Django tries calling the methods `item_title()` and `item_description()` on the `Feed` class. They are passed a single parameter, `item`, which is the object itself. These are optional; by default, the unicode representation of the object is used for both.

If you want to do any special formatting for either the title or description, Django templates can be used instead. Their paths can be specified with the `title_template` and `description_template` attributes on the `Feed`

class. The templates are rendered for each item and are passed two template context variables:

- `{{ obj }}`-: The current object (one of whichever objects you returned in `items()`).
- `{{ site }}`-: A Django `site` object representing the current site. This is useful for `{{ site.domain }}` or `{{ site.name }}`.

See *A complex example* below that uses a description template.

There is also a way to pass additional information to title and description templates, if you need to supply more than the two variables mentioned before. You can provide your implementation of `get_context_data` method in your `Feed` subclass. For example:

```
from mysite.models import Article
from django.contrib.syndication.views
import Feed

class ArticlesFeed(Feed):
    title = "My articles"
    description_template =
"feeds/articles.html"

    def items(self):
        return Article.objects.order_by(
            '-pub_date')[:5]

    def get_context_data(self, **kwargs):
        context = super(ArticlesFeed,
self).get_context_data(**kwargs)
        context['foo'] = 'bar'
        return context
```

And the template:

```
Something about {{ foo }}: {{  
    obj.description }}
```

This method will be called once per item in the list returned by `items()` with the following keyword arguments:

- `item`: the current item. For backward compatibility reasons, the name of this context variable is `{{ obj }}`.
- `obj`: the object returned by `get_object()`. By default, this is not exposed to the templates to avoid confusion with `{{ obj }}` (see above), but you can use it in your implementation of `get_context_data()`.
- `site`: current site as described above.
- `request`: current request.

The behavior of `get_context_data()` mimics that of generic views—you're supposed to call `super()` to retrieve context data from the parent class, add your data and return the modified dictionary.

To specify the contents of `<link>`, you have two options. For each item in `items()`, Django first tries calling the `item_link()` method on the `Feed` class. In a similar way to the title and description, it's passed a single parameter—`item`. If that method doesn't exist, Django tries executing a `get_absolute_url()` method on that object.

Both `get_absolute_url()` and `item_link()` should

return the item's URL as a normal Python string. As with `get_absolute_url()`, the result of `item_link()` will be included directly in the URL, so you are responsible for doing all necessary URL quoting and conversion to ASCII inside the method itself.

A complex example

The framework also supports more complex feeds, via arguments. For example, a website could offer an RSS feed of recent crimes for every police beat in a city. It'd be silly to create a separate `Feed` class for each police beat; that would violate the DRY principle and would couple data to programming logic.

Instead, the syndication framework lets you access the arguments passed from your URLconf so feeds can output items based on information in the feed's URL. The police beat feeds could be accessible via URLs like this:

- `beats613/rss/-`: Returns recent crimes for beat 613.
- `beats1424/rss/-`: Returns recent crimes for beat 1424.

These can be matched with a URLconf line such as:

```
url(r'^beats/(?P[0-9]+)/rss/$',
    BeatFeed()),
```

Like a view, the arguments in the URL are passed to the `get_object()` method along with the request object. Here's the code for these beat-specific feeds:

```
from django.contrib.syndication.views
import FeedDoesNotExist
from django.shortcuts import
get_object_or_404

class BeatFeed(Feed):
    description_template =
'feeds/beat_description.html'

    def get_object(self, request,
beat_id):
        return get_object_or_404(Beat,
pk=beat_id)

    def title(self, obj):
        return "Police beat central:
Crimes for beat %s" % obj.beat

    def link(self, obj):
        return obj.get_absolute_url()

    def description(self, obj):
        return "Crimes recently reported
in police beat %s" % obj.beat

    def items(self, obj):
        return
Crime.objects.filter(beat=obj).order_by(
        '-crime_date')[::30]
```

To generate the feed's `<title>`, `<link>` and `<description>`, Django uses the `title()`, `link()` and `description()` methods.

In the previous example, they were simple string class attributes, but this example illustrates that they can be either strings or methods. For each of `title`, `link` and `description`, Django follows this algorithm:

- First, it tries to call a method, passing the `obj` argument, where `obj` is the object returned by `get_object()`.
- Failing that, it tries to call a method with no arguments.
- Failing that, it uses the class attribute.

Also note that `items()` also follows the same algorithm-first, it tries `items(obj)`, then `items()`, then finally an `items` class attribute (which should be a list). We are using a template for the item descriptions. It can be very simple:

```
{{ obj.description }}
```

However, you are free to add formatting as desired. The `ExampleFeed` class below gives full documentation on methods and attributes of `Feed` classes.

Specifying the type of feed

By default, feeds produced in this framework use RSS 2.0. To change that, add a `feed_type` attribute to your `Feed` class, like so:

```
from django.utils.feedgenerator import  
Atom1Feed  
  
class MyFeed(Feed):  
    feed_type = Atom1Feed
```

Note that you set `feed_type` to a class object, not an instance. Currently available feed types are:

- `django.utils.feedgenerator.Rss201rev2Feed` (RSS 2.01).

Default.)

- `django.utils.feedgenerator.RssUserland091Feed` (RSS 0.91.)
- `django.utils.feedgenerator.Atom1Feed` (Atom 1.0.)

Enclosures

To specify enclosures, such as those used in creating podcast feeds, use the `item_enclosure_url`, `item_enclosure_length` and `item_enclosure_mime_type` hooks. See the `ExampleFeed` class below for usage examples.

Language

Feeds created by the syndication framework automatically include the appropriate `<language>` tag (RSS 2.0) or `xml:lang` attribute (Atom). This comes directly from your `LANGUAGE_CODE` setting.

URLs

The `link` method/attribute can return either an absolute path (for example, `blog`) or a URL with the fully-qualified domain and protocol (for example, `http://www.example.com/blog`). If `link` doesn't return the domain, the syndication framework will insert the domain of the current site, according to your `SITE_ID` setting. Atom feeds require a `<link rel="self">` that defines the feed's current location. The syndication framework populates this automatically,

using the domain of the current site according to the `SITE_ID` setting.

Publishing Atom and RSS Feeds in tandem

Some developers like to make available both Atom and RSS versions of their feeds. That's easy to do with Django: Just create a subclass of your `Feed` class and set the `feed_type` to something different. Then update your URLconf to add the extra versions. Here's a full example:

```
from django.contrib.syndication.views
import Feed
from policebeat.models import NewsItem
from django.utils.feedgenerator import
Atom1Feed

class RssSiteNewsFeed(Feed):
    title = "Police beat site news"
    link = "sitenews"
    description = "Updates on changes and
additions to police beat central."

    def items(self):
        return
NewsItem.objects.order_by('-pub_date')[:5]

class AtomSiteNewsFeed(RssSiteNewsFeed):
    feed_type = Atom1Feed
    subtitle = RssSiteNewsFeed.description
```

NOTE

In this example, the RSS feed uses a `description` while the Atom feed uses a `subtitle`. That's because Atom feeds don't provide for a feed-level description, but they

do provide for a subtitle. If you provide a `description` in your `Feed` class, Django will not automatically put that into the `subtitle` element, because a subtitle and description are not necessarily the same thing. Instead, you should define a `subtitle` attribute.

In the above example, we simply set the Atom feed's `subtitle` to the RSS feed's `description`, because it's quite short already. And the accompanying URLconf:

```
from django.conf.urls import url
from myproject.feeds import
    RssSiteNewsFeed, AtomSiteNewsFeed

urlpatterns = [
    # ...
    url(r'^sitenews/rss/$', RssSiteNewsFeed()),
    url(r'^sitenews/atom/$', AtomSiteNewsFeed()),
    # ...
]
```

NOTE

For an example that illustrates all possible attributes and methods for a `Feed` class, see:
<https://docs.djangoproject.com/en/1.8/ref/contrib/syndication/#feed-class-reference>

The low-level framework

Behind the scenes, the high-level RSS framework uses a lower-level framework for generating feeds' XML. This framework lives in a single module:

[django/utils/feedgenerator.py](#). You use this framework on your own, for lower-level feed generation. You can also create custom feed generator subclasses for use with the [feed_type](#) Feed option.

SyndicationFeed classes

The [feedgenerator](#) module contains a base class:

- [django.utils.feedgenerator.SyndicationFeed](#)

and several subclasses:

- [django.utils.feedgenerator.RssUserland091Feed](#)
- [django.utils.feedgenerator.Rss201rev2Feed](#)
- [django.utils.feedgenerator.Atom1Feed](#)

Each of these three classes knows how to render a certain type of feed as XML. They share this interface:

SYNDICATIONFEED.__INIT__()

Initialize the feed with the given dictionary of metadata, which applies to the entire feed. Required keyword arguments are:

- `title`
- `link`
- `description`

There's also a bunch of other optional keywords:

- `language`
- `author_email`
- `author_name`
- `author_link`
- `subtitle`
- `categories`
- `feed_url`
- `feed_copyright`
- `feed_guid`
- `ttl`

Any extra keyword arguments you pass to `__init__` will be stored in `self.feed` for use with custom feed generators.

All parameters should be Unicode objects, except `categories`, which should be a sequence of Unicode objects.

SYNDICATIONFEED.ADD_ITEM()

Add an item to the feed with the given parameters.

Required keyword arguments are:

- `title`

- `link`
- `description`

Optional keyword arguments are:

- `author_email`
- `author_name`
- `author_link`
- `pubdate`
- `comments`
- `unique_id`
- `enclosure`
- `categories`
- `item_copyright`
- `ttl`
- `updateddate`

Extra keyword arguments will be stored for custom feed generators. All parameters, if given, should be Unicode objects, except:

- `pubdate` should be a Python `datetime` object.
- `updateddate` should be a Python `datetime` object.
- `enclosure` should be an instance of
`django.utils.feedgenerator.Enclosure`.
- `categories` should be a sequence of Unicode objects.

SYNDICATIONFEED.WRITE()

Outputs the feed in the given encoding to `outfile`, which is a file-like object.

SYNDICATIONFEED.WRITESTRING()

Returns the feed as a string in the given encoding. For example, to create an Atom 1.0 feed and print it to standard output:

```
>>> from django.utils import feedgenerator
>>> from datetime import datetime
>>> f = feedgenerator.Atom1Feed(
...     ,
...     link="http://www.example.com/",
...     description="In which I write
about what I ate today.",
...     language="en",
...     author_name="Myself",
...
feed_url="http://example.com/atom.xml")
>>> f.add_item(),
...
link="http://www.example.com/entries/1/",
...     pubdate=datetime.now(),
...     description=<p>Today I had a
Vienna Beef hot dog. It was pink, plump
and perfect.</p>")
>>> print(f.writeString('UTF-8'))
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom"
xml:lang="en">
...
</feed>
```

Custom feed generators

If you need to produce a custom feed format, you've got a couple of options. If the feed format is totally custom, you'll want to subclass [SyndicationFeed](#) and

completely replace the `write()` and `writeString()` methods. However, if the feed format is a spin-off of RSS or Atom (that is, GeoRSS, (link to website <http://georss.org/>), Apple's iTunes podcast format (link to website <http://www.apple.com/itunes/podcasts/specs.html>), and so on.), you've got a better choice.

These types of feeds typically add extra elements and/or attributes to the underlying format, and there are a set of methods that `SyndicationFeed` calls to get these extra attributes. Thus, you can subclass the appropriate feed generator class (`Atom1Feed` or `Rss201rev2Feed`) and extend these call-backs. They are:

SYNDICATIONFEED.ROOT_ATTRIBUTE S(SELF,)

Return a `dict` of attributes to add to the root feed element (`feed/channel`).

SYNDICATIONFEED.ADD_ROOT_ELEME NTS(SELF, HANDLER)

Callback to add elements inside the root feed element (`feed/channel`). `handler` is an `XMLGenerator` from Python's built-in SAX library; you'll call methods on it to add to the XML document in process.

SYNDICATIONFEED.ITEM_ATTRIBUTES (SELF, ITEM)

Return a `dict` of attributes to add to each item (`item/entry`) element. The argument, `item`, is a dictionary of all the data passed to `SyndicationFeed.add_item()`.

SYNDICATIONFEED.ADD_ITEM_ELEMENTS(SELF, HANDLER, ITEM)

Callback to add elements to each item (`item/entry`) element. `handler` and `item` are as above.

NOTE

If you override any of these methods, be sure to call the superclass methods since they add the required elements for each feed format.

For example, you might start implementing an iTunes RSS feed generator like so:

```
class iTunesFeed(Rss201rev2Feed):
    def root_attributes(self):
        attrs = super(iTunesFeed,
self).root_attributes()
        attrs['xmlns:itunes'] =
        'http://www.itunes.com/dtds/podcast-
1.0.dtd'
        return attrs

    def add_root_elements(self, handler):
        super(iTunesFeed,
self).add_root_elements(handler)

        handler.addQuickElement('itunes:explicit',
'clean')
```

Obviously there's a lot more work to be done for a complete custom feed class, but the above example should demonstrate the basic idea.

The Sitemap framework

A **sitemap** is an XML file on your website that tells search engine indexers how frequently your pages change and how important certain pages are in relation to other pages on your site. This information helps search engines index your site. For more on sitemaps, see the [sitemaps.org](#) website.

The Django sitemap framework automates the creation of this XML file by letting you express this information in Python code. It works much like Django's syndication framework. To create a sitemap, just write a [**Sitemap**](#) class and point to it in your URLconf.

Installation

To install the sitemap app, follow these steps:

- Add "`django.contrib.sitemaps`" to your `INSTALLED_APPS` setting.
- Make sure your `TEMPLATES` setting contains a `DjangoTemplates` backend whose `APP_DIRS` option is set to True. It's in there by default, so you'll only need to change this if you've changed that setting.
- Make sure you've installed the sites framework.

Initialization

To activate sitemap generation on your Django site, add

this line to your URLconf:

```
from django.contrib.sitemaps.views import  
sitemap  
  
url(r'^sitemap\.xml$', sitemap,  
{'sitemaps': sitemaps},  
  
name='django.contrib.sitemaps.views.sitema  
p')
```

This tells Django to build a sitemap when a client accesses `/sitemap.xml`. The name of the sitemap file is not important, but the location is. Search engines will only index links in your sitemap for the current URL level and below. For instance, if `sitemap.xml` lives in your root directory, it may reference any URL in your site. However, if your sitemap lives at `contentsitemap.xml`, it may only reference URLs that begin with `content`.

The sitemap view takes an extra, required argument: `{'sitemaps': sitemaps}`. `sitemaps` should be a dictionary that maps a short section label (for example, `blog` or `news`) to its `Sitemap` class (for example, `BlogSitemap` or `NewsSitemap`). It may also map to an instance of a `Sitemap` class (for example, `BlogSitemap(some_var)`).

Sitemap classes

A `Sitemap` class is a simple Python class that

represents a section of entries in your sitemap. For example, one [Sitemap](#) class could represent all the entries of your weblog, while another could represent all of the events in your events calendar.

In the simplest case, all these sections get lumped together into one [sitemap.xml](#), but it's also possible to use the framework to generate a sitemap index that references individual sitemap files, one per section. (See [Creating a sitemap index](#) below.)

[Sitemap](#) classes must subclass [django.contrib.sitemaps.Sitemap](#). They can live anywhere in your codebase.

A simple example

Let's assume you have a blog system, with an [Entry](#) model, and you want your sitemap to include all the links to your individual blog entries. Here's how your sitemap class might look:

```
from django.contrib.sitemaps import  
Sitemap  
from blog.models import Entry  
  
class BlogSitemap(Sitemap):  
    changefreq = "never"  
    priority = 0.5  
  
    def items(self):  
        return  
        Entry.objects.filter(is_draft=False)
```

```
def lastmod(self, obj):  
    return obj.pub_date
```

Note:

- `changefreq` and `priority` are class attributes corresponding to `<changefreq>` and `<pri>` elements, respectively. They can be made callable as functions, as `lastmod` was in the example.
- `items()` is simply a method that returns a list of objects. The objects returned will get passed to any callable methods corresponding to a sitemap property (`location`, `lastmod`, `changefreq` and `priority`).
- `lastmod` should return a Python `datetime` object.
- There is no `location` method in this example, but you can provide it in order to specify the URL for your object. By default, `location()` calls `get_absolute_url()` on each object and returns the result.

Sitemap class reference

A `Sitemap` class can define the following methods/attributes:

ITEMS

Required. A method that returns a list of objects. The framework doesn't care what *type* of objects they are; all that matters is that these objects get passed to the `location()`, `lastmod()`, `changefreq()` and `priority()` methods.

LOCATION

Optional. Either a method or attribute. If it's a method, it should return the absolute path for a given object as

returned by `items()`. If it's an attribute, its value should be a string representing an absolute path to use for every object returned by `items()`.

In both cases, absolute path means a URL that doesn't include the protocol or domain. Examples:

- Good: '`foobar/`'
- Bad: '`example.comfoobar/`'
- Bad: '`http://example.comfoobar/`'

If `location` isn't provided, the framework will call the `get_absolute_url()` method on each object as returned by `items()`. To specify a protocol other than `http`, use `protocol`.

LASTMOD

Optional. Either a method or attribute. If it's a method, it should take one argument—an object as returned by `items()`-and return that object's last-modified date/time, as a Python `datetime.datetime` object.

If it's an attribute, its value should be a Python `datetime.datetime` object representing the last-modified date/time for every object returned by `items()`. If all items in a sitemap have a `lastmod`, the sitemap generated by `views.sitemap()` will have a `Last-Modified` header equal to the latest `lastmod`.

You can activate the `ConditionalGetMiddleware` to make Django respond appropriately to requests with an

If-Modified-Since header which will prevent sending the sitemap if it hasn't changed.

CHANGEFREQ

Optional. Either a method or attribute. If it's a method, it should take one argument-an object as returned by `items()`-and return that object's change frequency, as a Python string. If it's an attribute, its value should be a string representing the change frequency of every object returned by `items()`. Possible values for `changefreq`, whether you use a method or attribute, are:

- 'always'
- 'hourly'
- 'daily'
- 'weekly'
- 'monthly'
- 'yearly'
- 'never'

PRIORITY

Optional. Either a method or attribute. If it's a method, it should take one argument-an object as returned by `items()`-and return that object's priority, as either a string or float.

If it's an attribute, its value should be either a string or float representing the priority of every object returned by `items()`. Example values for `priority`: `0.4, 1.0`.

The default priority of a page is `0.5`. See the [sitemaps.org](#) documentation for more.

PROTOCOL

Optional. This attribute defines the protocol (`http` or `https`) of the URLs in the sitemap. If it isn't set, the protocol with which the sitemap was requested is used. If the sitemap is built outside the context of a request, the default is `http`.

I18N

Optional. A Boolean attribute that defines if the URLs of this sitemap should be generated using all of your [LANGUAGES](#). The default is `False`.

Shortcuts

The sitemap framework provides a convenience class for a common case-

`django.contrib.syndication.GenericSitemap`

The `django.contrib.sitemaps.GenericSitemap` class allows you to create a sitemap by passing it a dictionary which has to contain at least a `queryset` entry. This queryset will be used to generate the items of the sitemap. It may also have a `date_field` entry that specifies a date field for objects retrieved from the `queryset`.

This will be used for the `lastmod` attribute in the generated sitemap. You may also pass `priority` and `changefreq` keyword arguments to the `GenericSitemap` constructor to specify these attributes for all URLs.

EXAMPLE

Here's an example of a URLconf using `GenericSitemap`:

```
from django.conf.urls import url
from django.contrib.sitemaps import
GenericSitemap
from django.contrib.sitemaps.views import
sitemap
from blog.models import Entry

info_dict = {
    'queryset': Entry.objects.all(),
    'date_field': 'pub_date',
}

urlpatterns = [
    # some generic view using info_dict
    # ...

    # the sitemap
    url(r'^sitemap\.xml$', sitemap,
        {'sitemaps': {'blog':
GenericSitemap(info_dict, priority=0.6)}},

    name='django.contrib.sitemaps.views.sitema
p'),
]
```

Sitemap for static views

Often you want the search engine crawlers to index views which are neither object detail pages nor flat pages. The solution is to explicitly list URL names for these views in `items` and call `reverse()` in the `location` method of the sitemap. For example:

```
# sitemaps.py
from django.contrib import sitemaps
from django.core.urlresolvers import
    reverse

class StaticViewSitemap(sitemaps.Sitemap):
    priority = 0.5
    changefreq = 'daily'

    def items(self):
        return ['main', 'about',
    'license']

    def location(self, item):
        return reverse(item)

# urls.py
from django.conf.urls import url
from django.contrib.sitemaps.views import
    sitemap

from .sitemaps import StaticViewSitemap
from . import views

sitemaps = {
    'static': StaticViewSitemap,
}

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

```
        url(r'^about/$', views.about,
            name='about'),
        url(r'^license/$', views.license,
            name='license'),
        # ...
        url(r'^sitemap\.xml$', sitemap,
            {'sitemaps': sitemaps},
            name='django.contrib.sitemaps.views.sitemap')
    ]
```

Creating a sitemap index

The sitemap framework also has the ability to create a sitemap index that references individual sitemap files, one per each section defined in your `sitemaps` dictionary. The only differences in usage are:

- You use two views in your URLconf:
`django.contrib.sitemaps.views.index()` and
`django.contrib.sitemaps.views.sitemap()`.
- The `django.contrib.sitemaps.views.sitemap()` view should take a `section` keyword argument.

Here's what the relevant URLconf lines would look like for the example above:

```
from django.contrib.sitemaps import views

urlpatterns = [
    url(r'^sitemap\.xml$', views.index,
        {'sitemaps': sitemaps}),
    url(r'^sitemap-(?P<section>.+)\.xml$', views.sitemap,
        {'sitemaps': sitemaps}),
]
```

This will automatically generate a `sitemap.xml` file that references both `sitemap-flatpages.xml` and `sitemap-blog.xml`. The `Sitemap` classes and the `sitemaps` dictionary don't change at all.

You should create an index file if one of your sitemaps has more than 50,000 URLs. In this case, Django will automatically paginate the sitemap, and the index will reflect that. If you're not using the vanilla sitemap view—for example, if it's wrapped with a caching decorator—you must name your sitemap view and pass `sitemap_url_name` to the index view:

```
from django.contrib.sitemaps import views
as sitemaps_views
from django.views.decorators.cache import
cache_page

urlpatterns = [
    url(r'^sitemap\.xml$',
        cache_page(86400)
    (sitemaps_views.index),
        {'sitemaps': sitemaps,
    'sitemap_url_name': 'sitemaps'}),
    url(r'^sitemap-(?P<section>.+)\.xml$',
        cache_page(86400)
    (sitemaps_views.sitemap),
        {'sitemaps': sitemaps},
    name='sitemaps'),
]
```

Template customization

If you wish to use a different template for each sitemap

or sitemap index available on your site, you may specify it by passing a `template_name` parameter to the `sitemap` and `index` views via the URLconf:

```
from django.contrib.sitemaps import views

urlpatterns = [
    url(r'^custom-sitemap\.xml$', views.index, {
        'sitemaps': sitemaps,
        'template_name':
        'custom_sitemap.html'
    }),
    url(r'^custom-sitemap-(?
P<section>+)\.xml$', views.sitemap, {
        'sitemaps': sitemaps,
        'template_name': 'custom_sitemap.html'
    }),
]
```

Context variables

When customizing the templates for the `index()` and `sitemap()` views, you can rely on the following context variables.

INDEX

The variable `sitemaps` is a list of absolute URLs to each of the sitemaps.

SITEMAP

The variable `urlset` is a list of URLs that should appear in the sitemap. Each URL exposes attributes as defined

in the `Sitemap` class:

- `changefreq`
- `item`
- `lastmod`
- `location`
- `priority`

The `item` attribute has been added for each URL to allow more flexible customization of the templates, such as Google news sitemaps. Assuming Sitemap's `items()` would return a list of items with `publication_data` and a `tags` field something like this would generate a Google compatible sitemap:

```
{% spaceless %}  
{% for url in urlset %}  
    {{ url.location }}  
    {% if url.lastmod %}{%  
        url.lastmod|date:"Y-m-d" %}{% endif %}  
    {% if url.changefreq %}{%  
        url.changefreq %}{% endif %}  
    {% if url.priority %}{%  
        url.priority %}{% endif %}  
  
    {% if url.item.publication_date %}{%  
        url.item.publication_date|date:"Y-m-d" %}  
    {% endif %}  
    {% if url.item.tags %}{%  
        url.item.tags %}{% endif %}  
  
{% endfor %}  
{% endspaceless %}
```

Pinging google

You may want to ping Google when your sitemap changes, to let it know to reindex your site. The sitemaps framework provides a function to do just that:

DJANGO.CONTRIB.SYNDICATION.PING _GOOGLE()

`ping_google()` takes an optional argument, `sitemap_url`, which should be the absolute path to your site's sitemap (for example, `'/sitemap.xml'`). If this argument isn't provided, `ping_google()` will attempt to figure out your sitemap by performing a reverse looking in your URLconf. `ping_google()` raises the exception `django.contrib.sitemaps.SitemapNotFound` if it cannot determine your sitemap URL.

One useful way to call `ping_google()` is from a model's `save()` method:

```
from django.contrib.sitemaps import
ping_google

class Entry(models.Model):
    # ...
    def save(self, force_insert=False,
force_update=False):
        super(Entry,
self).save(force_insert, force_update)
        try:
            ping_google()
        except Exception:
```

```
# Bare 'except' because we
could get a variety
# of HTTP-related exceptions.
pass
```

A more efficient solution, however, would be to call `ping_google()` from a cron script, or some other scheduled task. The function makes an HTTP request to Google's servers, so you may not want to introduce that network overhead each time you call `save()`.

PINGING GOOGLE VIA MANAGE.PY

Once the sitemaps application is added to your project, you may also ping Google using the `ping_google` management command:

```
python manage.py ping_google
[/sitemap.xml]
```

NOTE

Register with Google first! The `ping_google()` command only works if you have registered your site with Google webmaster Tools.

What's next?

Next, we'll continue to dig deeper into the built-in tools Django gives you by taking a closer look at the Django session framework.

Chapter 15. Django Sessions

Imagine you had to log back in to a website every time you navigated to another page, or your favorite websites forgot all of your settings and you had to enter them again each time you visited?

Modern websites could not provide the usability and convenience we are used to without some way of remembering who you are and your previous activities on the website. HTTP is, by design, *stateless*-there is no persistence between one request and the next, and there is no way the server can tell whether successive requests come from the same person.

This lack of state is managed using *sessions*, which are a semi-permanent, two-way communication between your browser and the web server. When you visit a modern website, in the majority of cases, the web server will use an *anonymous session* to keep track of data relevant to your visit. The session is called anonymous because the web server can only record what you did, not who you are.

We have all experienced this when we have returned to an e-commerce site at a later date and found the items we put in the cart are still there, despite not having provided any personal details. Sessions are most often persisted using the often maligned, but rarely understood

`cookie`. Like all other web frameworks, Django also uses cookies, but does so in a more clever and secure manner, as you will see.

Django provides full support for anonymous sessions. The session framework lets you store and retrieve arbitrary data on a per-site-visitor basis. It stores data on the server side and abstracts the sending and receiving of cookies. Cookies contain a session ID—not the data itself (unless you're using the cookie based backend); a more secure way of implementing cookies than some other frameworks.

Enabling sessions

Sessions are implemented via a piece of middleware. To enable session functionality, edit the `MIDDLEWARE_CLASSES` setting and make sure it contains

`'django.contrib.sessions.middleware.SessionMiddleware'`. The default `settings.py` created by `django-admin startproject` has `SessionMiddleware` activated.

If you don't want to use sessions, you might as well remove the `SessionMiddleware` line from `MIDDLEWARE_CLASSES` and `'django.contrib.sessions'` from your `INSTALLED_APPS`. It'll save you a small bit of overhead.

Configuring the session engine

By default, Django stores sessions in your database (using the model `django.contrib.sessions.models.Session`). Though this is convenient, in some setups it's faster to store session data elsewhere, so Django can be configured to store session data on your filesystem or in your cache.

Using database-backed sessions

If you want to use a database-backed session, you need to add `'django.contrib.sessions'` to your `INSTALLED_APPS` setting. Once you have configured your installation, run `manage.py migrate` to install the single database table that stores session data.

Using cached sessions

For better performance, you may want to use a cache-based session backend. To store session data using Django's cache system, you'll first need to make sure you've configured your cache; see the cache documentation for details.

NOTE

You should only use cache-based sessions if you're using the Memcached cache backend. The local-memory cache backend doesn't retain data long enough to be a good choice, and it'll be faster to use file or database sessions directly instead of sending everything through the file or database cache backends. Additionally, the local-memory cache backend is NOT multi-process safe, therefore probably not a good choice for production environments.

If you have multiple caches defined in `CACHES`, Django will use the default cache. To use another cache, set `SESSION_CACHE_ALIAS` to the name of that cache. Once your cache is configured, you've got two choices for how to store data in the cache:

- Set `SESSION_ENGINE` to `"django.contrib.sessions.backends.cache"` for a simple caching session store. Session data will be stored directly in your cache. However, session data may not be persistent: cached data can be evicted if the cache fills up or if the cache server is restarted.
- For persistent, cached data, set `SESSION_ENGINE` to `"django.contrib.sessions.backends.cached_db"`. This uses a write-through cache—every write to the cache will also be written to the database. Session reads only use the database if the data is not already in the cache.

Both session stores are quite fast, but the simple cache is faster because it disregards persistence. In most cases, the `cached_db` backend will be fast enough, but if you need that last bit of performance, and are willing to let session data be expunged from time to time, the `cache` backend is for you. If you use the `cached_db` session backend, you also need to follow the configuration instructions for the using database-backed sessions.

Using file-based sessions

To use file-based sessions, set the `SESSION_ENGINE` setting to

`"django.contrib.sessions.backends.file"`.

You might also want to set the `SESSION_FILE_PATH` setting (which defaults to output from `tempfile.gettempdir()`, most likely `/tmp`) to control where Django stores session files. Be sure to check that your web server has permissions to read and write to this location.

Using cookie-based sessions

To use cookies-based sessions, set the

`SESSION_ENGINE` setting to

`"django.contrib.sessions.backends.signed_cookies"`. The session data will be stored using Django's tools for cryptographic signing and the `SECRET_KEY` setting.

It's recommended to leave the

`SESSION_COOKIE_HTTPONLY` setting on `True` to prevent access to the stored data from JavaScript.

NOTE

If the `SECRET_KEY` is not kept secret and you are using the `PickleSerializer`, this can lead to arbitrary remote code execution.

An attacker in possession of the `SECRET_KEY` can not only generate falsified session data, which your site will trust, but also remotely execute arbitrary code, as the data is serialized using pickle. If you use cookie-based sessions, pay extra care that your secret key is always kept completely secret, for any system which might be remotely accessible.

NOTE

The session data is signed but not encrypted

When using the cookies backend, the session data can be read by the client. A MAC (Message Authentication Code) is used to protect the data against changes by the client, so that the session data will be invalidated when being tampered with. The same invalidation happens if the client storing the cookie (for example, your user's browser) can't store all of the session cookie and drops data. Even though Django compresses the data, it's still entirely possible to exceed the common limit of 4096 bytes per cookie.

NOTE

No freshness guarantee

Note also that while the MAC can guarantee the authenticity of the data (that it was generated by your site, and not someone else), and the integrity of the data (that it is all there and correct), it cannot guarantee freshness that is, you are being sent back the last thing you sent to the client. This means that for some uses of session data, the cookie backend might open you up to replay attacks. Unlike other session backends which keep a server-side record of each session and invalidate it when a user logs out, cookie-based sessions are not invalidated when a user logs out. Thus if an attacker steals a user's cookie, they can use that cookie to login as that user even if the user logs out. Cookies will only be detected as 'stale' if they are older than your `SESSION_COOKIE_AGE`.

Finally, assuming the above warnings have not discouraged you from using cookie based sessions: the size of a cookie can also have an impact on the speed of your site.

Using Sessions in Views

When `SessionMiddleware` is activated, each `HttpRequest` object—the first argument to any Django view function—will have a `session` attribute, which is a dictionary-like object. You can read it and write to `request.session` at any point in your view. You can edit it multiple times.

All session objects inherit from the base class `backends.base.SessionBase`. It has the following standard dictionary methods:

- `__getitem__(key)`
- `__setitem__(key, value)`
- `__delitem__(key)`
- `__contains__(key)`
- `get(key, default=None)`
- `pop(key)`
- `keys()`
- `items()`
- `setdefault()`
- `clear()`

It also has these methods:

flush()

Delete the current session data from the session and delete the session cookie. This is used if you want to ensure that the previous session data can't be accessed again from the user's browser (for example, the `django.contrib.auth.logout()` function calls it).

set_test_cookie()

Sets a test cookie to determine whether the user's browser supports cookies. Due to the way cookies work, you won't be able to test this until the user's next page request. See *Setting test cookies* below for more information.

test_cookie_worked()

Returns either `True` or `False`, depending on whether the user's browser accepted the test cookie. Due to the way cookies work, you'll have to call `set_test_cookie()` on a previous, separate page request. See *Setting test cookies* below for more information.

delete_test_cookie()

Deletes the test cookie. Use this to clean up after yourself.

set_expiry(value)

Sets the expiration time for the session. You can pass a

number of different values:

- If `value` is an integer, the session will expire after that many seconds of inactivity. For example, calling `request.session.set_expiry(300)` would make the session expire in 5 minutes.
- If `value` is a `datetime` or `timedelta` object, the session will expire at that specific date/time. Note that `datetime` and `timedelta` values are only serializable if you are using the `PickleSerializer`.
- If `value` is `0`, the user's session cookie will expire when the user's web browser is closed.
- If `value` is `None`, the session reverts to using the global session expiry policy.

Reading a session is not considered activity for expiration purposes. Session expiration is computed from the last time the session was modified.

get_expiry_age()

Returns the number of seconds until this session expires. For sessions with no custom expiration (or those set to expire at browser close), this will equal `SESSION_COOKIE_AGE`. This function accepts two optional keyword arguments:

- `modification`: last modification of the session, as a `datetime` object. Defaults to the current time
- `expiry`: expiry information for the session, as a `datetime` object, an `int` (in seconds), or `None`. Defaults to the value stored in the session by `set_expiry()`, if there is one, or `None`

get_expiry_date()

Returns the date this session will expire. For sessions with no custom expiration (or those set to expire at browser close), this will equal the date `SESSION_COOKIE_AGE` seconds from now. This function accepts the same keyword arguments as `get_expiry_age()`.

get_expire_at_browser_close()

Returns either `True` or `False`, depending on whether the user's session cookie will expire when the user's web browser is closed.

clear_expired()

Removes expired sessions from the session store. This class method is called by `clearsessions`.

cycle_key()

Creates a new session key while retaining the current session data. `django.contrib.auth.login()` calls this method to mitigate against session fixation.

Session object guidelines

- Use normal Python strings as dictionary keys on `request.session`.
This is more of a convention than a hard-and-fast rule.
- Session dictionary keys that begin with an underscore are reserved
for internal use by Django.

Don't override `request.session` with a new object,
and don't access or set its attributes. Use it like a Python
dictionary.

Session serialization

Before version 1.6, Django defaulted to using `pickle` to serialize session data before storing it in the backend. If you're using the signed cookie session backend and `SECRET_KEY` is known by an attacker (there isn't an inherent vulnerability in Django that would cause it to leak), the attacker could insert a string into their session which, when unpickled, executes arbitrary code on the server. The technique for doing so is simple and easily available on the internet.

Although the cookie session storage signs the cookie-stored data to prevent tampering, a `SECRET_KEY` leak immediately escalates to a remote code execution vulnerability. This attack can be mitigated by serializing session data using JSON rather than `pickle`. To facilitate this, Django 1.5.3 introduced a new setting, `SESSION_SERIALIZER`, to customize the session serialization format. For backwards compatibility, this setting defaults to using `django.contrib.sessions.serializers.PickleSerializer` in Django 1.5.x, but, for security hardening, defaults to `django.contrib.sessions.serializers.JSONSerializer` from Django 1.6 onwards.

Even with the caveats described in custom-serializers, we highly recommend sticking with JSON serialization

especially if you are using the cookie backend.

Bundled serializers

SERIALIZERS.JSONSERIALIZER

A wrapper around the JSON serializer from `django.core.signing`. Can only serialize basic data types. In addition, as JSON supports only string keys, note that using non-string keys in `request.session` won't work as expected:

```
>>> # initial assignment
>>> request.session[0] = 'bar'
>>> # subsequent requests following
     serialization & deserialization
>>> # of session data
>>> request.session[0]  # KeyError
>>> request.session['0']
'bar'
```

See the custom-serializers section for more details on limitations of JSON serialization.

SERIALIZERS.PICKLESERIALIZER

Supports arbitrary Python objects, but, as described above, can lead to a remote code execution vulnerability if `SECRET_KEY` becomes known by an attacker.

Write your own serializer

Note that unlike `PickleSerializer`, the

`JSONSerializer` cannot handle arbitrary Python data types. As is often the case, there is a trade-off between convenience and security. If you wish to store more advanced data types including `datetime` and `Decimal` in JSON backed sessions, you will need to write a custom serializer (or convert such values to a JSON serializable object before storing them in `request.session`).

While serializing these values is fairly straightforward (`django.core.serializers.json.DateTimeAwareJSONEncoder` may be helpful), writing a decoder that can reliably get back the same thing that you put in is more fragile. For example, you run the risk of returning a `datetime` that was actually a string that just happened to be in the same format chosen for `datetime`).

Your serializer class must implement two methods, `dumps(self, obj)` and `loads(self, data)`, to serialize and deserialize the dictionary of session data, respectively.

Setting test cookies

As a convenience, Django provides an easy way to test whether the user's browser accepts cookies. Just call the `set_test_cookie()` method of `request.session` in a view, and call `test_cookie_worked()` in a subsequent view-not in the same view call.

This awkward split between `set_test_cookie()` and `test_cookie_worked()` is necessary due to the way cookies work. When you set a cookie, you can't actually tell whether a browser accepted it until the browser's next request. It's good practice to use `delete_test_cookie()` to clean up after yourself. Do this after you've verified that the test cookie worked.

Here's a typical usage example:

```
def login(request):
    if request.method == 'POST':
        if
request.session.test_cookie_worked():

    request.session.delete_test_cookie()
        return HttpResponseRedirect("You're
logged in.")
    else:
        return HttpResponseRedirect("Please
enable cookies and try again.")
    request.session.set_test_cookie()
    return
render_to_response('foo/login_form.html')
```

Using sessions out of views

The examples in this section import the `SessionStore` object directly from the

`django.contrib.sessions.backends.db` backend. In your own code, you should consider importing `SessionStore` from the session engine designated by `SESSION_ENGINE`, as below:

```
>>> from
importlib import import_module
>>> from django.conf
import settings
>>> SessionStore =
import_module(settings.SESSION_ENGINE).SessionStore
```

An API is available to manipulate session data outside of a view:

```
>>> from
django.contrib.sessions.backends.db import
SessionStore
>>> s = SessionStore()
>>> # stored as seconds since epoch since
datetimes are not serializable in JSON.
>>> s['last_login'] = 1376587691
>>> s.save()
>>> s.session_key
'2b1189a188b44ad18c35e113ac6ceead'

>>> s =
SessionStore(session_key='2b1189a188b44ad1
8c35e113ac6ceead')
>>> s['last_login']
1376587691
```

In order to mitigate session fixation attacks, sessions keys that don't exist are regenerated:

```
>>> from  
django.contrib.sessions.backends.db import  
SessionStore  
>>> s = SessionStore(session_key='no-such-  
session-here')  
>>> s.save()  
>>> s.session_key  
'ff882814010ccbc3c870523934fee5a2'
```

If you're using the [django.contrib.sessions.backends.db](#) backend, each session is just a normal Django model. The [Session](#) model is defined in [django/contrib/sessions/models.py](#). Because it's a normal model, you can access sessions using the normal Django database API: >>> from django.contrib.sessions.models import Session >>> s = Session.objects.get(pk='2b1189a188b44ad18c35e113ac6ceead') >>> s.expire_date datetime.datetime(2005, 8, 20, 13, 35, 12) Note that you'll need to call `get_decoded()` to get the session dictionary. This is necessary because the dictionary is stored in an encoded format: >>> s.session_data
'KGRwMQpTJ19hdXRoX3VzZXJfaWQnCnAyCkkxCnMuMTExY2ZjODI2Yj...' >>> s.get_decoded() {'user_id': 42}

When sessions are saved

By default, Django only saves to the session database when the session has been modified—that is if any of its dictionary values have been assigned or deleted:

```
# Session is modified.  
request.session['foo'] = 'bar'  
  
# Session is modified.  
del request.session['foo']  
  
# Session is modified.  
request.session['foo'] = {}  
  
# Gotcha: Session is NOT modified, because  
this alters  
# request.session['foo'] instead of  
request.session.  
request.session['foo']['bar'] = 'baz'
```

In the last case of the above example, we can tell the session object explicitly that it has been modified by setting the `modified` attribute on the session object:

```
request.session.modified = True
```

To change this default behavior, set the `SESSION_SAVE_EVERY_REQUEST` setting to `True`.

When set to `True`, Django will save the session to the database on every single request. Note that the session cookie is only sent when a session has been created or

modified. If `SESSION_SAVE_EVERY_REQUEST` is `True`, the session cookie will be sent on every request.

Similarly, the `expires` part of a session cookie is updated each time the session cookie is sent. The session is not saved if the response's status code is 500.

Browser-length sessions vs. persistent sessions

You can control whether the session framework uses browser-length sessions vs. persistent sessions with the `SESSION_EXPIRE_AT_BROWSER_CLOSE` setting. By default, `SESSION_EXPIRE_AT_BROWSER_CLOSE` is set to `False`, which means session cookies will be stored in users' browsers for as long as `SESSION_COOKIE_AGE`. Use this if you don't want people to have to log in every time they open a browser.

If `SESSION_EXPIRE_AT_BROWSER_CLOSE` is set to `True`, Django will use browser-length cookies-cookies that expire as soon as the user closes their browser.

NOTE

Some browsers (Chrome, for example) provide settings that allow users to continue browsing sessions after closing and re-opening the browser. In some cases, this can interfere with the `SESSION_EXPIRE_AT_BROWSER_CLOSE` setting and prevent sessions from expiring on browser close. Please be aware of this while testing Django applications which have the `SESSION_EXPIRE_AT_BROWSER_CLOSE` setting enabled.

Clearing the session store

As users create new sessions on your website, session data can accumulate in your session store. Django does not provide automatic purging of expired sessions.

Therefore, it's your job to purge expired sessions on a regular basis. Django provides a clean-up management command for this purpose: `clearsessions`. It's recommended to call this command on a regular basis, for example as a daily cron job.

Note that the cache backend isn't vulnerable to this problem, because caches automatically delete stale data. Neither is the cookie backend, because the session data is stored by the users' browsers.

What's next

Next, we will be continuing our look into more advanced Django topics by examining Django's caching backend.

Chapter 16. Djangos Cache Framework

A fundamental trade-off in dynamic websites is, well, they're dynamic. Each time a user requests a page, the web server makes all sorts of calculations, from database queries to template rendering to business logic to creating the page that your site's visitors see. This is a lot more expensive, from a processing-overhead perspective, than your standard read-a-file-off-the-filesystem server arrangement.

For most web applications, this overhead isn't a big deal. Most web applications aren't www.washingtonpost.com or www.slashdot.org; they're simply small-to medium-sized sites with so-so traffic. But for medium-to high-traffic sites, it's essential to cut as much overhead as possible.

That's where caching comes in. To cache something is to save the result of an expensive calculation so that you don't have to perform the calculation next time. Here's some pseudocode explaining how this would work for a dynamically generated web page:

```
given a URL, try finding that page in the
cache
if the page is in the cache:
    return the cached page
else:
```

```
    generate the page  
    save the generated page in the cache  
(for next time)  
    return the generated page
```

Django comes with a robust cache system that lets you save dynamic pages so they don't have to be calculated for each request. For convenience, Django offers different levels of cache granularity: You can cache the output of specific views, you can cache only the pieces that are difficult to produce, or you can cache your entire site.

Django also works well with downstream caches, such as Squid (for more information visit <http://www.squid-cache.org/>) and browser-based caches. These are the types of caches that you don't directly control, but to which you can provide hints (via HTTP headers) about which parts of your site should be cached, and how.

Setting up the cache

The cache system requires a small amount of setup. Namely, you have to tell it where your cached data should live; whether in a database, on the filesystem or directly in memory. This is an important decision that affects your cache's performance.

Your cache preference goes in the `CACHES` setting in your settings file.

Memcached

The fastest, most efficient type of cache supported natively by Django, Memcached (for more information visit <http://memcached.org/>) is an entirely memory-based cache server, originally developed to handle high loads at LiveJournal.com and subsequently open-sourced by Danga Interactive. It's used by sites such as Facebook and Wikipedia to reduce database access and dramatically increase site performance.

Memcached runs as a daemon and is allotted a specified amount of RAM. All it does is provide a fast interface for adding, retrieving and deleting data in the cache. All data is stored directly in memory, so there's no overhead of database or filesystem usage.

After installing Memcached itself, you'll need to install a Memcached binding. There are several Python Memcached bindings available; the two most common are `python-memcached` (<ftp://ftp.tummy.com/pub/python-memcached/>) and `pylibmc` (<http://sendapatch.se/projects/pylibmc/>). To use Memcached with Django:

- Set `BACKEND` to
`django.core.cache.backends.memcached.MemcachedCache` or `django.core.cache.backends.memcached.PyLibMCCache` (depending on your chosen memcached binding)
- Set `LOCATION` to `ip:port` values, where `ip` is the IP address of the Memcached daemon and `port` is the port on which Memcached is running, or to a `unix:path` value, where `path` is the path to a

Memcached Unix socket file.

In this example, Memcached is running on localhost (127.0.0.1) port 11211, using the [python-memcached](#) binding:

```
CACHES = {
    'default': {
        'BACKEND':
'django.core.cache.backends.memcached.Memc
achedCache',
        'LOCATION': '127.0.0.1:11211',
    }
}
```

In this example, Memcached is available through a local Unix socket file [*tmpmemcached.sock*](#) using the [python-memcached](#) binding:

```
CACHES = {
    'default': {
        'BACKEND':
'django.core.cache.backends.memcached.Memc
achedCache',
        'LOCATION':
'unix:/tmpmemcached.sock',
    }
}
```

One excellent feature of Memcached is its ability to share a cache over multiple servers. This means you can run Memcached daemons on multiple machines, and the program will treat the group of machines as a *single* cache, without the need to duplicate cache values on each machine. To take advantage of this feature,

include all server addresses in `LOCATION`, either separated by semicolons or as a list.

In this example, the cache is shared over Memcached instances running on IP address `172.19.26.240` and `172.19.26.242`, both on port 11211:

```
CACHES = {
    'default': {
        'BACKEND':
'django.core.cache.backends.memcached.Memc
achedCache',
        'LOCATION': [
            '172.19.26.240:11211',
            '172.19.26.242:11211',
        ]
    }
}
```

In the following example, the cache is shared over Memcached instances running on the IP addresses `172.19.26.240` (port 11211), `172.19.26.242` (port 11212), and `172.19.26.244` (port 11213):

```
CACHES = {
    'default': {
        'BACKEND':
'django.core.cache.backends.memcached.Memc
achedCache',
        'LOCATION': [
            '172.19.26.240:11211',
            '172.19.26.242:11212',
            '172.19.26.244:11213',
        ]
    }
}
```

A final point about Memcached is that memory-based caching has a disadvantage: because the cached data is stored in memory, the data will be lost if your server crashes.

Clearly, memory isn't intended for permanent data storage, so don't rely on memory-based caching as your only data storage. Without a doubt, none of the Django caching backends should be used for permanent storage-they're all intended to be solutions for caching, not storage-but we point this out here because memory-based caching is particularly temporary.

Database caching

Django can store its cached data in your database. This works best if you've got a fast, well-indexed database server. To use a database table as your cache backend:

- Set `BACKEND` to
`django.core.cache.backends.db.DatabaseCache`
- Set `LOCATION` to `tablename`, the name of the database table. This name can be whatever you want, as long as it's a valid table name that's not already being used in your database.

In this example, the cache table's name is `my_cache_table`:

```
CACHES = {
    'default': {
        'BACKEND':
            'django.core.cache.backends.db.DatabaseCac
he',
        'LOCATION': 'my_cache_table',
```

```
    }  
}
```

CREATING THE CACHE TABLE

Before using the database cache, you must create the cache table with this command:

```
python manage.py createcachetable
```

This creates a table in your database that is in the proper format that Django's database-cache system expects.

The name of the table is taken from [LOCATION](#). If you are using multiple database caches,

`createcachetable` creates one table for each cache.

If you are using multiple databases,

`createcachetable` observes the [allow_migrate\(\)](#) method of your database routers (see below). Like `migrate`, `createcachetable` won't touch an existing table. It will only create missing tables.

MULTIPLE DATABASES

If you use database caching with multiple databases, you'll also need to set up routing instructions for your database cache table. For the purposes of routing, the database cache table appears as a model named `CacheEntry`, in an application named `django_cache`.

This model won't appear in the model cache, but the model details can be used for routing purposes.

For example, the following router would direct all cache

read operations to `cache_replica`, and all write operations to `cache_primary`. The cache table will only be synchronized onto `cache_primary`:

```
class CacheRouter(object):
    """A router to control all database
    cache operations"""

    def db_for_read(self, model, **hints):
        # All cache read operations go to
        the replica
        if model._meta.app_label in
('django_cache',):
            return 'cache_replica'
        return None

    def db_for_write(self, model,
    **hints):
        # All cache write operations go to
        primary
        if model._meta.app_label in
('django_cache',):
            return 'cache_primary'
        return None

    def allow_migrate(self, db, model):
        # Only install the cache model on
        primary
        if model._meta.app_label in
('django_cache',):
            return db == 'cache_primary'
        return None
```

If you don't specify routing directions for the database cache model, the cache backend will use the `default` database. Of course, if you don't use the database cache

backend, you don't need to worry about providing routing instructions for the database cache model.

Filesystem caching

The filebased backend serializes and stores each cache value as a separate file. To use this backend set

`BACKEND` to

`'django.core.cache.backends.filebased.FileBasedCache'` and `LOCATION` to a suitable directory.

For example, to store cached data in
`/var/tmpdjango_cache`, use this setting:

```
CACHES = {
    'default': {
        'BACKEND':
            'django.core.cache.backends.filebased.File
            BasedCache',
        'LOCATION': '/var/tmpdjango_cache',
    }
}
```

If you're on Windows, put the drive letter at the beginning of the path, like this:

```
CACHES = {
    'default': {
        'BACKEND':
            'django.core.cache.backends.filebased.File
            BasedCache',
        'LOCATION': 'c:/foo/bar',
    }
}
```

The directory path should be absolute—that is, it should start at the root of your filesystem. It doesn't matter whether you put a slash at the end of the setting. Make sure the directory pointed to by this setting exists and is readable and writable by the system user under which your web server runs. Continuing the above example, if your server runs as the user `apache`, make sure the directory `/var/tmp/django_cache` exists and is readable and writable by the user `apache`.

Local-memory caching

This is the default cache if another is not specified in your settings file. If you want the speed advantages of in-memory caching but don't have the capability of running Memcached, consider the local-memory cache backend. To use it, set `BACKEND` to `django.core.cache.backends.locmem.LocMemCache`. For example:

```
CACHES = {
    'default': {
        'BACKEND':
            'django.core.cache.backends.locmem.LocMemC
ache',
        'LOCATION': 'unique-snowflake'
    }
}
```

The cache `LOCATION` is used to identify individual memory stores. If you only have one `locmem` cache, you can omit the `LOCATION`; however, if you have more than

one local memory cache, you will need to assign a name to at least one of them in order to keep them separate.

Note that each process will have its own private cache instance, which means no cross-process caching is possible. This obviously also means the local memory cache isn't particularly memory-efficient, so it's probably not a good choice for production environments. It's nice for development.

Dummy caching (for development)

Finally, Django comes with a dummy cache that doesn't actually cache—it just implements the cache interface without doing anything. This is useful if you have a production site that uses heavy-duty caching in various places but a development/test environment where you don't want to cache and don't want to have to change your code to special-case the latter. To activate dummy caching, set `BACKEND` like so:

```
CACHES = {  
    'default': {  
        'BACKEND':  
            'django.core.cache.backends.dummy.DummyCac  
he',  
        }  
}
```

Using a custom cache backend

While Django includes support for a number of cache backends out-of-the-box, sometimes you might want to use a customized cache backend. To use an external cache backend with Django, use the Python import path as the `BACKEND` of the `CACHES` setting, like so:

```
CACHES = {  
    'default': {  
        'BACKEND': 'path.to.backend',  
    }  
}
```

If you're building your own backend, you can use the standard cache backends as reference implementations. You'll find the code in the [django/core/cache/backends/](#) directory of the Django source.

NOTE

Without a really compelling reason, such as a host that doesn't support them, you should stick to the cache backends included with Django. They've been well-tested and are easy to use.

Cache arguments

Each cache backend can be given additional arguments to control caching behavior. These arguments are provided as additional keys in the `CACHES` setting. Valid arguments are as follows:

- `TIMEOUT`: The default timeout, in seconds, to use for the cache. This argument defaults to 300 seconds (5 minutes). You can set `TIMEOUT` to `None` so that, by default, cache keys never expire. A value of `0` causes keys to immediately expire (effectively don't cache).

- **OPTIONS**: Any options that should be passed to the cache backend. The list of valid options will vary with each backend, and cache backends backed by a third-party library will pass their options directly to the underlying cache library.
- Cache backends that implement their own culling strategy (that is, the `locmem`, `filesystem` and `database` backends) will honor the following options:
 - **MAX_ENTRIES**: The maximum number of entries allowed in the cache before old values are deleted. This argument defaults to `300`.
 - **CULL_FREQUENCY**: The fraction of entries that are culled when `MAX_ENTRIES` is reached. The actual ratio is $1 / \text{CULL_FREQUENCY}$, so set `CULL_FREQUENCY` to `2` to cull half the entries when `MAX_ENTRIES` is reached. This argument should be an integer and defaults to `3`.
 - A value of `0` for `CULL_FREQUENCY` means that the entire cache will be dumped when `MAX_ENTRIES` is reached. On some backends (`database` in particular) this makes culling *much* faster at the expense of more cache misses.
- **KEY_PREFIX**: A string that will be automatically included (prepended by default) to all cache keys used by the Django server.
- **VERSION**: The default version number for cache keys generated by the Django server.
- **KEY_FUNCTION**: A string containing a dotted path to a function that defines how to compose a prefix, version, and key into a final cache key.

In this example, a filesystem backend is being configured with a timeout of 60 seconds, and a maximum capacity of 1000 items:

```
CACHES = {
    'default': {
        'BACKEND':
            'django.core.cache.backends.filebased.File
BasedCache',
```

```
        'LOCATION': '/var/tmp/django_cache',
        'TIMEOUT': 60,
        'OPTIONS': {'MAX_ENTRIES': 1000}
    }
}
```

The per-site cache

Once the cache is set up, the simplest way to use caching is to cache your entire site. You'll need to add `'django.middleware.cache.UpdateCacheMiddleware'` and `'django.middleware.cache.FetchFromCacheMiddleware'` to your `MIDDLEWARE_CLASSES` setting, as in this example:

```
MIDDLEWARE_CLASSES = [  
  
    'django.middleware.cache.UpdateCacheMiddleware',  
  
    'django.middleware.common.CommonMiddleware',  
  
    'django.middleware.cache.FetchFromCacheMiddleware',  
]
```

NOTE

No, that's not a typo: the update middleware must be first in the list, and the fetch middleware must be last. The details are a bit obscure, but see Order of `MIDDLEWARE_CLASSES` in the next chapter if you'd like the full story.

Then, add the following required settings to your Django settings file:

- `CACHE_MIDDLEWARE_ALIAS`: The cache alias to use for storage.
- `CACHE_MIDDLEWARE_SECONDS`: The number of seconds each page should be cached.

- `CACHE_MIDDLEWARE_KEY_PREFIX`-: If the cache is shared across multiple sites using the same Django installation, set this to the name of the site, or some other string that is unique to this Django instance, to prevent key collisions. Use an empty string if you don't care.

`FetchFromCacheMiddleware` caches `GET` and `HEAD` responses with `status 200`, where the request and response headers allow. Responses to requests for the same URL with different query parameters are considered to be unique pages and are cached separately. This middleware expects that a `HEAD` request is answered with the same response headers as the corresponding `GET` request; in which case it can return a cached `GET` response for `HEAD` request. Additionally, `UpdateCacheMiddleware` automatically sets a few headers in each `HttpResponse`:

- Sets the `Last-Modified` header to the current date/time when a fresh (not cached) version of the page is requested.
- Sets the `Expires` header to the current date/time plus the defined `CACHE_MIDDLEWARE_SECONDS`.
- Sets the `Cache-Control` header to give a max age for the page-again, from the `CACHE_MIDDLEWARE_SECONDS` setting.

If a view sets its own cache expiry time (that is, it has a `max-age` section in its

`Cache-Control` header) then the page will be cached until the expiry time, rather than `CACHE_MIDDLEWARE_SECONDS`. Using the decorators in `django.views.decorators.cache` you can easily set a view's expiry time (using the `cache_control` decorator) or disable caching for a view (using the

`never_cache` decorator). See the using other headers section for more on these decorators.

If `USE_I18N` is set to `True` then the generated cache key will include the name of the active language. This allows you to easily cache multilingual sites without having to create the cache key yourself.

Cache keys also include the active language when `USE_L10N` is set to `True` and the current time zone when `USE_TZ` is set to `True`.

The per-view cache

A more granular way to use the caching framework is by caching the output of individual views.

`django.views.decorators.cache` defines a `cache_page` decorator that will automatically cache the view's response for you. It's easy to use:

```
from django.views.decorators.cache import
cache_page

@cache_page(60 * 15)
def my_view(request):
    ...
```

`cache_page` takes a single argument: the cache timeout, in seconds. In the above example, the result of the `my_view()` view will be cached for 15 minutes. (Note that I've written it as `60 * 15` for the purpose of readability. `60 * 15` will be evaluated to `900`-that is, 15 minutes multiplied by 60 seconds per minute.)

The per-view cache, like the per-site cache, is keyed off of the URL. If multiple URLs point at the same view, each URL will be cached separately. Continuing the `my_view` example, if your URLconf looks like this:

```
urlpatterns = [
    url(r'^foo/([0-9]{1,2})/$', my_view),
]
```

then requests to `foo1/` and `foo23/` will be cached separately, as you may expect. But once a particular URL (for example, `foo23/`) has been requested, subsequent requests to that URL will use the cache.

`cache_page` can also take an optional keyword argument, `cache`, which directs the decorator to use a specific cache (from your `CACHES` setting) when caching view results.

By default, the `default` cache will be used, but you can specify any cache you want:

```
@cache_page(60 * 15,
cache="special_cache")
def my_view(request):
    ...
```

You can also override the cache prefix on a per-view basis. `cache_page` takes an optional keyword argument, `key_prefix`, which works in the same way as the `CACHE_MIDDLEWARE_KEY_PREFIX` setting for the middleware. It can be used like this:

```
@cache_page(60 * 15, key_prefix="site1")
def my_view(request):
    ...
```

The `key_prefix` and `cache` arguments may be specified together. The `key_prefix` argument and the `KEY_PREFIX` specified under `CACHES` will be concatenated.

Specifying per-view Cache in the URLconf

The examples in the previous section have hard-coded the fact that the view is cached, because `cache_page` alters the `my_view` function in place. This approach couples your view to the cache system, which is not ideal for several reasons. For instance, you might want to reuse the view functions on another, cache-less site, or you might want to distribute the views to people who might want to use them without being cached.

The solution to these problems is to specify the per-view cache in the URLconf rather than next to the view functions themselves. Doing so is easy: simply wrap the view function with `cache_page` when you refer to it in the URLconf.

Here's the old URLconf from earlier:

```
urlpatterns = [
    url(r'^foo/([0-9]{1,2})/$', my_view),
]
```

Here's the same thing, with `my_view` wrapped in `cache_page`:

```
from django.views.decorators.cache import
cache_page

urlpatterns = [
    url(r'^foo/([0-9]{1,2})/$',
        cache_page(60 * 15)(my_view))
```

```
    cache_page(0000000000000000ULL, my_vlow),
```

```
]
```

Template fragment caching

If you're after even more control, you can also cache template fragments using the `cache` template tag. To give your template access to this tag, put

`{% load cache %}` near the top of your template. The `{% cache %}` template tag caches the contents of the block for a given amount of time.

It takes at least two arguments: the cache timeout, in seconds, and the name to give the cache fragment. The name will be taken as is, do not use a variable.

For example:

```
{% load cache %}
{% cache 500 sidebar %}
    .. sidebar ..
{% endcache %}
```

Sometimes you might want to cache multiple copies of a fragment depending on some dynamic data that appears inside the fragment.

For example, you might want a separate cached copy of the sidebar used in the previous example for every user of your site. Do this by passing additional arguments to the `{% cache %}` template tag to uniquely identify the cache fragment:

```
{% load cache %}  
{% cache 500 sidebar request.user.username  
%}  
    .. sidebar for logged in user ..  
{% endcache %}
```

It's perfectly fine to specify more than one argument to identify the fragment. Simply pass as many arguments to `{% cache %}` as you need. If `USE_I18N` is set to `True` the per-site middleware cache will respect the active language.

For the `cache` template tag you could use one of the translation-specific variables available in templates to achieve the same result:

```
{% load i18n %}  
{% load cache %}  
  
{% get_current_language as LANGUAGE_CODE  
%}  
  
{% cache 600 welcome LANGUAGE_CODE %}  
    {% trans "Welcome to example.com" %}  
{% endcache %}
```

The cache timeout can be a template variable, as long as the template variable resolves to an integer value.

For example, if the template variable `my_timeout` is set to the value `600`, then the following two examples are equivalent:

```
{% cache 600 sidebar %} ... {% endcache %}  
{% cache my_timeout sidebar %} ... {%
```

```
    endcache %}
```

This feature is useful in avoiding repetition in templates. You can set the timeout in a variable, in one place, and just reuse that value. By default, the cache tag will try to use the cache called `template_fragments`. If no such cache exists, it will fall back to using the default cache. You may select an alternate cache backend to use with the `using` keyword argument, which must be the last argument to the tag.

```
{% cache 300 local-thing ...  
    using="localcache" %}
```

It is considered an error to specify a cache name that is not configured.

If you want to obtain the cache key used for a cached fragment, you can use

`make_template_fragment_key`. `fragment_name` is the same as second argument to the `cache` template tag; `vary_on` is a list of all additional arguments passed to the tag. This function can be useful for invalidating or overwriting a cached item, for example:

```
>>> from django.core.cache import cache  
>>> from django.core.cache.utils import  
    make_template_fragment_key  
    # cache key for {% cache 500 sidebar  
    username %}  
>>> key =  
    make_template_fragment_key('sidebar',  
    [username])  
>>> cache.delete(key) # invalidates cached
```

template fragment

The low-level cache API

Sometimes, caching an entire rendered page doesn't gain you very much and is, in fact, inconvenient overkill. Perhaps, for instance, your site includes a view whose results depend on several expensive queries, the results of which change at different intervals. In this case, it would not be ideal to use the full-page caching that the per-site or per-view cache strategies offer, because you wouldn't want to cache the entire result (since some of the data changes often), but you'd still want to cache the results that rarely change.

For cases like this, Django exposes a simple, low-level cache API. You can use this API to store objects in the cache with any level of granularity you like. You can cache any Python object that can be pickled safely: strings, dictionaries, lists of model objects, and so forth. (Most common Python objects can be pickled; refer to the Python documentation for more information about pickling.)

Accessing the cache

You can access the caches configured in the `CACHES` setting through a dictionary-like object: `django.core.cache.caches`. Repeated requests for the same alias in the same thread will return the same object.

```
>>> from django.core.cache import caches  
>>> cache1 = caches['myalias']  
>>> cache2 = caches['myalias']  
>>> cache1 is cache2  
True
```

If the named key does not exist, `InvalidCacheBackendError` will be raised. To provide thread-safety, a different instance of the cache backend will be returned for each thread.

As a shortcut, the default cache is available as `django.core.cache.cache`:

```
>>> from django.core.cache import cache
```

This object is equivalent to `caches['default']`.

Basic usage

The basic interface is `set(key, value, timeout)` and `get(key)`:

```
>>> cache.set('my_key', 'hello, world!',  
30)  
>>> cache.get('my_key')  
'hello, world!'
```

The `timeout` argument is optional and defaults to the `timeout` argument of the appropriate backend in the `CACHES` setting (explained above). It's the number of seconds the value should be stored in the cache. Passing in `None` for `timeout` will cache the value

forever. A `timeout` of `0` won't cache the value. If the object doesn't exist in the cache, `cache.get()` returns `None`:

```
# Wait 30 seconds for 'my_key' to
# expire...
>>> cache.get('my_key')
None
```

We advise against storing the literal value `None` in the cache, because you won't be able to distinguish between your stored `None` value and a cache miss signified by a return value of `None`. `cache.get()` can take a `default` argument. This specifies which value to return if the object doesn't exist in the cache:

```
>>> cache.get('my_key', 'has expired')
'has expired'
```

To add a key only if it doesn't already exist, use the `add()` method. It takes the same parameters as `set()`, but it will not attempt to update the cache if the key specified is already present:

```
>>> cache.set('add_key', 'Initial value')
>>> cache.add('add_key', 'New value')
>>> cache.get('add_key')
'Initial value'
```

If you need to know whether `add()` stored a value in the cache, you can check the return value. It will return `True` if the value was stored, `False` otherwise. There's also a

`get_many()` interface that only hits the cache once. `get_many()` returns a dictionary with all the keys you asked for that actually exist in the cache (and haven't expired):

```
>>> cache.set('a', 1)
>>> cache.set('b', 2)
>>> cache.set('c', 3)
>>> cache.get_many(['a', 'b', 'c'])
{'a': 1, 'b': 2, 'c': 3}
```

To set multiple values more efficiently, use `set_many()` to pass a dictionary of key-value pairs:

```
>>> cache.set_many({'a': 1, 'b': 2, 'c':
3})
>>> cache.get_many(['a', 'b', 'c'])
{'a': 1, 'b': 2, 'c': 3}
```

Like `cache.set()`, `set_many()` takes an optional `timeout` parameter. You can delete keys explicitly with `delete()`. This is an easy way of clearing the cache for a particular object:

```
>>> cache.delete('a')
```

If you want to clear a bunch of keys at once, `delete_many()` can take a list of keys to be cleared:

```
>>> cache.delete_many(['a', 'b', 'c'])
```

Finally, if you want to delete all the keys in the cache, use `cache.clear()`. Be careful with this; `clear()` will

remove everything from the cache, not just the keys set by your application.

```
>>> cache.clear()
```

You can also increment or decrement a key that already exists using the `incr()` or `decr()` methods, respectively. By default, the existing cache value will be incremented or decremented by 1. Other increment/decrement values can be specified by providing an argument to the increment/decrement call.

A `ValueError` will be raised if you attempt to increment or decrement a non-existent cache key.:

```
>>> cache.set('num', 1)
>>> cache.incr('num')
2
>>> cache.incr('num', 10)
12
>>> cache.decr('num')
11
>>> cache.decr('num', 5)
6
```

You can close the connection to your cache with `close()` if implemented by the cache backend.

```
>>> cache.close()
```

Note that for caches that don't implement `close` methods `close()` is a no-op.

Cache key prefixing

If you are sharing a cache instance between servers, or between your production and development environments, it's possible for data cached by one server to be used by another server. If the format of cached data is different between servers, this can lead to some very hard to diagnose problems.

To prevent this, Django provides the ability to prefix all cache keys used by a server. When a particular cache key is saved or retrieved, Django will automatically prefix the cache key with the value of the `KEY_PREFIX` cache setting. By ensuring each Django instance has a different `KEY_PREFIX`, you can ensure that there will be no collisions in cache values.

Cache versioning

When you change a running code that uses cached values, you may need to purge any existing cached values. The easiest way to do this is to flush the entire cache, but this can lead to the loss of cache values that are still valid and useful. Django provides a better way to target individual cache values.

Django's cache framework has a system-wide version identifier, specified using the `VERSION` cache setting. The value of this setting is automatically combined with the cache prefix and the user-provided cache key to obtain the final cache key.

By default, any key request will automatically include the site default cache key version. However, the primitive cache functions all include a `version` argument, so you can specify a particular cache key version to set or get.

For example:

```
# Set version 2 of a cache key
>>> cache.set('my_key', 'hello world!',
version=2)
# Get the default version (assuming
version=1)
>>> cache.get('my_key')
None
# Get version 2 of the same key
>>> cache.get('my_key', version=2)
'hello world!'
```

The version of a specific key can be incremented and decremented using the `incr_version()` and `decr_version()` methods. This enables specific keys to be bumped to a new version, leaving other keys unaffected. Continuing our previous example:

```
# Increment the version of 'my_key'
>>> cache.incr_version('my_key')
# The default version still isn't
available
>>> cache.get('my_key')
None
# Version 2 isn't available, either
>>> cache.get('my_key', version=2)
None
# But version 3 is available
>>> cache.get('my_key', version=3)
'hello world!'
```

Cache key transformation

As described in the previous two sections, the cache key provided by a user is not used verbatim—it is combined with the cache prefix and key version to provide a final cache key. By default, the three parts are joined using colons to produce a final string:

```
def make_key(key, key_prefix, version):
    return ':' . join([key_prefix,
                      str(version), key])
```

If you want to combine the parts in different ways, or apply other processing to the final key (for example, taking a hash digest of the key parts), you can provide a custom key function. The `KEY_FUNCTION` cache setting specifies a dotted-path to a function matching the prototype of `make_key()` above. If provided, this custom key function will be used instead of the default key combining function.

Cache key warnings

Memcached, the most commonly-used production cache backend, does not allow cache keys longer than 250 characters or containing whitespace or control characters, and using such keys will cause an exception. To encourage cache-portable code and minimize unpleasant surprises, the other built-in cache backends issue a warning
(`djongo.core.cache.backends.base.CacheKeyW`

`warning`) if a key is used that would cause an error on memcached.

If you are using a production backend that can accept a wider range of keys (a custom backend, or one of the non-memcached built-in backends), and want to use this wider range without warnings, you can silence `CacheKeyWarning` with this code in the `management` module of one of your `INSTALLED_APPS`:

```
import warnings

from django.core.cache import
CacheKeyWarning

warnings.simplefilter("ignore",
CacheKeyWarning)
```

If you want to instead provide custom key validation logic for one of the built-in backends, you can subclass it, override just the `validate_key` method, and follow the instructions for using a custom cache backend.

For instance, to do this for the `locmem` backend, put this code in a module:

```
from django.core.cache.backends.locmem
import LocMemCache

class CustomLocMemCache(LocMemCache):
    def validate_key(self, key):
        # Custom validation, raising
        exceptions or warnings as needed.
        # ...
```

... and use the dotted Python path to this class in the `BACKEND` portion of your `CACHES` setting.

Downstream caches

So far, this chapter has focused on caching your own data. But another type of caching is relevant to web development, too: caching performed by downstream caches. These are systems that cache pages for users even before the request reaches your website. Here are a few examples of downstream caches:

- Your ISP may cache certain pages, so if you requested a page from <http://example.com/>, your ISP would send you the page without having to access example.com directly. The maintainers of example.com have no knowledge of this caching; the ISP sits between example.com and your web browser, handling all of the caching transparently.
- Your Django website may sit behind a *proxy cache*, such as Squid web Proxy Cache (for more information visit <http://www.squid-cache.org/>), that caches pages for performance. In this case, each request first would be handled by the proxy, and it would be passed to your application only if needed.
- Your web browser caches pages, too. If a web page sends out the appropriate headers, your browser will use the local cached copy for subsequent requests to that page, without even contacting the web page again to see whether it has changed.

Downstream caching is a nice efficiency boost, but there's a danger to it: Many web pages' contents differ based on authentication and a host of other variables, and cache systems that blindly save pages based purely on URLs could expose incorrect or sensitive data to subsequent visitors to those pages.

For example, say you operate a web email system, and the contents of the inbox page obviously depend on which user is logged in. If an ISP blindly cached your site, then the first user who logged in through that ISP would have their user-specific inbox page cached for subsequent visitors to the site. That's not cool.

Fortunately, HTTP provides a solution to this problem. A number of HTTP headers exist to instruct downstream caches to differ their cache contents depending on designated variables, and to tell caching mechanisms not to cache particular pages. We'll look at some of these headers in the sections that follow.

Using `vary` headers

The `Vary` header defines which request headers a cache mechanism should take into account when building its cache key. For example, if the contents of a web page depend on a user's language preference, the page is said to vary on language. By default, Django's cache system creates its cache keys using the requested fully-qualified URL—for example, http://www.example.com/stories/2005/?order_by=author.

This means every request to that URL will use the same cached version, regardless of user-agent differences such as cookies or language preferences. However, if this page produces different content based on some difference in request headers—such as a cookie, or a language, or a user-agent—you'll need to use the `Vary` header to tell caching mechanisms that the page output depends on those things.

To do this in Django, use the convenient `django.views.decorators.vary.vary_on_headers()` view decorator, like so:

```
from django.views.decorators.vary import  
vary_on_headers  
  
@vary_on_headers('User-Agent')  
def my_view(request):  
    ""
```

```
# ...
```

In this case, a caching mechanism (such as Django's own cache middleware) will cache a separate version of the page for each unique user-agent. The advantage to using the `vary_on_headers` decorator rather than manually setting the `Vary` header (using something like `response['Vary'] = 'user-agent'`) is that the decorator adds to the `Vary` header (which may already exist), rather than setting it from scratch and potentially overriding anything that was already in there. You can pass multiple headers to `vary_on_headers()`:

```
@vary_on_headers('User-Agent', 'Cookie')
def my_view(request):
    # ...
```

This tells downstream caches to vary on both, which means each combination of user-agent and cookie will get its own cache value. For example, a request with the user-agent `Mozilla` and the cookie value `foo=bar` will be considered different from a request with the user-agent `Mozilla` and the cookie value `foo=ham`. Because varying on cookie is so common, there's a `django.views.decorators.vary.vary_on_cookie()` decorator. These two views are equivalent:

```
@vary_on_cookie
def my_view(request):
    # ...

@vary_on_headers('Cookie')
def my_view(request):
    # ...
```

The headers you pass to `vary_on_headers` are not case sensitive; `User-Agent` is the same thing as `user-agent`. You can also use a helper function, `django.utils.cache.patch_vary_headers()`, directly. This function sets, or adds to, the `Vary` header. For example:

```
from django.utils.cache import
patch_vary_headers

def my_view(request):
    # ...
    response =
    render_to_response('template_name',
    context)
    patch_vary_headers(response,
    ['Cookie'])
    return response
```

`patch_vary_headers` takes an `HttpResponse` instance as its first argument and a list/tuple of case-insensitive header names as its second argument. For more on `Vary` headers, see the official Vary specification (for more information visit <http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.44>).

Controlling cache: using other headers

Other problems with caching are the privacy of data and the question of where data should be stored in a cascade of caches. A user usually faces two kinds of caches: their own browser cache (a private cache) and their provider's cache (a public cache).

A public cache is used by multiple users and controlled by someone else. This poses problems with sensitive data—you don't want, say, your bank account number stored in a public cache. So web applications need a way to tell caches which data is private and which is public.

The solution is to indicate a page's cache should be private. To do this in Django, use the `cache_control` view decorator. Example:

```
from django.views.decorators.cache import  
cache_control  
  
@cache_control(private=True)  
def my_view(request):  
    # ...
```

This decorator takes care of sending out the appropriate HTTP header behind the scenes. Note that the cache control settings `private` and `public` are mutually

exclusive. The decorator ensures that the public directive is removed if private should be set (and vice versa).

An example use of the two directives would be a blog site that offers both private and public entries. Public entries may be cached on any shared cache. The following code uses

`django.utils.cache.patch_cache_control()`, the manual way to modify the cache control header (it is internally called by the `cache_control` decorator):

```
from django.views.decorators.cache import
patch_cache_control
from django.views.decorators.vary import
vary_on_cookie

@vary_on_cookie
def list_blog_entries_view(request):
    if request.user.is_anonymous():
        response =
render_only_public_entries()
        patch_cache_control(response,
public=True)
    else:
        response =
render_private_and_public_entries(request.
user)
        patch_cache_control(response,
private=True)

    return response
```

There are a few other ways to control cache parameters. For example, HTTP allows applications to do the following:

- Define the maximum time a page should be cached.
- Specify whether a cache should always check for newer versions, only delivering the cached content when there are no changes. (Some caches might deliver cached content even if the server page changed, simply because the cache copy isn't yet expired.)

In Django, use the `cache_control` view decorator to specify these cache parameters. In this example, `cache_control` tells caches to revalidate the cache on every access and to store cached versions for, at most, 3,600 seconds:

```
from django.views.decorators.cache import
cache_control

@cache_control(must_revalidate=True,
max_age=3600)
def my_view(request):
    # ...
```

Any valid `Cache-Control` HTTP directive is valid in `cache_control()`. Here's a full list:

- `public=True`
- `private=True`
- `no_cache=True`
- `no_transform=True`
- `must_revalidate=True`
- `proxy_revalidate=True`
- `max_age=num_seconds`
- `s_maxage=num_seconds`

For explanation of Cache-Control HTTP directives, see

the Cache-Control specification (for more information visit <http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.9>). (Note that the caching middleware already sets the cache header's `max-age` with the value of the `CACHE_MIDDLEWARE_SECONDS` setting. If you use a custom `max_age` in a `cache_control` decorator, the decorator will take precedence, and the header values will be merged correctly.)

If you want to use headers to disable caching altogether, `django.views.decorators.cache.never_cache` is a view decorator that adds headers to ensure the response won't be cached by browsers or other caches. Example:

```
from django.views.decorators.cache import
never_cache

@never_cache
def myview(request):
    # ...
```

What's next?

In the next chapter, we will be looking at Django's middleware.

Chapter 17. Django Middleware

Middleware is a framework of hooks into Django's request/response processing. It's a light, low-level plugin system for globally altering Django's input or output.

Each middleware component is responsible for doing some specific function. For example, Django includes a middleware component, [AuthenticationMiddleware](#), that associates users with requests using sessions.

This document explains how middleware works, how you activate middleware, and how to write your own middleware. Django ships with some built-in middleware you can use right out of the box. See *Available middleware* later in this chapter.

Activating middleware

To activate a middleware component, add it to the [MIDDLEWARE_CLASSES](#) list in your Django settings.

In [MIDDLEWARE_CLASSES](#), each middleware component is represented by a string: the full Python path to the middleware's class name. For example, here's the default value created by [django-admin](#)

`startproject:`

```
MIDDLEWARE_CLASSES = [  
  
    'django.contrib.sessions.middleware.SessionMiddleware',  
  
    'django.middleware.common.CommonMiddleware',  
  
    'django.middleware.csrf.CsrfViewMiddleware',  
  
    'django.contrib.auth.middleware.AuthenticationMiddleware',  
  
    'django.contrib.messages.middleware.MessageMiddleware',  
  
    'django.middleware.clickjacking.XFrameOptionsMiddleware',  
]
```

A Django installation doesn't require any middleware- `MIDDLEWARE_CLASSES` can be empty, if you'd like-but it's strongly suggested that you at least use `CommonMiddleware`.

The order in `MIDDLEWARE_CLASSES` matters because a middleware can depend on other middleware. For instance, `AuthenticationMiddleware` stores the authenticated user in the session; therefore, it must run after `SessionMiddleware`. See *Middleware ordering* later in this chapter for some common hints about ordering of Django middleware classes.

Hooks and application order

During the request phase, before calling the view, Django applies middleware in the order it's defined in `MIDDLEWARE_CLASSES`, top-down. Two hooks are available:

- `process_request()`
- `process_view()`

During the response phase, after calling the view, middleware are applied in reverse order, from the bottom up. Three hooks are available:

- `process_exception()`
- `process_template_response()`
- `process_response()`

If you prefer, you can also think of it like an onion: each middleware class is a layer that wraps the view.

The behavior of each hook is described below.

Writing your own middleware

Writing your own middleware is easy. Each middleware component is a single Python class that defines one or more of the following methods:

`process_request`

Method: `process_request(request)`

- `request` is an `HttpRequest` object.
- `process_request()` is called on each request, before Django decides which view to execute.

It should return either `None` or an `HttpResponse` object. If it returns `None`, Django will continue processing this request, executing any other `process_request()` middleware, then, `process_view()` middleware, and finally, the appropriate view.

If it returns an `HttpResponse` object, Django won't bother calling any other request, view or exception middleware, or the appropriate view; it'll apply response middleware to that `HttpResponse`, and return the result.

`process_view`

Method: `process_view(request, view_func, view_args, view_kwargs)`

- `request` is an `HttpRequest` object.
- `view_func` is the Python function that Django is about to use. (It's the actual function object, not the name of the function as a string.)
- `view_args` is a list of positional arguments that will be passed to the view.
- `view_kwargs` is a dictionary of keyword arguments that will be passed to the view.
- Neither `view_args` nor `view_kwargs` include the first view argument (`request`).

`process_view()` is called just before Django calls the view. It should return either `None` or an `HttpResponse` object. If it returns `None`, Django will continue processing this request, executing any other `process_view()` middleware and, then, the appropriate view.

If it returns an `HttpResponse` object, Django won't bother calling any other view or exception middleware, or the appropriate view; it'll apply response middleware to that `HttpResponse`, and return the result.

NOTE

Accessing `request.POST` inside middleware from `process_request` or `process_view` will prevent any view running after the middleware from being able to modify the upload handlers for the request, and should normally be avoided.

The `CsrfViewMiddleware` class can be considered an exception, as it provides the `csrf_exempt()` and `csrf_protect()` decorators which allow views to explicitly control at what point the CSRF validation

should occur.

process_template_response

Method: `process_template_response(request, response)`

- `request` is an `HttpRequest` object.
- `response` is the `TemplateResponse` object (or equivalent) returned by a Django view or by a middleware.

`process_template_response()` is called just after the view has finished executing, if the response instance has a `render()` method, indicating that it is a `TemplateResponse` or equivalent.

It must return a response object that implements a `render` method. It could alter the given `response` by changing `response.template_name` and `response.context_data`, or it could create and return a brand-new `TemplateResponse` or equivalent.

You don't need to explicitly render responses—responses will be automatically rendered once all template response middleware has been called.

Middleware are run in reverse order during the response phase, which includes `process_template_response()`.

process_response

Method: `process_response(request, response)`

- `request` is an `HttpRequest` object.
- `response` is the `HttpResponse` or `StreamingHttpResponse` object returned by a Django view or by a middleware.

`process_response()` is called on all responses before they're returned to the browser. It must return an `HttpResponse` or `StreamingHttpResponse` object. It could alter the given `response`, or it could create and return a brand-new `HttpResponse` or `StreamingHttpResponse`.

Unlike the `process_request()` and `process_view()` methods, the `process_response()` method is always called, even if the `process_request()` and `process_view()` methods of the same middleware class were skipped (because an earlier middleware method returned an `HttpResponse`). In particular, this means that your `process_response()` method cannot rely on setup done in `process_request()`.

Finally, remember that during the response phase, middleware are applied in reverse order, from the bottom up. This means classes defined at the end of `MIDDLEWARE_CLASSES` will be run first.

DEALING WITH STREAMING RESPONSES

Unlike `HttpResponse`, `StreamingHttpResponse`

does not have a `content` attribute. As a result, middleware can no longer assume that all responses will have a `content` attribute. If they need access to the content, they must test for streaming responses and adjust their behavior accordingly:

```
if response.streaming:
    response.streaming_content =
    wrap_streaming_content(response.streaming_
    content)
else:
    response.content =
    alter_content(response.content)
```

`streaming_content` should be assumed to be too large to hold in memory. Response middleware may wrap it in a new generator, but must not consume it. Wrapping is typically implemented as follows:

```
def wrap_streaming_content(content):
    for chunk in content:
        yield alter_content(chunk)
```

process_exception

Method: `process_exception(request, exception)`

- `request` is an `HttpRequest` object.
- `exception` is an `Exception` object raised by the view function.

Django calls `process_exception()` when a view raises an exception. `process_exception()` should

return either `None` or an `HttpResponse` object. If it returns an `HttpResponse` object, the template response and response middleware will be applied, and the resulting response returned to the browser. Otherwise, default exception handling kicks in.

Again, middleware are run in reverse order during the response phase, which includes `process_exception`. If an exception middleware returns a response, the middleware classes above that middleware will not be called at all.

`__init__`

Most middleware classes won't need an initializer since middleware classes are essentially placeholders for the `process_*` methods. If you do need some global state, you may use `__init__` to set up. However, keep in mind a couple of caveats:

1. Django initializes your middleware without any arguments, so you can't define `__init__` as requiring any arguments.
2. Unlike the `process_*` methods which get called once per request, `__init__` gets called only once, when the web server responds to the first request.

MARKING MIDDLEWARE AS UNUSED

It's sometimes useful to determine at run-time whether a piece of middleware should be used. In these cases, your middleware's `__init__` method may raise `django.core.exceptions.MiddlewareNotUsed`. Django will then remove that piece of middleware from

the middleware process and a debug message will be logged to the `django.request` logger when `DEBUG` is set to `True`.

Additional guidelines

- Middleware classes don't have to subclass anything.
- The middleware class can live anywhere on your Python path. All Django cares about is that the `MIDDLEWARE_CLASSES` setting includes the path to it.
- Feel free to look at Django's available middleware for examples.
- If you write a middleware component that you think would be useful to other people, contribute to the community! Let us know and we'll consider adding it to Django.

Available middleware

Cache middleware

`django.middleware.cache.UpdateCacheMiddleware`; and
`django.middleware.cache.FetchFromCacheMiddleware`

Enable the site-wide cache. If these are enabled, each Django-powered page will be cached for as long as the `CACHE_MIDDLEWARE_SECONDS` setting defines. See the cache documentation.

Common middleware

`django.middleware.common.CommonMiddleware`

Adds a few conveniences for perfectionists:

- Forbids access to user agents in the `DISALLOWED_USER_AGENTS` setting, which should be a list of compiled regular expression objects.
- Performs URL rewriting based on the `APPEND_SLASH` and `PREPEND_WWW` settings.
- If `APPEND_SLASH` is `True` and the initial URL doesn't end with a slash, and it is not found in the URLconf, then a new URL is formed by appending a slash at the end. If this new URL is found in the URLconf, then Django redirects the request to this new URL. Otherwise, the initial URL is processed as usual.
- For example, `foo.com/bar` will be redirected to `foo.com/bar/` if

you don't have a valid URL pattern for `foo.com/bar` but do have a valid pattern for `foo.com/bar/`.

- If `PREPEND_WWW` is `True`, URLs that lack a leading `www.` will be redirected to the same URL with a leading `www.`
- Both of these options are meant to normalize URLs. The philosophy is that each URL should exist in one, and only one, place. Technically a URL `foo.com/bar` is distinct from `foo.com/bar/-`-a search engine indexer would treat them as separate URLs-so it's best practice to normalize URLs.
- Handles ETags based on the `USE_ETAGS` setting. If `USE_ETAGS` is set to `True`, Django will calculate an ETag for each request by MD5-hashing the page content, and it'll take care of sending `Not Modified` responses, if appropriate.
- `CommonMiddleware.response_redirect_class`. Defaults to `HttpResponsePermanentRedirect`. Subclass `CommonMiddleware` and override the attribute to customize the redirects issued by the middleware.
- `django.middleware.common.BrokenLinkEmailsMiddleware`. Sends broken link notification emails to `MANAGERS`.

GZip middleware

`django.middleware.gzip.GZipMiddleware`

NOTE

Security researchers recently revealed that when compression techniques (including `GZipMiddleware`) are used on a website, the site becomes exposed to a number of possible attacks. These approaches can be used to compromise, among other things, Django's CSRF protection. Before using `GZipMiddleware` on your site, you should consider very carefully whether you are subject to these attacks. If you're in any doubt about whether you're affected, you should avoid using `GZipMiddleware`. For more details, see [breachattack.com](#).

Compresses content for browsers that understand GZip compression (all modern browsers).

This middleware should be placed before any other middleware that need to read or write the response body so that compression happens afterward.

It will NOT compress content if any of the following are true:

- The content body is less than 200 bytes long.
- The response has already set the [Content-Encoding](#) header.
- The request (the browser) hasn't sent an [Accept-Encoding](#) header containing [gzip](#).

You can apply GZip compression to individual views using the [gzip_page\(\)](#) decorator.

Conditional GET middleware

[django.middleware.http.ConditionalGetMiddleware](#)

Handles conditional GET operations. If the response has a [ETag](#) or [Last-Modified](#) header, and the request has [If-None-Match](#) or [If-Modified-Since](#), the response is replaced by an [HttpResponseNotModified](#).

Also sets the [Date](#) and [Content-Length](#) response-headers.

Locale middleware

[django.middleware.locale.LocaleMiddleware](#)

Enables language selection based on data from the request. It customizes content for each user. See the internationalization documentation.

`LocaleMiddleware.response_redirect_class`

Defaults to `HttpResponseRedirect`. Subclass `LocaleMiddleware` and override the attribute to customize the redirects issued by the middleware.

Message middleware

`django.contrib.messages.middleware.MessageMiddleware`

Enables cookie-and session-based message support. See the messages documentation.

Security middleware

NOTE

If your deployment situation allows, it's usually a good idea to have your front-end web server perform the functionality provided by the `SecurityMiddleware`. That way, if there are requests that aren't served by Django (such as static media or user-uploaded files), they will have the same protections as requests to your Django application.

The `django.middleware.security.SecurityMiddleware` provides several security enhancements to the request/response cycle. The `SecurityMiddleware` achieves this by passing special headers to the browser. Each one can be independently enabled or disabled with

a setting.

HTTP STRICT TRANSPORT SECURITY

Settings:

- `SECURE_HSTS_INCLUDE_SUBDOMAINS`
- `SECURE_HSTS_SECONDS`

For sites that should only be accessed over HTTPS, you can instruct modern browsers to refuse to connect to your domain name via an insecure connection (for a given period of time) by setting the `Strict-Transport-Security` header. This reduces your exposure to some SSL-stripping man-in-the-middle (MITM) attacks.

`SecurityMiddleware` will set this header for you on all HTTPS responses if you set the `SECURE_HSTS_SECONDS` setting to a non-zero integer value.

When enabling HSTS, it's a good idea to first use a small value for testing, for example, `SECURE_HSTS_SECONDS = 3600` for one hour. Each time a web browser sees the HSTS header from your site, it will refuse to communicate non-securely (using HTTP) with your domain for the given period of time.

Once you confirm that all assets are served securely on your site (that is, HSTS didn't break anything), it's a good idea to increase this value so that infrequent visitors will

be protected (31536000 seconds, that is, 1 year, is common).

Additionally, if you set the `SECURE_HSTS_INCLUDE_SUBDOMAINS` setting to `True`, `SecurityMiddleware` will add the `includeSubDomains` tag to the `Strict-Transport-Security` header. This is recommended (assuming all subdomains are served exclusively using HTTPS), otherwise your site may still be vulnerable via an insecure connection to a subdomain.

NOTE

The HSTS policy applies to your entire domain, not just the URL of the response that you set the header on. Therefore, you should only use it if your entire domain is served via HTTPS only.

Browsers properly respecting the HSTS header will refuse to allow users to bypass warnings and connect to a site with an expired, self-signed, or otherwise invalid SSL certificate. If you use HSTS, make sure your certificates are in good shape and stay that way!

X-CONTENT-TYPE-OPTIONS: NOSNIFF

Setting:

- `SECURE_CONTENT_TYPE_NOSNIFF`

Some browsers will try to guess the content types of the assets that they fetch, overriding the `Content-Type` header. While this can help display sites with improperly configured servers, it can also pose a security risk.

If your site serves user-uploaded files, a malicious user could upload a specially-crafted file that would be

interpreted as HTML or Javascript by the browser when you expected it to be something harmless.

To prevent the browser from guessing the content type and force it to always use the type provided in the [Content-Type](#) header, you can pass the [X-Content-Type-Options: nosniff](#) header.

[SecurityMiddleware](#) will do this for all responses if the [SECURE_CONTENT_TYPE_NOSNIFF](#) setting is [True](#).

Note that in most deployment situations where Django isn't involved in serving user-uploaded files, this setting won't help you. For example, if your [MEDIA_URL](#) is served directly by your front-end web server (nginx, Apache, and so on.) then you'd want to set this header there.

On the other hand, if you are using Django to do something like require authorization in order to download files and you cannot set the header using your web server, this setting will be useful.

X-XSS-PROTECTION

Setting:

- [SECURE_BROWSER_XSS_FILTER](#)

Some browsers have the ability to block content that appears to be an XSS attack. They work by looking for Javascript content in the GET or POST parameters of a page. If the Javascript is replayed in the server's

response, the page is blocked from rendering and an error page is shown instead.

The [X-XSS-Protection header](#) is used to control the operation of the XSS filter.

To enable the XSS filter in the browser, and force it to always block suspected XSS attacks, you can pass the [X-XSS-Protection: 1; mode=block](#) header.

[SecurityMiddleware](#) will do this for all responses if the [SECURE_BROWSER_XSS_FILTER](#) setting is [True](#).

NOTE

The browser XSS filter is a useful defense measure, but must not be relied upon exclusively. It cannot detect all XSS attacks and not all browsers support the header. Ensure you are still validating and all input to prevent XSS attacks.

SSL REDIRECT

Settings:

- [SECURE_REDIRECT_EXEMPT](#)
- [SECURE_SSL_HOST](#)
- [SECURE_SSL_REDIRECT](#)

If your site offers both HTTP and HTTPS connections, most users will end up with an unsecured connection by default. For best security, you should redirect all HTTP connections to HTTPS.

If you set the [SECURE_SSL_REDIRECT](#) setting to True, [SecurityMiddleware](#) will permanently (HTTP 301) redirect all HTTP connections to HTTPS.

For performance reasons, it's preferable to do these redirects outside of Django, in a front-end load balancer or reverse-proxy server such as nginx.

`SECURE_SSL_REDIRECT` is intended for the deployment situations where this isn't an option.

If the `SECURE_SSL_HOST` setting has a value, all redirects will be sent to that host instead of the originally-requested host.

If there are a few pages on your site that should be available over HTTP, and not redirected to HTTPS, you can list regular expressions to match those URLs in the `SECURE_REDIRECT_EXEMPT` setting.

If you are deployed behind a load-balancer or reverse-proxy server and Django can't seem to tell when a request actually is already secure, you may need to set the `SECURE_PROXY_SSL_HEADER` setting.

Session middleware

`django.contrib.sessions.middleware.SessionMiddleware`

Enables session support. See [Chapter 15, Django Sessions](#), for more information.

Site middleware

`django.contrib.sites.middleware.CurrentSite`

`siteMiddleware`

Adds the `site` attribute representing the current site to every incoming `HttpRequest` object. See the sites documentation (<https://docs.djangoproject.com/en/1.8/ref/contrib/sites/>) for more information.

Authentication middleware

`django.contrib.auth.middleware` provides three middlewares for use in authentication:

- `*.AuthenticationMiddleware`. Adds the `user` attribute, representing the currently-logged-in user, to every incoming `HttpRequest` object.
- `*.RemoteUserMiddleware`. Middleware for utilizing web server provided authentication.
- `*.SessionAuthenticationMiddleware`. Allows a user's sessions to be invalidated when their password changes. This middleware must appear after `*.AuthenticationMiddleware` in `MIDDLEWARE_CLASSES`.

For more on user authentication in Django, see [Chapter 11, User Authentication in Django](#).

CSRF protection middleware

`django.middleware.csrf.CsrfViewMiddleware`

Adds protection against Cross Site Request Forgeries (CSRF) by adding hidden form fields to POST forms and checking requests for the correct value. See [Chapter 19](#),

Security in Django, for more information on CSRF protection.

X-Frame-options middleware

`django.middleware.clickjacking.XFrameOptionsMiddleware`

Simple clickjacking protection via the X-Frame-Options header.

Middleware ordering

Table 17.1 provides some hints about the ordering of various Django middleware classes:

Class	Notes
UpdateCacheMiddleware	Before those that modify the Vary header (SessionMiddleware , GZipMiddleware , LocaleMiddleware).
GZipMiddleware	Before any middleware that may change or use the response body. After UpdateCacheMiddleware : Modifies Vary header.
ConditionalGetMiddleware	Before CommonMiddleware : uses its Etag header when USE_ETAGS = True .

SessionMiddleware	After UpdateCacheMiddleware : Modifies Vary header.
LocaleMiddleware	One of the topmost, after SessionMiddleware (uses session data) and CacheMiddleware (modifies Vary header).
Common Middleware	Before any middleware that may change the response (it calculates ETags). After GZipMiddleware so it won't calculate an ETag header on gzipped contents. Close to the top: it redirects when APPEND_SLASH or PREPEND_WWW are set to True .
CsrfViewMiddleware	Before any view middleware that assumes that CSRF attacks have been dealt with.
AuthenticationMiddleware	After SessionMiddleware : uses session

SessionMiddleware	storage.
MessageMiddleware	After SessionMiddleware : can use session-based storage.
FetchFromCacheMiddleware	After any middleware that modifies the Vary header: that header is used to pick a value for the cache hash-key.
FlatpageFallbackMiddleware	Should be near the bottom as it's a last-resort type of middleware.
RedirectFallbackMiddleware	Should be near the bottom as it's a last-resort type of middleware.

Table 17.1: Ordering of middleware classes

What's next?

In the next chapter, we will be looking at internationalization in Django.

Chapter 18. Internationalization

When creating message files from JavaScript source code Django was originally developed right in the middle of the United States—quite literally, as Lawrence, Kansas, is less than 40 miles from the geographic center of the continental United States. Like most open source projects, though, Django's community grew to include people from all over the globe. As Django's community became increasingly diverse, *internationalization* and *localization* became increasingly important.

Django itself is fully internationalized; all strings are marked for translation, and settings control the display of locale-dependent values like dates and times. Django also ships with more than 50 different localization files. If you're not a native English speaker, there's a good chance that Django is already translated into your primary language.

The same internationalization framework used for these localizations is available for you to use in your own code and templates.

Because many developers have at best a fuzzy understanding of what internationalization and localization actually mean, we will begin with a few

definitions.

Definitions

Internationalization

Refers to the process of designing programs for the potential use of any locale. This process is usually done by software developers. Internationalization includes marking text (such as UI elements and error messages) for future translation, abstracting the display of dates and times so that different local standards may be observed, providing support for differing time zones, and generally making sure that the code contains no assumptions about the location of its users. You'll often see internationalization abbreviated *I18N*. (The 18 refers to the number of letters omitted between the initial I and the terminal N.)

Localization

Refers to the process of actually translating an internationalized program for use in a particular locale. This work is usually done by translators. You'll sometimes see localization abbreviated as *L10N*.

Here are some other terms that will help us to handle a common language:

LOCALE NAME

A locale name, either a language specification of the form [ll](#) or a combined language and country specification of the form [ll_CC](#). Examples: [it](#), [de_AT](#), [es](#), [pt_BR](#). The language part is always in lower case and the country part in upper case. The separator is an underscore.

LANGUAGE CODE

Represents the name of a language. Browsers send the names of the languages they accept in the [Accept-Language](#) HTTP header using this format. Examples: [it](#), [de-at](#), [es](#), [pt-br](#). Language codes are generally represented in lower-case, but the HTTP [Accept-Language](#) header is case-insensitive. The separator is a dash.

MESSAGE FILE

A message file is a plain-text file, representing a single language, that contains all available translation strings and how they should be represented in the given language. Message files have a [.po](#) file extension.

TRANSLATION STRING

A literal that can be translated.

FORMAT FILE

A format file is a Python module that defines the data formats for a given locale.

Translation

In order to make a Django project translatable, you have to add a minimal number of hooks to your Python code and templates. These hooks are called translation strings. They tell Django: This text should be translated into the end user's language, if a translation for this text is available in that language. It's your responsibility to mark translatable strings; the system can only translate strings it knows about.

Django then provides utilities to extract the translation strings into a message file. This file is a convenient way for translators to provide the equivalent of the translation strings in the target language. Once the translators have filled in the message file, it must be compiled. This process relies on the GNU [gettext](#) toolset.

Once this is done, Django takes care of translating web apps on the fly in each available language, according to users' language preferences.

Essentially, Django does two things:

- It lets developers and template authors specify which parts of their applications should be translatable.
- It uses that information to translate web applications for particular users according to their language preferences.

Django's internationalization hooks are on by default,

and that means there's a bit of i18n-related overhead in certain places of the framework. If you don't use internationalization, you should take the two seconds to set `USE_I18N = False` in your settings file. Then Django will make some optimizations so as not to load the internationalization machinery, which will save you some overhead. There is also an independent but related `USE_L10N` setting that controls if Django should implement format localization.

Internationalization: in Python code

Standard translation

Specify a translation string by using the function `ugettext()`. It's convention to import this as a shorter alias, `_`, to save typing.

Python's standard library `gettext` module installs `_()` into the global namespace, as an alias for `gettext()`. In Django, we have chosen not to follow this practice, for a couple of reasons:

- For international character set (Unicode) support, `ugettext()` is more useful than `gettext()`. Sometimes, you should be using `ugettext_lazy()` as the default translation method for a particular file. Without `_()` in the global namespace, the developer has to think about which is the most appropriate translation function.
- The underscore character (`_`) is used to represent the previous result in Python's interactive shell and doctest tests. Installing a global `_()` function causes interference. Explicitly importing `ugettext()` as `_()` avoids this problem.

In this example, the text "Welcome to my site." is marked as a translation string:

```
from django.utils.translation import
ugettext as
from django.http import HttpResponseRedirect
```

```
def myview(request):
    output = _("Welcome to my site.")
    return HttpResponseRedirect(output)
```

Obviously, you could code this without using the alias.
This example is identical to the previous one:

```
from django.utils.translation import
ugettext
from django.http import HttpResponseRedirect

def my_view(request):
    output = ugettext("Welcome to my
site.")
    return HttpResponseRedirect(output)
```

Translation also works on computed values. This
example is identical to the previous two:

```
def my_view(request):
    words = ['Welcome', 'to', 'my',
'site.']
    output = (' '.join(words))
    return HttpResponseRedirect(output)
```

... and on variables. Again, here's an identical example:

```
def my_view(request):
    sentence = 'Welcome to my site.'
    output = _(sentence)
    return HttpResponseRedirect(output)
```

(The caveat with using variables or computed values, as
in the previous two examples, is that Django's
translation-string-detecting utility, [django-admin](#)
[makemessages](#), won't be able to find these strings.

More on `makemessages` later.)

The strings you pass to `_()` or `ugettext()` can take placeholders, specified with Python's standard named-string interpolation syntax. Example:

```
def my_view(request, m, d):
    output = _('Today is %(month)s % (day)s.')
    return HttpResponseRedirect(output % {'month': m, 'day': d})
```

This technique lets language-specific translations reorder the placeholder text. For example, an English translation may be "*Today is November 26.*", while a Spanish translation may be "*Hoy es 26 de Noviembre.*" - with the month and the day placeholders swapped.

For this reason, you should use named-string interpolation (for example, `%(day)s`) instead of positional interpolation (for example, `%s` or `%d`) whenever you have more than a single parameter. If you used positional interpolation, translations wouldn't be able to reorder placeholder text.

Comments for Translators

If you would like to give translators hints about a translatable string, you can add a comment prefixed with the `Translators` keyword on the line preceding the string, for example:

```
def my_view(request):
    """ Translators: This is a translatable string. """
```

```
# translators: This message appears on  
the home page only  
output = ugettext("Welcome to my  
site.")
```

The comment will then appear in the resulting `.po` file associated with the translatable construct located below it and should also be displayed by most translation tools.

Just for completeness, this is the corresponding fragment of the resulting `.po` file:

```
#. Translators: This message appears on  
the home page only  
# path/to/python/file.py:123  
msgid "Welcome to my site."  
msgstr ""
```

This also works in templates. See `translator-comments-in-templates` for more details.

Marking strings as No-Op

Use the function

`django.utils.translation.ugettext_noop()` to mark a string as a translation string without translating it. The string is later translated from a variable.

Use this if you have constant strings that should be stored in the source language because they are exchanged over systems or users-such as strings in a database-but should be translated at the last possible point in time, such as when the string is presented to the

user.

Pluralization

Use the function

`django.utils.translation.ungettext()` to specify pluralized messages.

`ungettext` takes three arguments: the singular translation string, the plural translation string and the number of objects.

This function is useful when you need your Django application to be localizable to languages where the number and complexity of plural forms is greater than the two forms used in English ('object' for the singular and 'objects' for all the cases where `count` is different from one, irrespective of its value.)

For example:

```
from django.utils.translation import
ungettext
from django.http import HttpResponseRedirect

def hello_world(request, count):
    page = ungettext(
        'there is %(count)d object',
        'there are %(count)d objects',
        count) % {
            'count': count,
        }
    return HttpResponseRedirect(page)
```

In this example the number of objects is passed to the translation languages as the `count` variable.

Note that pluralization is complicated and works differently in each language. Comparing `count` to 1 isn't always the correct rule. This code looks sophisticated, but will produce incorrect results for some languages:

```
from django.utils.translation import
ungettext
from myapp.models import Report

count = Report.objects.count()
if count == 1:
    name = Report._meta.verbose_name
else:
    name =
Report._meta.verbose_name_plural

text = ungettext(
    'There is %(count)d %(name)s
available.',
    'There are %(count)d %(name)s
available.',
    count
) % {
    'count': count,
    'name': name
}
```

Don't try to implement your own singular-or-plural logic, it won't be correct. In a case like this, consider something like the following:

```
text = ungettext(
    'There is %(count)d %(name)s object
available.'
```

```
    'There are %(count)d %(name)s objects
available.',
    count
) % {
    'count': count,
    'name': Report._meta.verbose_name,
}
```

When using `gettext()`, make sure you use a single name for every extrapolated variable included in the literal. In the examples above, note how we used the `name` Python variable in both translation strings. This example, besides being incorrect in some languages as noted above, would fail:

```
text = gettext(
    'There is %(count)d %(name)s
available.',
    'There are %(count)d %(plural_name)s
available.',
    count
) % {
    'count': Report.objects.count(),
    'name': Report._meta.verbose_name,
    'plural_name':
Report._meta.verbose_name_plural
}
```

You would get an error when running `django-admin compilemessages`:

```
a format specification for argument
'name', as in 'msgstr[0]', doesn't exist
in 'msgid'
```

Contextual markers

Sometimes words have several meanings, such as *May* in English, which refers to a month name and to a verb. To enable translators to translate these words correctly in different contexts, you can use the `django.utils.translation.pgettext()` function, or the `django.utils.translation.ngettext()` function if the string needs pluralization. Both take a context string as the first variable.

In the resulting `.po` file, the string will then appear as often as there are different contextual markers for the same string (the context will appear on the `msgctxt` line), allowing the translator to give a different translation for each of them.

For example:

```
from django.utils.translation import  
pgettext  
  
month = pgettext("month name", "May")
```

or:

```
from django.db import models  
from django.utils.translation import  
pgettext_lazy  
  
class MyThing(models.Model):  
    name =  
models.CharField(help_text=pgettext_lazy(  
    'help text for MyThing model').
```

```
'This is the help text'))
```

will appear in the `.po` file as:

```
msgctxt "month name"
msgid "May"
msgstr ""
```

Contextual markers are also supported by the `trans` and `blocktrans` template tags.

Lazy translation

Use the lazy versions of translation functions in `django.utils.translation` (easily recognizable by the `lazy` suffix in their names) to translate strings lazily—when the value is accessed rather than when they're called.

These functions store a lazy reference to the string—not the actual translation. The translation itself will be done when the string is used in a string context, such as in template rendering.

This is essential when calls to these functions are located in code paths that are executed at module load time.

This is something that can easily happen when defining models, forms and model forms, because Django implements these such that their fields are actually class-level attributes. For that reason, make sure to use

lazy translations in the following cases.

MODEL FIELDS AND RELATIONSHIPS

For example, to translate the help text of the `name` field in the following model, do the following:

```
from django.db import models
from django.utils.translation import
ugettext_lazy as

class MyThing(models.Model):
    name =
models.CharField(helptext=_('This is the
help text'))
```

You can mark names of `ForeignKey`, `ManyToManyField` or `OneToOneField` relationship as translatable by using their `verbose_name` options:

```
class MyThing(models.Model):
    kind = models.ForeignKey(ThingKind,
related_name='kinds',
verbose_name=_('kind'))
```

Just like you would do in `verbose_name` you should provide a lowercase verbose name text for the relation as Django will automatically title case it when required.

MODEL VERBOSE NAMES VALUES

It is recommended to always provide explicit `verbose_name` and `verbose_name_plural` options rather than relying on the fall-back English-centric and

somewhat naïve determination of verbose names

Django performs by looking at the model's class name:

```
from django.db import models
from django.utils.translation import
ugettext_lazy as

class MyThing(models.Model):
    name = models.CharField(('name'),
    help_text=_('This is the help text'))

    class Meta:
        verbose_name = ('my thing')
        verbose_name_plural = _('my
things')
```

MODEL METHODS SHORT_DESCRIPTION ATTRIBUTE VALUES

For model methods, you can provide translations to Django and the admin site with the `short_description` attribute:

```
from django.db import models
from django.utils.translation import
ugettext_lazy as

class MyThing(models.Model):
    kind = models.ForeignKey(ThingKind,
    relatedname='kinds',

    verbose_name=_('kind'))

    def is_mouse(self):
        return self.kind.type ==
```

```
MOUSE_TYPE
    is_mouse.short_description = _('Is
it a mouse?')
```

Working with lazy translation objects

The result of a `ugettext_lazy()` call can be used wherever you would use a Unicode string (an object with type `unicode`) in Python. If you try to use it where a bytestring (a `str` object) is expected, things will not work as expected, since a `ugettext_lazy()` object doesn't know how to convert itself to a bytestring. You can't use a Unicode string inside a bytestring, either, so this is consistent with normal Python behavior. For example:

```
# This is fine: putting a unicode proxy
into a unicode string.
"Hello %s" % ugettext_lazy("people")

# This will not work, since you cannot
insert a unicode object
# into a bytestring (nor can you insert
our unicode proxy there)
b"Hello %s" % ugettext_lazy("people")
```

If you ever see output that looks like "`hello <django.utils.functional...>`", you have tried to insert the result of `ugettext_lazy()` into a bytestring. That's a bug in your code.

If you don't like the long `ugettext_lazy` name, you can just alias it as `_` (underscore), like so:

```
from django.db import models
from django.utils.translation import
ugettext_lazy as

class MyThing(models.Model):
    name =
models.CharField(helpText=_('This is the
help text'))
```

Using `ugettext_lazy()` and `ungettext_lazy()` to mark strings in models and utility functions is a common operation. When you're working with these objects elsewhere in your code, you should ensure that you don't accidentally convert them to strings, because they should be converted as late as possible (so that the correct locale is in effect). This necessitates the use of the helper function described next.

LAZY TRANSLATIONS AND PLURAL

When using lazy translation for a plural string (`[u]n[p]gettext_lazy`), you generally don't know the `number` argument at the time of the string definition. Therefore, you are authorized to pass a key name instead of an integer as the `number` argument. Then `number` will be looked up in the dictionary under that key during string interpolation. Here's example:

```
from django import forms
from django.utils.translation import
ugettext_lazy

class MyForm(forms.Form):
    error_message = ungettext_lazy("You
only provided %(num)d
```

```
only provided %d argument",
    "You only provided %d arguments", 'num')

    def clean(self):
        # ...
        if error:
            raise
    forms.ValidationError(self.error_message %
        {'num': number})
```

If the string contains exactly one unnamed placeholder, you can interpolate directly with the `number` argument:

```
class MyForm(forms.Form):
    error_message = ungettext_lazy("You
provided %d argument",
    "You provided %d arguments")

    def clean(self):
        # ...
        if error:
            raise
    forms.ValidationError(self.error_message %
number)
```

JOINING STRINGS: STRING_CONCAT()

Standard Python string joins (`''.join([...])`) will not work on lists containing lazy translation objects. Instead, you can use

`django.utils.translation.string_concat()`, which creates a lazy object that concatenates its contents and converts them to strings only when the result is included in a string. For example:

```
from django.utils.translation import
```

```
string_concat
from django.utils.translation import
ugettext_lazy
#
name = ugettext_lazy('John Lennon')
instrument = ugettext_lazy('guitar')
result = string_concat(name, ': ',
instrument)
```

In this case, the lazy translations in `result` will only be converted to strings when `result` itself is used in a string (usually at template rendering time).

OTHER USES OF LAZY IN DELAYED TRANSLATIONS

For any other case where you would like to delay the translation, but have to pass the translatable string as an argument to another function, you can wrap this function inside a lazy call yourself. For example:

```
from django.utils import six # Python 3
compatibility
from django.utils.functional import lazy
from django.utils.safestring import
mark_safe
from django.utils.translation import
ugettext_lazy as

marksafe_lazy = lazy(mark_safe,
six.text_type)
```

And then later:

```
lazy_string = mark_safe_lazy(_("<p>My
<strong>string!</strong></p>"))
```

Localized names of languages

The `get_language_info()` function provides detailed information about languages:

```
>>> from django.utils.translation import  
get_language_info  
>>> li = get_language_info('de')  
>>> print(li['name'], li['name_local'],  
li['bidi'])  
German Deutsch False
```

The `name` and `name_local` attributes of the dictionary contain the name of the language in English and in the language itself, respectively. The `bidi` attribute is True only for bi-directional languages.

The source of the language information is the `django.conf.locale` module. Similar access to this information is available for template code. See below.

Internationalization: In template code

Translations in Django templates uses two template tags and a slightly different syntax than in Python code. To give your template access to these tags, put

`{% load i18n %}` toward the top of your template. As with all template tags, this tag needs to be loaded in all templates which use translations, even those templates that extend from other templates which have already loaded the `i18n` tag.

trans template tag

The `{% trans %}` template tag translates either a constant string (enclosed in single or double quotes) or variable content:

```
<title>{% trans "This is the title." %}</title>
<title>{% trans myvar %}</title>
```

If the `noop` option is present, variable lookup still takes place but the translation is skipped. This is useful when stubbing out content that will require translation in the future:

```
<title>{% trans "myvar" noop %}</title>
```

Internally, inline translations use an `ugettext()` call.

In case a template variable (`myvar` above) is passed to the tag, the tag will first resolve such variable to a string at run-time and then look up that string in the message catalogs.

It's not possible to mix a template variable inside a string within `{% trans %}`. If your translations require strings with variables (placeholders), use `{% blocktrans %}` instead. If you'd like to retrieve a translated string without displaying it, you can use the following syntax:

```
{% trans "This is the title" as the_title
%}
```

In practice you'll use this to get strings that are used in multiple places or should be used as arguments for other template tags or filters:

```
{% trans "starting point" as start %}
{% trans "end point" as end %}
{% trans "La Grande Boucle" as race %}

<h1>
    <a href="" >{{ race }}</a>
</h1>
<p>
    {% for stage in tour_stages %}
        {% cycle start end %}: {{ stage }}%
    if forloop.counter|divisibleby:2 %}<br >{%
    else %}, {% endif %}
    {% endfor %}
<p>
```

`{% trans %}` also supports contextual markers using the `context` keyword:

```
{% trans "May" context "month name" %}
```

blocktrans template tag

The `blocktrans` tag allows you to mark complex sentences consisting of literals and variable content for translation by making use of placeholders:

```
{% blocktrans %}This string will have {{  
    value }} inside.{% endblocktrans %}
```

To translate a template expression-say, accessing object attributes or using template filters-you need to bind the expression to a local variable for use within the translation block. Examples:

```
{% blocktrans with amount=article.price %}  
That will cost $ {{ amount }}.  
{% endblocktrans %}
```

```
{% blocktrans with myvar=value|filter %}  
This will have {{ myvar }} inside.  
{% endblocktrans %}
```

You can use multiple expressions inside a single `blocktrans` tag:

```
{% blocktrans with book_t=book|title  
author_t=author|title %}  
This is {{ book_t }} by {{ author_t }}  
{% endblocktrans %}
```

The previous more verbose format is still supported: `{% blocktrans with book|title as book_t and author|title as author_t %}`

Other block tags (for example `{% for %}` or `{% if %}`) are not allowed inside a `blocktrans` tag.

If resolving one of the block arguments fails, `blocktrans` will fall back to the default language by deactivating the currently active language temporarily with the `deactivate_all()` function.

This tag also provides for pluralization. To use it:

- Designate and bind a counter value with the name `count`. This value will be the one used to select the right plural form.
- Specify both the singular and plural forms separating them with the `{% plural %}` tag within the `{% blocktrans %}` and `{% endblocktrans %}` tags.

An example:

```
{% blocktrans count counter=list|length %}
There is only one {{ name }} object.
{% plural %}
There are {{ counter }} {{ name }}
objects.
{% endblocktrans %}
```

A more complex example:

```
{% blocktrans with amount=article.price
count years=i.length %}
That will cost $ {{ amount }} per year
```

```
That will cost $ {{ amount }} per year.  
{% plural %}  
That will cost $ {{ amount }} per {{ years }}  
} years.  
{% endblocktrans %}
```

When you use both the pluralization feature and bind values to local variables in addition to the counter value, keep in mind that the `blocktrans` construct is internally converted to an `gettext` call. This means the same notes regarding `gettext` variables apply.

Reverse URL lookups cannot be carried out within the `blocktrans` and should be retrieved (and stored) beforehand:

```
{% url 'path.to.view' arg arg2 as the_url  
%}  
{% blocktrans %}  
This is a URL: {{ the_url }}  
{% endblocktrans %}
```

`{% blocktrans %}` also supports contextual using the `context` keyword:

```
{% blocktrans with name=user.username  
context "greeting" %}  
Hi {{ name }}{% endblocktrans %}
```

Another feature `{% blocktrans %}` supports is the `trimmed` option. This option will remove newline characters from the beginning and the end of the content of the `{% blocktrans %}` tag, replace any whitespace at the beginning and end of a line and merge all lines

into one using a space character to separate them.

This is quite useful for indenting the content of a `{% blocktrans %}` tag without having the indentation characters end up in the corresponding entry in the PO file, which makes the translation process easier.

For instance, the following `{% blocktrans %}` tag:

```
{% blocktrans trimmed %}
    First sentence.
    Second paragraph.
{% endblocktrans %}
```

will result in the entry "`First sentence. Second paragraph.`" in the PO file, compared to "`\n First sentence.\n Second sentence.\n`", if the `trimmed` option had not been specified.

String literals passed to tags and filters

You can translate string literals passed as arguments to tags and filters by using the familiar `_()` syntax:

```
{% some_tag ("Page not found")
value|yesno:("yes,no") %}
```

In this case, both the tag and the filter will see the translated string, so they don't need to be aware of translations.

In this example, the translation infrastructure will be passed the string "[yes](#), [no](#)", not the individual strings "[yes](#)" and "[no](#)". The translated string will need to contain the comma so that the filter parsing code knows how to split up the arguments. For example, a German translator might translate the string "[yes](#), [no](#)" as "[ja](#), [nein](#)" (keeping the comma intact).

Comments for translators in templates

Just like with Python code, these notes for translators can be specified using comments, either with the [comment](#) tag:

```
{% comment %}Translators: View verb{%
endcomment %}
{% trans "View" %}

{% comment %}Translators: Short intro
blurb{% endcomment %}
<p>{% blocktrans %}
    A multiline translatable literal.
    {% endblocktrans %}
</p>
```

or with the [{# ... #}](#) one-line comment constructs:

```
{# Translators: Label of a button that
triggers search #}
<button type="submit">{% trans "Go" %}
</button>

{# Translators: This is a text of the base
button label #}
```

```
template #}
{% blocktrans %}Ambiguous translatable
block of text{% endblocktrans %}
```

Just for completeness, these are the corresponding fragments of the resulting .po file:

```
#. Translators: View verb
# path/to/template/file.html:10
msgid "View"
msgstr ""

#. Translators: Short intro blurb
# path/to/template/file.html:13
msgid ""
"A multiline translatable"
"literal."
msgstr ""

# ...

#. Translators: Label of a button that
triggers search
# path/to/template/file.html:100
msgid "Go"
msgstr ""

#. Translators: This is a text of the base
template
# path/to/template/file.html:103
msgid "Ambiguous translatable block of
text"
msgstr ""
```

Switching language in templates

If you want to select a language within a template, you can use the `language` template tag:

```
{% load i18n %}

{% get_current_language as LANGUAGE_CODE %}

<!-- Current language: {{ LANGUAGE_CODE }} -->
<p>{% trans "Welcome to our page" %}</p>

{% language 'en' %}

    {% get_current_language as LANGUAGE_CODE %}
    <!-- Current language: {{ LANGUAGE_CODE }} -->
    <p>{% trans "Welcome to our page" %}</p>

{% endlanguage %}
```

While the first occurrence of Welcome to our page uses the current language, the second will always be in English.

Other tags

These tags also require a `{% load i18n %}`.

- `{% get_available_languages as LANGUAGES %}` returns a list of tuples in which the first element is the language code and the second is the language name (translated into the currently active locale).
- `{% get_current_language as LANGUAGE_CODE %}` returns the current user's preferred language, as a string. Example: `en-us`. (See *How django discovers language preference* later in this chapter.)
- `{% get_current_language_bidi as LANGUAGE_BIDI %}` returns the current locale's direction. If True, it's a right-to-left language, for example, Hebrew, Arabic. If False it's a left-to-right

language, for example, English, French, German and so on.

If you enable the

`django.template.context_processors.i18n` context processor then each `RequestContext` will have access to `LANGUAGES`, `LANGUAGE_CODE`, and `LANGUAGE_BIDI` as defined above.

The `i18n` context processor is not enabled by default for new projects.

You can also retrieve information about any of the available languages using provided template tags and filters. To get information about a single language, use the `{% get_language_info %}` tag:

```
{% get_language_info for LANGUAGE_CODE as lang %}  
{% get_language_info for "pl" as lang %}
```

You can then access the information:

```
Language code: {{ lang.code }}<br >  
Name of language: {{ lang.name_local }}<br  
>  
Name in English: {{ lang.name }}<br />  
Bi-directional: {{ lang.bidi }}
```

You can also use the `{% get_language_info_list %}` template tag to retrieve information for a list of languages (for example active languages as specified in `LANGUAGES`). See the section about the `set_language` redirect view for an example of how to display a

language selector using `{% get_language_info_list %}`.

In addition to `LANGUAGES` style list of tuples, `{% get_language_info_list %}` supports simple lists of language codes. If you do this in your view:

```
context = {'available_languages': ['en', 'es', 'fr']}
return render(request, 'mytemplate.html', context)
```

you can iterate over those languages in the template:

```
{% get_language_info_list for available_languages as langs %}
{% for lang in langs %} ... {% endfor %}
```

There are also simple filters available for convenience:

- `{{ LANGUAGE_CODE|language_name }}` (German)
- `{{ LANGUAGE_CODE|language_name_local }}` (Deutsch)
- `{{ LANGUAGE_CODE|language bidi }}` (False)

Internationalization: In Javascript code

Adding translations to JavaScript poses some problems:

- JavaScript code doesn't have access to a `gettext` implementation.
- JavaScript code doesn't have access to `.po` or `.mo` files; they need to be delivered by the server.
- The translation catalogs for JavaScript should be kept as small as possible.

Django provides an integrated solution for these problems: It passes the translations into JavaScript, so you can call `gettext`, and so on, from within JavaScript.

The `javascript_catalog` view

The main solution to these problems is the `djongo.views.i18n.javascript_catalog()` view, which sends out a JavaScript code library with functions that mimic the `gettext` interface, plus an array of translation strings.

Those translation strings are taken from applications or Django core, according to what you specify in either the `info_dict` or the URL. Paths listed in `LOCALE_PATHS` are also included.

You hook it up like this:

```
from django.views.i18n import  
    javascript_catalog  
  
js_info_dict = {  
    'packages': ('your.app.package', ),  
}  
  
urlpatterns = [  
    url(r'^jsi18n/$', javascript_catalog,  
        js_info_dict),  
]
```

Each string in `packages` should be in Python dotted-package syntax (the same format as the strings in `INSTALLED_APPS`) and should refer to a package that contains a `locale` directory. If you specify multiple packages, all those catalogs are merged into one catalog. This is useful if you have JavaScript that uses strings from different applications.

The precedence of translations is such that the packages appearing later in the `packages` argument have higher precedence than the ones appearing at the beginning, this is important in the case of clashing translations for the same literal.

By default, the view uses the `djangojs gettext` domain. This can be changed by altering the `domain` argument.

You can make the view dynamic by putting the packages

into the URL pattern:

```
urlpatterns = [
    url(r'^jsi18n/(?P<packages>\S+?)/$',
        javascript_catalog),
]
```

With this, you specify the packages as a list of package names delimited by '+' signs in the URL. This is especially useful if your pages use code from different apps and this changes often and you don't want to pull in one big catalog file. As a security measure, these values can only be either `django.conf` or any package from the `INSTALLED_APPS` setting.

The JavaScript translations found in the paths listed in the `LOCALE_PATHS` setting are also always included. To keep consistency with the translations lookup order algorithm used for Python and templates, the directories listed in `LOCALE_PATHS` have the highest precedence with the ones appearing first having higher precedence than the ones appearing later.

Using the JavaScript translation catalog

To use the catalog, just pull in the dynamically generated script like this:

```
<script type="text/javascript" src="{% url
    'django.views.i18n.javascript_catalog'
%}"></script>
```

This uses reverse URL lookup to find the URL of the JavaScript catalog view. When the catalog is loaded, your JavaScript code can use the standard `gettext` interface to access it:

```
document.write(gettext('this is to be  
translated'));
```

There is also an `ngettext` interface:

```
var object_cnt = 1 // or 0, or 2, or 3,  
...  
s = ngettext('literal for the singular  
case',  
            'literal for the plural case',  
            object_cnt);
```

and even a string interpolation function:

```
function interpolate(fmt, obj, named);
```

The interpolation syntax is borrowed from Python, so the `interpolate` function supports both positional and named interpolation:

- Positional interpolation: `obj` contains a JavaScript array object whose elements values are then sequentially interpolated in their corresponding `fmt` placeholders in the same order they appear. For example:

```
fmcts = ngettext('There is %s object.  
Remaining: %s',  
                 'There are %s objects.  
Remaining: %s', 11);  
s = interpolate(fmcts, [11, 20]);  
// s is 'There are 11 objects.'
```

Remaining: 20'

- Named interpolation: This mode is selected by passing the optional boolean named parameter as true. `obj` contains a JavaScript object or associative array. For example:

```
d = {  
    count: 10,  
    total: 50  
};  
  
fmts = ngettext('Total: %(total)s,  
there is %(count)s  
object',  
'there are %(count)s of a total  
of %(total)s objects',  
d.count);  
s = interpolate(fmts, d, true);
```

You shouldn't go over the top with string interpolation, though: this is still JavaScript, so the code has to make repeated regular-expression substitutions. This isn't as fast as string interpolation in Python, so keep it to those cases where you really need it (for example, in conjunction with `ngettext` to produce proper pluralization).

Note on performance

The `javascript_catalog()` view generates the catalog from `.mo` files on every request. Since its output is constant-at least for a given version of a site-it's a good candidate for caching.

Server-side caching will reduce CPU load. It's easily

implemented with the `cache_page()` decorator. To trigger cache invalidation when your translations change, provide a version-dependent key prefix, as shown in the example below, or map the view at a version-dependent URL.

```
from django.views.decorators.cache import
cache_page
from django.views.i18n import
javascript_catalog

# The value returned by get_version() must
change when translations change.
@cache_page(86400, key_prefix='js18n-%s' %
get_version())
def cached_javascript_catalog(request,
domain='djangojs', packages=None):
    return javascript_catalog(request,
domain, packages)
```

Client-side caching will save bandwidth and make your site load faster. If you're using ETags (`USE_ETAGS = True`), you're already covered. Otherwise, you can apply conditional decorators. In the following example, the cache is invalidated whenever you restart your application server.

```
from django.utils import timezone
from django.views.decorators.http import
last_modified
from django.views.i18n import
javascript_catalog

last_modified_date = timezone.now()

@last_modified(lambda req, **kw:
```

```
last_modified_date)
def cached_javascript_catalog(request,
domain='djangojs', packages=None):
    return javascript_catalog(request,
domain, packages)
```

You can even pre-generate the JavaScript catalog as part of your deployment procedure and serve it as a static file (<http://django-statici18n.readthedocs.org/en/latest/>).

Internationalization: In URL patterns

Django provides two mechanisms to internationalize URL patterns:

- Adding the language prefix to the root of the URL patterns to make it possible for `LocaleMiddleware` to detect the language to activate from the requested URL.
- Making URL patterns themselves translatable via the `django.utils.translation.ugettext_lazy()` function.

Using either one of these features requires that an active language be set for each request; in other words, you need to have

`django.middleware.locale.LocaleMiddleware` in your `MIDDLEWARE_CLASSES` setting.

Language prefix in URL patterns

This function can be used in your root `URLconf` and Django will automatically prepend the current active language code to all URL patterns defined within `i18n_patterns()`. Example URL patterns:

```
from django.conf.urls import include, url
from django.conf.urls.i18n import
    i18n_patterns
from about import views as about_views
from news import views as news_views
from sitemap.views import sitemap
```

```
urlpatterns = [
    url(r'^sitemap\.xml$', sitemap,
name='sitemap_xml'),
]

news_patterns = [
    url(r'^$', news_views.index,
name='index'),
    url(r'^category/(?P<slug>[\w-]+)/$', news_views.category,
name='category'),
    url(r'^(?P<slug>[\w-]+)/$', news_views.details, name='detail'),
]

urlpatterns += i18n_patterns(
    url(r'^about/$', about_views.main,
name='about'),
    url(r'^news/$', include(news_patterns,
namespace='news')),
)
```

After defining these URL patterns, Django will automatically add the language prefix to the URL patterns that were added by the `i18n_patterns` function. Example:

```
from django.core.urlresolvers import
reverse
from django.utils.translation import
activate

>>> activate('en')
>>> reverse('sitemap_xml')
'sitemap.xml'
>>> reverse('news:index')
'en/news/'
```

```
>>> activate('nl')
>>> reverse('news:detail', kwargs={'slug':
'news-slug'})
'nlnews/news-slug/'
```

`i18n_patterns()` is only allowed in your root URLconf. Using it within an included URLconf will throw an `ImproperlyConfigured` exception.

Translating URL patterns

URL patterns can also be marked translatable using the `ugettext_lazy()` function. Example:

```
from django.conf.urls import include, url
from django.conf.urls.i18n import
i18n_patterns
from django.utils.translation import
ugettext_lazy as

from about import views as aboutviews
from news import views as news_views
from sitemaps.views import sitemap

urlpatterns = [
    url(r'^sitemap\.xml$', sitemap,
name='sitemap_xml'),
]

news_patterns = [
    url(r'^$', news_views.index,
name='index'),

url(_(r'^category/(?P<slug>[\w-]+)$'),
news_views.category,
name='category'),
url(r'^(?P<slug>[\w-]+)/$',
```

```
news_views.details, name='detail'),
]

urlpatterns += i18n_patterns(
    url(r'^about/$', about_views.main,
        name='about'),
    url(r'^news/'),
    include(news_patterns, namespace='news')),
)
```

After you've created the translations, the `reverse()` function will return the URL in the active language.

Example:

```
>>> from django.core.urlresolvers import
reverse
>>> from django.utils.translation import
activate

>>> activate('en')
>>> reverse('news:category',
kwargs={'slug': 'recent'})
'ennews/category/recent/'

>>> activate('nl')
>>> reverse('news:category',
kwargs={'slug': 'recent'})
'nlnieuws/categorie/recent/'
```

In most cases, it's best to use translated URLs only within a language-code-prefixed block of patterns (using `i18n_patterns()`), to avoid the possibility that a carelessly translated URL causes a collision with a non-translated URL pattern.

Reversing in templates

If localized URLs get reversed in templates, they always use the current language. To link to a URL in another language use the `language` template tag. It enables the given language in the enclosed template section:

```
{% load i18n %}

{% get_available_languages as languages %}

{% trans "View this category in:" %}
{% for lang_code, lang_name in languages
%}
    {% language lang_code %}
        <a href="{% url 'category'
slug=category.slug %}">{{ lang_name }}</a>
        {% endlanguage %}
    {% endfor %}
```

The `language` tag expects the language code as the only argument.

Localization: How to create language files

Once the string literals of an application have been tagged for later translation, the translation themselves need to be written (or obtained). Here's how that works.

Message files

The first step is to create a message file for a new language. A message file is a plain-text file, representing a single language, that contains all available translation strings and how they should be represented in the given language. Message files have a `.po` file extension.

Django comes with a tool, `django-admin makemessages`, that automates the creation and upkeep of these files.

The `makemessages` command (and `compilemessages` discussed later) use commands from the GNU `gettext` toolset: `xgettext`, `msgfmt`, `msgmerge` and `msguniq`.

The minimum version of the `gettext` utilities supported is 0.15.

To create or update a message file, run this command:

```
django-admin makemessages -l de
```

... where `de` is the locale name for the message file you want to create. For example, `pt_BR` for Brazilian Portuguese, `de_AT` for Austrian German or `id` for Indonesian.

The script should be run from one of two places:

- The root directory of your Django project (the one that contains `manage.py`).
- The root directory of one of your Django apps.

The script runs over your project source tree or your application source tree and pulls out all strings marked for translation (see `how-django-discovers-translations` and be sure `LOCALE_PATHS` is configured correctly). It creates (or updates) a message file in the directory `locale/LANG/LC_MESSAGES`. In the `de` example, the file will be `locale/de/LC_MESSAGES/django.po`.

When you run `makemessages` from the root directory of your project, the extracted strings will be automatically distributed to the proper message files. That is, a string extracted from a file of an app containing a `locale` directory will go in a message file under that directory. A string extracted from a file of an app without any `locale` directory will either go in a message file under the directory listed first in `LOCALE_PATHS` or will generate an error if `LOCALE_PATHS` is empty.

By default, `django-admin makemessages` examines

every file that has the `.html` or `.txt` file extension. In case you want to override that default, use the `-extension` or `-e` option to specify the file extensions to examine:

```
django-admin makemessages -l de -e txt
```

Separate multiple extensions with commas and/or use `-e` or `-extension` multiple times:

```
django-admin makemessages -l de -e  
html,txt -e xml
```

NOTE

When creating message files from JavaScript source code you need to use the special 'djangojs' domain, not `e js`.

If you don't have the `gettext` utilities installed, `makemessages` will create empty files. If that's the case, either install the `gettext` utilities or just copy the English message file (`locale/en/LC_MESSAGES/django.po`) if available and use it as a starting point; it's just an empty translation file.

If you're using Windows and need to install the GNU `gettext` utilities so `makemessages` works, see *gettext on windows* a little later in the chapter for more information.

The format of `.po` files is straightforward. Each `.po` file contains a small bit of metadata, such as the translation

maintainer's contact information, but the bulk of the file is a list of *messages*-simple mappings between translation strings and the actual translated text for the particular language.

For example, if your Django app contained a translation string for the text "`Welcome to my site.`", like so:

```
_("Welcome to my site.")
```

... then `django-admin makemessages` will have created a `.po` file containing the following snippet-a message:

```
#: path/to/python/module.py:23
msgid "Welcome to my site."
msgstr ""
```

A quick explanation:

- `msgid` is the translation string, which appears in the source. Don't change it.
- `msgstr` is where you put the language-specific translation. It starts out empty, so it's your responsibility to change it. Make sure you keep the quotes around your translation.
- As a convenience, each message includes, in the form of a comment line prefixed with `#` and located above the `msgid` line, the filename and line number from which the translation string was gleaned.

Long messages are a special case. There, the first string directly after the `msgstr` (or `msgid`) is an empty string. Then the content itself will be written over the next few lines as one string per line. Those strings are directly

concatenated. Don't forget trailing spaces within the strings; otherwise, they'll be tacked together without whitespace!

Due to the way the [gettext](#) tools work internally and because we want to allow non-ASCII source strings in Django's core and your applications, you must use UTF-8 as the encoding for your PO files (the default when PO files are created). This means that everybody will be using the same encoding, which is important when Django processes the PO files.

To re-examine all source code and templates for new translation strings and update all message files for all languages, run this:

```
django-admin makemessages -a
```

Compiling message files

After you create your message file-and each time you make changes to it-you'll need to compile it into a more efficient form, for use by [gettext](#). Do this with the [django-admin compilemessages](#) utility.

This tool runs over all available [.po](#) files and creates [.mo](#) files, which are binary files optimized for use by [gettext](#). In the same directory from which you ran [django-admin makemessages](#), run:

```
django-admin compilemessages
```

That's it. Your translations are ready for use.

If you're using Windows and need to install the GNU `gettext` utilities so `django-admin compilemessages` works, see `gettext` on Windows below for more information.

Django only supports `.po` files encoded in UTF-8 and without any BOM (Byte Order Mark) so if your text editor adds such marks to the beginning of files by default then you will need to reconfigure it.

Creating message files from JavaScript source code

You create and update the message files the same way as the other Django message files-with the `django-admin makemessages` tool. The only difference is you need to explicitly specify what in `gettext` parlance is known as a domain in this case the `djangajs` domain, by providing a `-d djangajs` parameter, like this:

```
django-admin makemessages -d djangajs -l  
de
```

This would create or update the message file for JavaScript for German. After updating message files, just run `django-admin compilemessages` the same way as you do with normal Django message files.

gettext on windows

This is only needed for people who either want to extract message IDs or compile message files ([.po](#)).

Translation work itself just involves editing existing files of this type, but if you want to create your own message files, or want to test or compile a changed message file, you will need the [gettext](#) utilities:

- Download the following zip files from the GNOME servers (<https://download.gnome.org/binaries/win32/dependencies/>)
 - [gettext-runtime-X.zip](#)
 - [gettext-tools-X.zip](#)

[X](#) is the version number; version [0.15](#) or higher is required.

- Extract the contents of the [bin\](#) directories in both files to the same folder on your system (that is [C:\Program Files\gettext-utils](#))
- Update the system PATH:
 - [Control Panel > System > Advanced > Environment Variables.](#)
 - In the [System variables](#) list, click [Path](#), click [Edit](#).
 - Add [;C:\Program Files\gettext-utils\bin](#) at the end of the [Variable value](#) field.

You may also use [gettext](#) binaries you have obtained elsewhere, so long as the [xgettext -version](#)

command works properly. Do not attempt to use Django translation utilities with a `gettext` package if the command `xgettext -version` entered at a Windows command prompt causes a popup window saying `xgettext.exe has generated errors and will be closed by Windows.`

Customizing the `makemessages` command

If you want to pass additional parameters to `xgettext`, you need to create a custom `makemessages` command and override its `xgettext_options` attribute:

```
from django.core.management.commands
import makemessages

class Command(makemessages.Command):
    xgettext_options =
        makemessages.Command.xgettext_options +
        ['-keyword=mytrans']
```

If you need more flexibility, you could also add a new argument to your custom `makemessages` command:

```
from django.core.management.commands
import makemessages

class Command(makemessages.Command):

    def add_arguments(self, parser):
        super(Command,
        self).add_arguments(parser)
```

```
parser.add_argument('-extra-keyword',
                    dest='xgettext_keywords',
                    action='append')

    def handle(self, args, *options):
        xgettext_keywords =
options.pop('xgettext_keywords')
        if xgettext_keywords:
            self.xgettext_options = (
makemessages.Command.xgettext_options[:] +
                ['-keyword=%s' % kwd for
kwd in xgettext_keywords]
            )
        super(Command, self).handle(args,
*options)
```

Explicitly setting the active language

You may want to set the active language for the current session explicitly. Perhaps a user's language preference is retrieved from another system, for example. You've already been introduced to

`django.utils.translation.activate()`. That applies to the current thread only. To persist the language for the entire session, also modify `LANGUAGE_SESSION_KEY` in the session: from
django.utils import translation user_language = 'fr'
translation.activate(user_language)
request.session[translation.LANGUAGE_SESSION_KEY] = user_language

You would typically want to use both:

`django.utils.translation.activate()` will change the language for this thread, and modifying the session makes this preference persist in future requests.

If you are not using sessions, the language will persist in a cookie, whose name is configured in

`LANGUAGE_COOKIE_NAME`. For example: from
django.utils import translation from django import http
from django.conf import settings user_language = 'fr'
translation.activate(user_language) response =
http.HttpResponse(...)

```
response.set_cookie(settings.LANGUAGE_COOKIE_NAME, user_language)
```

Using translations outside views and templates

While Django provides a rich set of internationalization tools for use in views and templates, it does not restrict the usage to Django-specific code. The Django translation mechanisms can be used to translate arbitrary texts to any language that is supported by Django (as long as an appropriate translation catalog exists, of course).

You can load a translation catalog, activate it and translate text to language of your choice, but remember to switch back to original language, as activating a translation catalog is done on per-thread basis and such change will affect code running in the same thread.

For example:

```
from django.utils import translation
def welcome_translated(language):
    cur_language =
        translation.get_language()
    try:
        translation.activate(language)
        text =
            translation.ugettext('welcome')
    finally:
        translation.activate(cur_language)
    return text
```

Calling this function with the value 'de' will give you "Willkommen", regardless of `LANGUAGE_CODE` and language set by middleware.

Functions of particular interest are `django.utils.translation.get_language()` which returns the language used in the current thread, `django.utils.translation.activate()` which activates a translation catalog for the current thread, and `django.utils.translation.check_for_languag e()` which checks if the given language is supported by Django.

Implementation notes

Specialties of Django translation

Django's translation machinery uses the standard `gettext` module that comes with Python. If you know `gettext`, you might note these specialties in the way Django does translation:

- The string domain is `django` or `djangojs`. This string domain is used to differentiate between different programs that store their data in a common message-file library (usually `usrshare/locale/`). The `django` domain is used for python and template translation strings and is loaded into the global translation catalogs. The `djangojs` domain is only used for JavaScript translation catalogs to make sure that those are as small as possible.
- Django doesn't use `xgettext` alone. It uses Python wrappers around `xgettext` and `msgfmt`. This is mostly for convenience.

How Django discovers language preference

Once you've prepared your translations-or, if you just want to use the translations that come with Django-you'll need to activate translation for your app.

Behind the scenes, Django has a very flexible model of deciding which language should be used-installation-wide, for a particular user, or both.

To set an installation-wide language preference, set `LANGUAGE_CODE`. Django uses this language as the default translation—the final attempt if no better matching translation is found through one of the methods employed by the locale middleware (see below).

If all you want is to run Django with your native language all you need to do is set `LANGUAGE_CODE` and make sure the corresponding message files and their compiled versions (`.mo`) exist.

If you want to let each individual user specify which language they prefer, then you also need to use the `LocaleMiddleware`. `LocaleMiddleware` enables language selection based on data from the request. It customizes content for each user.

To use `LocaleMiddleware`, add
`'django.middleware.locale.LocaleMiddleware'`
to your `MIDDLEWARE_CLASSES` setting. Because middleware order matters, you should follow these guidelines:

- Make sure it's one of the first middlewares installed.
- It should come after `SessionMiddleware`, because `LocaleMiddleware` makes use of session data. And it should come before `CommonMiddleware` because `CommonMiddleware` needs an activated language in order to resolve the requested URL.
- If you use `CacheMiddleware`, put `LocaleMiddleware` after it.

For example, your `MIDDLEWARE_CLASSES` might look like this:

```
MIDDLEWARE_CLASSES = [  
  
    'django.contrib.sessions.middleware.SessionMiddleware',  
  
    'django.middleware.locale.LocaleMiddleware',  
  
    'django.middleware.common.CommonMiddleware'  
,  
]  

```

For more on middleware, see [Chapter 17, Django Middleware](#).

[LocaleMiddleware](#) tries to determine the user's language preference by following this algorithm:

- First, it looks for the language prefix in the requested URL. This is only performed when you are using the [i18n_patterns](#) function in your root URLconf. See [internationalization](#) for more information about the language prefix and how to internationalize URL patterns.
- Failing that, it looks for the `LANGUAGE_SESSION_KEY` key in the current user's session.
- Failing that, it looks for a cookie. The name of the cookie used is set by the `LANGUAGE_COOKIE_NAME` setting. (The default name is `django_language`.)
- Failing that, it looks at the `Accept-Language` HTTP header. This header is sent by your browser and tells the server which language(s) you prefer, in order by priority. Django tries each language in the header until it finds one with available translations.
- * Failing that, it uses the global `LANGUAGE_CODE` setting.

Notes:

- In each of these places, the language preference is expected to be in

the standard language format, as a string. For example, Brazilian Portuguese is `pt-br`.

- If a base language is available but the sublanguage specified is not, Django uses the base language. For example, if a user specifies `de-at` (Austrian German) but Django only has `de` available, Django uses `de`.
- Only languages listed in the `LANGUAGES` setting can be selected. If you want to restrict the language selection to a subset of provided languages (because your application doesn't provide all those languages), set `LANGUAGES` to a list of languages. For example:

```
LANGUAGES = [
    ('de', ('German')),
    ('en', ('English')),
]
```

This example restricts languages that are available for automatic selection to German and English (and any sublanguage, like `de-ch` or `en-us`).

- If you define a custom `LANGUAGES` setting, as explained in the previous bullet, you can mark the language names as translation strings-but use `ugettext_lazy()` instead of `ugettext()` to avoid a circular import.

Here's a sample settings file:

```
from django.utils.translation import
ugettext_lazy as

LANGUAGES = [
    ('de', ('German')),
    ('en', _('English')),
]
```

Once `LocaleMiddleware` determines the user's preference, it makes this preference available as

`request.LANGUAGE_CODE` for each `HttpRequest`.

Feel free to read this value in your view code. Here's a simple example:

```
from django.http import HttpResponse

def hello_world(request, count):
    if request.LANGUAGE_CODE == 'de-at':
        return HttpResponse("You prefer to
read Austrian German.")
    else:
        return HttpResponse("You prefer to
read another language.")
```

Note that, with static (middleware-less) translation, the language is in `settings.LANGUAGE_CODE`, while with dynamic (middleware) translation, it's in `request.LANGUAGE_CODE`.

How Django discovers translations

At runtime, Django builds an in-memory unified catalog of literal translations. To achieve this, it looks for translations by following this algorithm regarding the order in which it examines the different file paths to load the compiled message files (`.mo`) and the precedence of multiple translations for the same literal:

- The directories listed in `LOCALE_PATHS` have the highest precedence, with the ones appearing first having higher precedence than the ones appearing later.
- Then, it looks for and uses if it exists a `locale` directory in each of the installed apps listed in `INSTALLED_APPS`. The ones appearing first have higher precedence than the ones appearing later.

- Finally, the Django-provided base translation in `django/conf/locale` is used as a fallback.

In all cases, the name of the directory containing the translation is expected to be named using locale name notation. For example, `de`, `pt_BR`, `es_AR`, and so on.

This way, you can write applications that include their own translations, and you can override base translations in your project. Or, you can just build a big project out of several apps and put all translations into one big common message file specific to the project you are composing. The choice is yours.

All message file repositories are structured the same way. They are:

- All paths listed in `LOCALE_PATHS` in your settings file are searched for `<language>/LC_MESSAGES/django.(po|mo)`
- `$APPPATH/locale/<language>/LC_MESSAGES/django.(po|mo)`
- `$PYTHONPATH/django/conf/locale/<language>/LC_MESSAGES/django.(po|mo)`.

To create message files, you use the `django-admin makemessages` tool. And you use `django-admin compilemessages` to produce the binary `.mo` files that are used by `gettext`.

You can also run `django-admin compilemessages` to make the compiler process all the directories in your `LOCALE_PATHS` setting.

What's next?

In the next chapter, we will be looking at security in Django.

Chapter 19. Security in Django

Ensuring that the sites you build are secure is of the utmost importance to a professional web applications developer.

The Django framework is now very mature and the majority of common security issues are addressed in some way by the framework itself, however no security measure is 100% guaranteed and there are new threats emerging all the time, so it's up to you as a web developer to ensure that your websites and applications are secure.

web security is too large a subject to cover in depth in a single book chapter. This chapter includes an overview of Django's security features and advice on securing a Django-powered site that will protect your sites 99% of the time, but it's up to you to keep abreast of changes in web security.

For more detailed information on web security, Django's archive of security issues([for more information visit https://docs.djangoproject.com/en/1.8/releases/security/](https://docs.djangoproject.com/en/1.8/releases/security/)) is a good place to start, along with Wikipedia's web application security page (https://en.wikipedia.org/wiki/Web_application_security).

Django's built in security features

Cross Site Scripting (XSS) protection

Cross Site Scripting (XSS) attacks allow a user to inject client side scripts into the browsers of other users.

This is usually achieved by storing the malicious scripts in the database where it will be retrieved and displayed to other users, or by getting users to click a link which will cause the attacker's JavaScript to be executed by the user's browser. However, XSS attacks can originate from any untrusted source of data, such as cookies or web services, whenever the data is not sufficiently sanitized before including in a page.

Using Django templates protects you against the majority of XSS attacks. However, it is important to understand what protections it provides and its limitations.

Django templates escape specific characters which are particularly dangerous to HTML. While this protects users from most malicious input, it is not entirely foolproof. For example, it will not protect the following:

```
<style class="{{ var }}>...</style>
```

If `var` is set to '`class1 onmouseover=javascript:func()`', this can result in unauthorized JavaScript execution, depending on how the browser renders imperfect HTML. (Quoting the attribute value would fix this case).

It is also important to be particularly careful when using `is_safe` with custom template tags, the `safe` template tag, `mark_safe`, and when `autoescape` is turned off.

In addition, if you are using the template system to output something other than HTML, there may be entirely separate characters and words which require escaping.

You should also be very careful when storing HTML in the database, especially when that HTML is retrieved and displayed.

Cross Site Request Forgery (CSRF) protection

Cross Site Request Forgery (CSRF) attacks allow a malicious user to execute actions using the credentials of another user without that user's knowledge or consent.

Django has built-in protection against most types of CSRF attacks, providing you have enabled and used it where appropriate. However, as with any mitigation technique, there are limitations.

For example, it is possible to disable the CSRF module globally or for particular views. You should only do this if you know what you are doing. There are other limitations if your site has subdomains that are outside of your control.

CSRF protection works by checking for a nonce in each [POST](#) request. This ensures that a malicious user cannot simply replay a form [POST](#) to your website and have another logged in user unwittingly submit that form. The malicious user would have to know the nonce, which is user specific (using a cookie).

When deployed with HTTPS, [CsrfViewMiddleware](#) will check that the HTTP referrer header is set to a URL on the same origin (including subdomain and port). Because HTTPS provides additional security, it is imperative to ensure connections use HTTPS where it is available by forwarding insecure connection requests and using HSTS for supported browsers.

Be very careful with marking views with the [csrf_exempt](#) decorator unless it is absolutely necessary.

Django's CSRF middleware and template tag provides easy-to-use protection against Cross Site Request Forgeries.

The first defense against CSRF attacks is to ensure that [GET](#) requests (and other 'safe' methods, as defined by

9.1.1 Safe Methods, HTTP 1.1, RFC 2616 (for more information visit <https://tools.ietf.org/html/rfc2616.html#section-9.1.1>) are side-effect free. Requests via 'unsafe' methods, such as **POST**, **PUT** and **DELETE**, can then be protected by following the steps below.

HOW TO USE IT

To take advantage of CSRF protection in your views, follow these steps:

1. The CSRF middleware is activated by default in the `MIDDLEWARE_CLASSES` setting. If you override that setting, remember that '`django.middleware.csrf.CsrfViewMiddleware`' should come before any view middleware that assume that CSRF attacks have been dealt with.
2. If you disabled it, which is not recommended, you can use `csrf_protect()` on particular views you want to protect (see below).
3. In any template that uses a **POST** form, use the `csrf_token` tag inside the `<form>` element if the form is for an internal URL, for example:

```
<form action"." method="post">{%
  csrf_token %}
```

4. This should not be done for **POST** forms that target external URLs, since that would cause the CSRF token to be leaked, leading to a vulnerability.
5. In the corresponding view functions, ensure that the '`django.template.context_processors.csrf`' context processor is being used. Usually, this can be done in one of two ways:
6. Use `RequestContext`, which always uses '`django.template.context_processors.csrf`' (no matter

what template context processors are configured in the [TEMPLATES](#) setting). If you are using generic views or contrib apps, you are covered already, since these apps use [RequestContext](#) throughout.

7. Manually import and use the processor to generate the CSRF token and add it to the template context. for example:

```
from django.shortcuts import
render_to_response
from
django.template.context_processors
import csrf

def my_view(request):
    c = {}
    c.update(csrf(request))
    # ... view code here
    return
render_to_response("a_template.html",
c)
```

8. You may want to write your own [render_to_response\(\)](#) wrapper that takes care of this step for you.

AJAX

While the above method can be used for AJAX POST requests, it has some inconveniences: you have to remember to pass the CSRF token in as POST data with every POST request. For this reason, there is an alternative method: on each [XMLHttpRequest](#), set a custom [X-CSRFToken](#) header to the value of the CSRF token. This is often easier, because many JavaScript frameworks provide hooks that allow headers to be set on every request.

As a first step, you must get the CSRF token itself. The

recommended source for the token is the `csrfmiddlewaretoken` cookie, which will be set if you've enabled CSRF protection for your views as outlined above.

The CSRF token cookie is named `csrfmiddlewaretoken` by default, but you can control the cookie name via the `CSRF_COOKIE_NAME` setting.

Acquiring the token is straightforward:

```
// using jQuery
function getCookie(name) {
    var cookieValue = null;
    if (document.cookie && document.cookie
!= '') {
        var cookies =
document.cookie.split(';");
        for (var i = 0; i <
cookies.length; i++) {
            var cookie =
jQuery.trim(cookies[i]);
            // Does this cookie string
begin with the name we want?
            if (cookie.substring(0,
name.length + 1) == (name + '=')) {
                cookieValue =
decodeURIComponent(cookie.substring(name.l
ength + 1));
                break;
            }
        }
    }
    return cookieValue;
}
var csrfmiddlewaretoken = getCookie('csrfmiddlewaretoken');
```

The above code could be simplified by using the jQuery

cookie plugin (<http://plugins.jquery.com/cookie/>) to replace `getCookie`:

```
var csrftoken = $.cookie('csrftoken');
```

NOTE

The CSRF token is also present in the DOM, but only if explicitly included using `csrf_token` in a template. The cookie contains the canonical token; the `CsrfViewMiddleware` will prefer the cookie to the token in the DOM. Regardless, you're guaranteed to have the cookie if the token is present in the DOM, so you should use the cookie!

NOTE

If your view is not rendering a template containing the `csrf_token` template tag, Django might not set the CSRF token cookie. This is common in cases where forms are dynamically added to the page. To address this case, Django provides a view decorator which forces setting of the cookie: `ensure_csrf_cookie()`.

Finally, you'll have to actually set the header on your AJAX request, while protecting the CSRF token from being sent to other domains using `settings.crossDomain` in jQuery 1.5.1 and newer:

```
function csrfSafeMethod(method) {
    // these HTTP methods do not require
    CSRF protection
    return
    (/^(GET|HEAD|OPTIONS|TRACE)$/.test(method)
);
}
$.ajaxSetup({
    beforeSend: function(xhr, settings) {
        if (!csrfSafeMethod(settings.type)
&& !this.crossDomain) {
            xhr.setRequestHeader("X-
CSRFToken", csrftoken);
        }
    }
});
```

```
});
```

OTHER TEMPLATE ENGINES

When using a different template engine than Django's built-in engine, you can set the token in your forms manually after making sure it's available in the template context.

For example, in the Jinja2 template language, your form could contain the following:

```
<div style="display:none">
    <input type="hidden"
name="csrfmiddlewaretoken" value="{{
csrf_token }}">
</div>
```

You can use JavaScript similar to the AJAX code above to get the value of the CSRF token.

THE DECORATOR METHOD

Rather than adding [CsrfViewMiddleware](#) as a blanket protection, you can use the [csrf_protect](#) decorator, which has exactly the same functionality, on particular views that need the protection. It must be used both on views that insert the CSRF token in the output, and on those that accept the [POST](#) form data. (These are often the same view function, but not always).

Use of the decorator by itself is not recommended, since if you forget to use it, you will have a security hole. The

belt and braces strategy of using both is fine, and will incur minimal overhead.

`django.views.decorators.csrf.csrf_protect(view)`

Decorator that provides the protection of [CsrfViewMiddleware](#) to a view.

Usage:

```
from django.views.decorators.csrf import
    csrf_protect
from django.shortcuts import render

@csrf_protect
def my_view(request):
    c = {}
    # ...
    return render(request,
        "a_template.html", c)
```

If you are using class-based views, you can refer to [Decorating class-based views](#).

REJECTED REQUESTS

By default, a *403 Forbidden* response is sent to the user if an incoming request fails the checks performed by [CsrfViewMiddleware](#). This should usually only be seen when there is a genuine Cross Site Request Forgery, or when, due to a programming error, the CSRF token has not been included with a [POST](#) form.

The error page, however, is not very friendly, so you may want to provide your own view for handling this condition. To do this, simply set the [CSRF_FAILURE_VIEW](#) setting.

How it works

The CSRF protection is based on the following things:

- A CSRF cookie that is set to a random value (a session independent nonce, as it is called), which other sites will not have access to.
- This cookie is set by [CsrfViewMiddleware](#). It is meant to be permanent, but since there is no way to set a cookie that never expires, it is sent with every response that has called `django.middleware.csrf.get_token()` (the function used internally to retrieve the CSRF token).
- A hidden form field with the name `csrfmiddlewaretoken` present in all outgoing POST forms. The value of this field is the value of the CSRF cookie.
- This part is done by the template tag.
- For all incoming requests that are not using HTTP [GET](#), [HEAD](#), [OPTIONS](#), or [TRACE](#), a CSRF cookie must be present, and the `csrfmiddlewaretoken` field must be present and correct. If it isn't, the user will get a 403 error.
- This check is done by [CsrfViewMiddleware](#).
- In addition, for HTTPS requests, strict referrer checking is done by [CsrfViewMiddleware](#). This is necessary to address a Man-In-The-Middle attack that is possible under HTTPS when using a session independent nonce, due to the fact that HTTP 'Set-Cookie' headers are (unfortunately) accepted by clients that are talking to a site under HTTPS. (Referer checking is not done for HTTP requests because the presence of the Referer header is not reliable enough under HTTP.)

This ensures that only forms that have originated from

your website can be used to [POST](#) data back.

It deliberately ignores [GET](#) requests (and other requests that are defined as 'safe' by RFC 2616). These requests ought never to have any potentially dangerous side effects, and so a CSRF attack with a [GET](#) request ought to be harmless. RFC 2616 defines [POST](#), [PUT](#), and [DELETE](#) as 'unsafe', and all other methods are assumed to be unsafe, for maximum protection.

CACHING

If the `csrf_token` template tag is used by a template (or the `get_token` function is called some other way), `CsrfViewMiddleware` will add a cookie and a `Vary: Cookie` header to the response. This means that the middleware will play well with the cache middleware if it is used as instructed (`UpdateCacheMiddleware` goes before all other middleware).

However, if you use cache decorators on individual views, the CSRF middleware will not yet have been able to set the `Vary` header or the CSRF cookie, and the response will be cached without either one.

In this case, on any views that will require a CSRF token to be inserted you should use the `djongo.views.decorators.csrf.csrf_protect()` decorator first:

```
from djongo.views.decorators.cache import  
cache_page
```

```
from django.views.decorators.csrf import  
    csrf_protect  
  
    @cache_page(60 * 15)  
    @csrf_protect  
    def my_view(request):  
        ...
```

If you are using class-based views, you can refer to decorating class-based views in the Django documentation

(<https://docs.djangoproject.com/en/1.8/topics/class-based-views/intro/#decorating-class-based-views>).

TESTING

The [CsrfViewMiddleware](#) will usually be a big hindrance to testing view functions, due to the need for the CSRF token which must be sent with every [POST](#) request. For this reason, Django's HTTP client for tests has been modified to set a flag on requests which relaxes the middleware and the [csrf_protect](#) decorator so that they no longer rejects requests. In every other respect (for example, sending cookies and so on), they behave the same.

If, for some reason, you want the test client to perform CSRF checks, you can create an instance of the test client that enforces CSRF checks:

```
>>> from django.test import Client  
>>> csrf_client =  
    Client(enforce_csrf_checks=True)
```

LIMITATIONS

Subdomains within a site will be able to set cookies on the client for the whole domain. By setting the cookie and using a corresponding token, subdomains will be able to circumvent the CSRF protection. The only way to avoid this is to ensure that subdomains are controlled by trusted users (or, at least unable to set cookies).

Note that even without CSRF, there are other vulnerabilities, such as session fixation, that make giving subdomains to untrusted parties a bad idea, and these vulnerabilities cannot easily be fixed with current browsers.

EDGE CASES

Certain views can have unusual requirements that mean they don't fit the normal pattern envisaged here. A number of utilities can be useful in these situations. The scenarios they might be needed in are described in the following section.

UTILITIES

The examples below assume you are using function-based views. If you are working with class-based views, you can refer to decorating class-based views in the Django documentation.

`django.views.decorators.csrf.csrf_exempt(view)`

Most views require CSRF protection, but a few do not.

Rather than disabling the middleware and applying `csrf_protect` to all the views that need it, enable the middleware and use `csrf_exempt()`.

This decorator marks a view as being exempt from the protection ensured by the middleware. Example:

```
from django.views.decorators.csrf import  
    csrf_exempt  
  
from django.http import HttpResponseRedirect  
  
@csrf_exempt  
def my_view(request):  
    return HttpResponseRedirect('Hello world')
```

django.views.decorators.csrf.requires_csrf_token(view)

There are cases when

`CsrfViewMiddleware.process_view` may not have run before your view is run-404 and 500 handlers, for example-but you still need the CSRF token in a form.

Normally the `csrf_token` template tag will not work if `CsrfViewMiddleware.process_view` or an equivalent like `csrf_protect` has not run. The view decorator `requires_csrf_token` can be used to ensure the template tag does work. This decorator works similarly to `csrf_protect`, but never rejects an incoming request.

Example:

```
from django.views.decorators.csrf import  
    requires_csrf_token
```

```
from django.shortcuts import render

@requires_csrf_token
def my_view(request):
    c = {}
    # ...
    return render(request,
    "a_template.html", c)
```

There may also be some views that are unprotected and have been exempted by `csrf_exempt`, but still need to include the CSRF token. In these cases, use `csrf_exempt()` followed by `requires_csrf_token()`. (that is, `requires_csrf_token` should be the innermost decorator).

A final example is where a view needs CSRF protection under one set of conditions only, and mustn't have it for the rest of the time. A solution is to use `csrf_exempt()` for the whole view function, and `csrf_protect()` for the path within it that needs protection.

For example:

```
from django.views.decorators.csrf import
csrf_exempt, csrf_protect

@csrf_exempt
def my_view(request):

    @csrf_protect
    def protected_path(request):
        do_something()
```

```
if some_condition():
    return protected_path(request)
else:
    do_something_else()
```

django.views.decorators.csrf.ensure_csrf_cookie(view)

This decorator forces a view to send the CSRF cookie. A scenario where this would be used is if a page makes a POST request via AJAX, and the page does not have an HTML form with a `csrf_token` that would cause the required CSRF cookie to be sent. The solution would be to use `ensure_csrf_cookie()` on the view that sends the page.

CONTRIB AND REUSABLE APPS

Because it is possible for the developer to turn off the `CsrfViewMiddleware`, all relevant views in contrib apps use the `csrf_protect` decorator to ensure the security of these applications against CSRF. It is recommended that the developers of other reusable apps that want the same guarantees also use the `csrf_protect` decorator on their views.

CSRF SETTINGS

A number of settings can be used to control Django's CSRF behavior:

- `CSRF_COOKIE_AGE`
- `CSRF_COOKIE_DOMAIN`
- `CSRF_COOKIE_HTTPONLY`

- `CSRF_COOKIE_NAME`
- `CSRF_COOKIE_PATH`
- `CSRF_COOKIE_SECURE`
- `CSRF_FAILURE_VIEW`

See [Appendix D, *Settings*](#), for more information on each of these settings.

SOL injection protection

SQL injection is a type of attack where a malicious user is able to execute arbitrary SQL code on a database. This can result in records being deleted or data leakage.

By using Django's querysets, the resulting SQL will be properly escaped by the underlying database driver. However, Django also gives developers power to write raw queries or execute custom SQL. These capabilities should be used sparingly and you should always be careful to properly escape any parameters that the user can control. In addition, you should exercise caution when using `extra()`.

Clickjacking protection

Clickjacking is a type of attack where a malicious site wraps another site in a frame. This type of attack occurs when a malicious site tricks a user into clicking on a concealed element of another site which they have loaded in a hidden frame or iframe.

Django contains clickjacking protection in the form of the [X-Frame-Options middleware](#) which in a supporting browser can prevent a site from being rendered inside a frame. It is possible to disable the protection on a per view basis or to configure the exact header value sent.

The middleware is strongly recommended for any site that does not need to have its pages wrapped in a frame by third party sites, or only needs to allow that for a small section of the site.

AN EXAMPLE OF CLICKJACKING

Suppose an online store has a page where a logged in user can click **Buy Now** to purchase an item. A user has chosen to stay logged into the store all the time for convenience. An attacker site might create an **I Like Ponies** button on one of their own pages, and load the store's page in a transparent [iframe](#) such that the **Buy Now** button is invisibly overlaid on the **I Like Ponies** button. If the user visits the attacker's site, clicking **I Like Ponies** will cause an inadvertent click on the **Buy Now** button and an unknowing purchase of the item.

PREVENTING CLICKJACKING

Modern browsers honor the X-Frame-Options (for more information visit https://developer.mozilla.org/en/The_X-FRAME-OPTIONS_response_header) HTTP header that indicates whether or not a resource is allowed to load within a frame or iframe. If the response contains the

header with a value of `SAMEORIGIN` then the browser will only load the resource in a frame if the request originated from the same site. If the header is set to `DENY` then the browser will block the resource from loading in a frame no matter which site made the request.

Django provides a few simple ways to include this header in responses from your site:

- A simple middleware that sets the header in all responses.
- A set of view decorators that can be used to override the middleware or to only set the header for certain views.

HOW TO USE IT

Setting X-Frame-Options for all responses

To set the same `X-Frame-Options` value for all responses in your site, put

`'django.middleware.clickjacking.XFrameOptionsMiddleware'` to `MIDDLEWARE_CLASSES`:

```
MIDDLEWARE_CLASSES = [  
    # ...  
  
    'django.middleware.clickjacking.XFrameOptionsMiddleware',  
    # ...  
]
```

This middleware is enabled in the settings file generated by `startproject`.

By default, the middleware will set the `X-Frame-Options` header to `SAMEORIGIN` for every outgoing `HttpResponse`. If you want `DENY` instead, set the `X_FRAME_OPTIONS` setting:

```
X_FRAME_OPTIONS = 'DENY'
```

When using the middleware there may be some views where you do not want the `X-Frame-Options` header set. For those cases, you can use a view decorator that tells the middleware not to set the header:

```
from django.http import HttpResponse
from django.views.decorators.clickjacking
import xframe_options_exempt

@xframe_options_exempt
def ok_to_load_in_a_frame(request):
    return HttpResponse("This page is safe
to load in a frame on any site.")
```

Setting X-Frame-Options per view

To set the `X-Frame-Options` header on a per view basis, Django provides these decorators:

```
from django.http import HttpResponse
from django.views.decorators.clickjacking
import xframe_options_deny
from django.views.decorators.clickjacking
import xframe_options_sameorigin

@xframe_options_deny
def view_one(request):
    return HttpResponse("I won't display
in any frame!")
```

```
@xframe_options_sameorigin
def view_two(request):
    return HttpResponse("Display in a
frame if it's from the same
origin as me.")
```

Note that you can use the decorators in conjunction with the middleware. Use of a decorator overrides the middleware.

LIMITATIONS

The [X-Frame-Options](#) header will only protect against clickjacking in a modern browser. Older browsers will quietly ignore the header and need other clickjacking prevention techniques.

BROWSERS THAT SUPPORT X-FRAME-OPTIONS

- Internet Explorer 8+
- Firefox 3.6.9+
- Opera 10.5+
- Safari 4+
- Chrome 4.1+

SSL/HTTPS

It is always better for security, though not always practical in all cases, to deploy your site behind HTTPS. Without this, it is possible for malicious network users to sniff authentication credentials or any other information

transferred between client and server, and in some cases-active network attackers-to alter data that is sent in either direction.

If you want the protection that HTTPS provides, and have enabled it on your server, there are some additional steps you may need:

- If necessary, set [SECURE_PROXY_SSL_HEADER](#), ensuring that you have understood the warnings there thoroughly. Failure to do this can result in CSRF vulnerabilities, and failure to do it correctly can also be dangerous!
- Set up redirection so that requests over HTTP are redirected to HTTPS.
- This could be done using a custom middleware. Please note the caveats under [SECURE_PROXY_SSL_HEADER](#). For the case of a reverse proxy, it may be easier or more secure to configure the main web server to do the redirect to HTTPS.
- Use *secure* cookies. If a browser connects initially via HTTP, which is the default for most browsers, it is possible for existing cookies to be leaked. For this reason, you should set your [SESSION_COOKIE_SECURE](#) and [CSRF_COOKIE_SECURE](#) settings to [True](#). This instructs the browser to only send these cookies over HTTPS connections. Note that this will mean that sessions will not work over HTTP, and the CSRF protection will prevent any [POST](#) data being accepted over HTTP (which will be fine if you are redirecting all HTTP traffic to HTTPS).
- Use HTTP Strict Transport Security (HSTS). HSTS is an HTTP header that informs a browser that all future connections to a particular site should always use HTTPS (see below). Combined with redirecting requests over HTTP to HTTPS, this will ensure that connections always enjoy the added security of SSL provided one successful connection has occurred. HSTS is usually configured on the web server.

HTTP STRICT TRANSPORT SECURITY

For sites that should only be accessed over HTTPS, you can instruct modern browsers to refuse to connect to your domain name via an insecure connection (for a given period of time) by setting the Strict-Transport-Security header. This reduces your exposure to some SSL-stripping Man-In-The-Middle (MITM) attacks.

[SecurityMiddleware](#) will set this header for you on all HTTPS responses if you set the `SECURE_HSTS_SECONDS` setting to a non-zero integer value.

When enabling HSTS, it's a good idea to first use a small value for testing, for example, `SECURE_HSTS_SECONDS = 3600` for one hour. Each time a web browser sees the HSTS header from your site, it will refuse to communicate non-securely (using HTTP) with your domain for the given period of time.

Once you confirm that all assets are served securely on your site (that is, HSTS didn't break anything), it's a good idea to increase this value so that infrequent visitors will be protected (31536000 seconds, that is, 1 year, is common).

Additionally, if you set the `SECURE_HSTS_INCLUDE_SUBDOMAINS` setting to `True`, [SecurityMiddleware](#) will add the `includeSubDomains` tag to the `Strict-Transport-Security` header. This is recommended (assuming all subdomains are served exclusively using HTTPS),

otherwise your site may still be vulnerable via an insecure connection to a subdomain.

NOTE

The HSTS policy applies to your entire domain, not just the URL of the response that you set the header on. Therefore, you should only use it if your entire domain is served via HTTPS only.

Browsers properly respecting the HSTS header will refuse to allow users to bypass warnings and connect to a site with an expired, self-signed, or otherwise invalid SSL certificate. If you use HSTS, make sure your certificates are in good shape and stay that way!

If you are deployed behind a load-balancer or reverse-proxy server, and the [Strict-Transport-Security](#) header is not being added to your responses, it may be because Django doesn't realize that it's on a secure connection; you may need to set the [SECURE_PROXY_SSL_HEADER](#) setting.

Host header validation

Django uses the `Host` header provided by the client to construct URLs in certain cases. While these values are sanitized to prevent Cross Site Scripting attacks, a fake `Host` value can be used for Cross-Site Request Forgery, cache poisoning attacks, and poisoning links in emails. Because even seemingly-secure web server configurations are susceptible to fake `Host` headers, Django validates `Host` headers against the `ALLOWED_HOSTS` setting in the `django.http.HttpRequest.get_host()` method. This validation only applies via `get_host()`; if your code accesses the `Host` header directly from `request.META` you are bypassing this security protection.

Session security

Similar to the CSRF limitations requiring a site to be deployed such that untrusted users don't have access to any subdomains, `django.contrib.sessions` also has limitations. See the session topic guide section on security for details.

USER-UPLOADED CONTENT

NOTE

Consider serving static files from a cloud service or CDN to avoid some of these issues.

- If your site accepts file uploads, it is strongly advised that you limit these uploads in your web server configuration to a reasonable size in order to prevent denial of service (DOS) attacks. In Apache, this can be easily set using the `LimitRequestBody` directive.
- If you are serving your own static files, be sure that handlers like Apache's `mod_php`, which would execute static files as code, are disabled. You don't want users to be able to execute arbitrary code by uploading and requesting a specially crafted file.
- Django's media upload handling poses some vulnerabilities when that media is served in ways that do not follow security best practices. Specifically, an HTML file can be uploaded as an image if that file contains a valid PNG header followed by malicious HTML. This file will pass verification of the library that Django uses for `ImageField` image processing (Pillow). When this file is subsequently displayed to a user, it may be displayed as HTML depending on the type and configuration of your web server.

No bulletproof technical solution exists at the framework level to safely validate all user uploaded file content, however, there are some other steps you can take to mitigate these attacks:

1. One class of attacks can be prevented by always serving user uploaded content from a distinct top-level or second-level domain. This prevents any exploit blocked by sameorigin policy(for more information visit http://en.wikipedia.org/wiki/Same-origin_policy) protections such as cross site scripting. For example, if your site runs on `example.com`, you would want to serve uploaded content (the `MEDIA_URL` setting) from something like `usercontent-example.com`. It's not sufficient to serve content from a subdomain like `usercontent.example.com`.
2. Beyond this, applications may choose to define a whitelist of allowable file extensions for user uploaded files and configure the web server to only serve such files.

Additional security tips

- While Django provides good security protection out of the box, it is still important to properly deploy your application and take advantage of the security protection of the web server, operating system and other components.
- Make sure that your Python code is outside of the web server's root. This will ensure that your Python code is not accidentally served as plain text (or accidentally executed).
- Take care with any user uploaded files.
- Django does not throttle requests to authenticate users. To protect against brute-force attacks against the authentication system, you may consider deploying a Django plugin or web server module to throttle these requests.
- Keep your `SECRET_KEY` a secret.
- It is a good idea to limit the accessibility of your caching system and database using a firewall.

Archive of security issues

Django's development team is strongly committed to responsible reporting and disclosure of security-related issues, as outlined in Django's security policies. As part of that commitment, they maintain an historical list of issues which have been fixed and disclosed. For the up to date list, see the archive of security issues (<https://docs.djangoproject.com/en/1.8/releases/security/>).

Cryptographic signing

The golden rule of web application security is to never trust data from untrusted sources. Sometimes it can be useful to pass data through an untrusted medium.

Cryptographically signed values can be passed through an untrusted channel safe in the knowledge that any tampering will be detected. Django provides both a low-level API for signing values and a high-level API for setting and reading signed cookies, one of the most common uses of signing in web applications. You may also find signing useful for the following:

- Generating *recover my account* URLs for sending to users who have lost their password.
- Ensuring data stored in hidden form fields has not been tampered with.
- Generating one-time secret URLs for allowing temporary access to a protected resource, for example, a downloadable file that a user has paid for.

PROTECTING THE SECRET_KEY

When you create a new Django project using `startproject`, the `settings.py` file is generated automatically and gets a random `SECRET_KEY` value. This value is the key to securing signed data—it is vital you keep this secure, or attackers could use it to generate their own signed values.

USING THE LOW-LEVEL API

Django's signing methods live in the `django.core.signing` module. To sign a value, first instantiate a `Signer` instance:

```
>>> from django.core.signing import Signer  
>>> signer = Signer()  
>>> value = signer.sign('My string')  
>>> value  
'My string:GdMGD6HNQ_qdgxYP8yBZAdAIV1w'
```

The signature is appended to the end of the string, following the colon. You can retrieve the original value using the `unsign` method:

```
>>> original = signer.unsign(value)  
>>> original  
'My string'
```

If the signature or value have been altered in any way, a `django.core.signing.BadSignature` exception will be raised:

```
>>> from django.core import signing  
>>> value += 'm'  
>>> try:  
...     original = signer.unsign(value)  
... except signing.BadSignature:  
...     print("Tampering detected!")
```

By default, the `Signer` class uses the `SECRET_KEY` setting to generate signatures. You can use a different secret by passing it to the `Signer` constructor:

```
>>> signer = Signer('my-other-secret')  
>>> value = signer.sign('My string')  
>>> value  
'My string:EkfQJafvGyiofrdGnuthdxImIJw'
```

`django.core.signing.Signer` returns a signer

which uses `key` to generate signatures and `sep` to separate values. `sep` cannot be in the URL safe base64 alphabet. This alphabet contains alphanumeric characters, hyphens, and underscores.

USING THE SALT ARGUMENT

If you do not wish for every occurrence of a particular string to have the same signature hash, you can use the optional `salt` argument to the `Signer` class. Using a salt will seed the signing hash function with both the salt and your `SECRET_KEY`:

```
>>> signer = Signer()
>>> signer.sign('My string')
'My string:GdMGD6HNQ_qdgxYP8yBZAdAIV1w'
>>> signer = Signer(salt='extra')
>>> signer.sign('My string')
'My string:Ee7vGi-ING6n02gkcJ-QLHg6vFw'
>>> signer.unsign('My string:Ee7vGi-
ING6n02gkcJ-QLHg6vFw')
'My string'
```

Using salt in this way puts the different signatures into different namespaces. A signature that comes from one namespace (a particular salt value) cannot be used to validate the same plaintext string in a different namespace that is using a different salt setting. The result is to prevent an attacker from using a signed string generated in one place in the code as input to another piece of code that is generating (and verifying) signatures using a different salt.

Unlike your `SECRET_KEY`, your salt argument does not need to stay secret.

VERIFYING TIMESTAMPED VALUES

`TimestampSigner` is a subclass of `Signer` that appends a signed timestamp to the value. This allows you to confirm that a signed value was created within a specified period of time:

```
>>> from datetime import timedelta
>>> from django.core.signing import
TimestampSigner
>>> signer = TimestampSigner()
>>> value = signer.sign('hello')
>>> value
'hello:1NMg5H:oPVuCqlJWmChm1rA2lyTUtelC-c'
>>> signer.unsign(value)
'hello'
>>> signer.unsign(value, max_age=10)
...
SignatureExpired: Signature age
15.5289158821 > 10 seconds
>>> signer.unsign(value, max_age=20)
'hello'
>>> signer.unsign(value,
max_age=timedelta(seconds=20))
'hello'
```

`sign(value)` signs `value` and appends the current timestamp.

`unsign(value, max_age=None)` checks if `value` was signed less than `max_age` seconds ago, otherwise raises `SignatureExpired`. The `max_age` parameter

can accept an integer or a `datetime.timedelta` object.

PROTECTING COMPLEX DATA STRUCTURES

If you wish to protect a list, tuple or dictionary you can do so using the signing module's `dumps` and `loads` functions. These imitate Python's pickle module, but use JSON serialization under the hood. JSON ensures that even if your `SECRET_KEY` is stolen an attacker will not be able to execute arbitrary commands by exploiting the pickle format:

```
>>> from django.core import signing
>>> value = signing.dumps({"foo": "bar"})
>>> value
'eyJmb28iOiJiYXIfQ:1NMg1b:zGcDE4-
TCkaeGzLeW9UQwZescii'
>>> signing.loads(value) {'foo': 'bar'}
```

Because of the nature of JSON (there is no native distinction between lists and tuples) if you pass in a tuple, you will get a list from `signing.loads(object)`:

```
>>> from django.core import signing
>>> value = signing.dumps(('a', 'b', 'c'))
>>> signing.loads(value)
['a', 'b', 'c']
```

`django.core.signing.dumps(obj, key=None,`
`salt='django.core.signing',`

```
compress=False)
```

Returns URL-safe, sha1 signed base64 compressed JSON string. Serialized object is signed using [TimestampSigner](#).

```
django.core.signing.loads(string,  
key=None, salt='django.core.signing',  
max_age=None)
```

Reverse of [dumps\(\)](#), raises [BadSignature](#) if signature fails. Checks [max_age](#) (in seconds) if given.

SECURITY MIDDLEWARE

NOTE

If your deployment situation allows, it's usually a good idea to have your front-end web server perform the functionality provided by the [SecurityMiddleware](#). That way, if there are requests that aren't served by Django (such as static media or user-uploaded files), they will have the same protections as requests to your Django application.

The

[django.middleware.security.SecurityMiddleware](#) provides several security enhancements to the request/response cycle. Each one can be independently enabled or disabled with a setting.

- [SECURE_BROWSER_XSS_FILTER](#)
- [SECURE_CONTENT_TYPE_NOSNIFF](#)
- [SECURE_HSTS_INCLUDE_SUBDOMAINS](#)
- [SECURE_HSTS_SECONDS](#)
- [SECURE_REDIRECT_EXEMPT](#)
- [SECURE_SSL_HOST](#)

- [SECURE_SSL_REDIRECT](#)

For more on security headers and these settings, see [Chapter 17, *Django Middleware*](#).

WHAT'S NEXT?

In the next chapter, we will expand on the quick install guide from [Chapter 1, *Introduction to Django and Getting Started*](#) and look at some additional installation and configuration options for Django.

Chapter 20. More on Installing Django

This chapter covers some of the more common additional options and scenarios associated with installing and maintaining Django. Firstly, we will look at installation configurations for using databases other than SQLite and then we will cover how to upgrade Django as well as how you can manually install Django. Finally, we will cover how to install the development version of Django just in case you want to play with the bleeding edge of Django development.

Running other databases

If you plan to use Django's database API functionality, you'll need to make sure a database server is running. Django supports many different database servers and is officially supported with PostgreSQL, MySQL, Oracle and SQLite.

Chapter 21, Advanced Database Management, contains additional information specific to connecting Django to each of these databases, however, it's beyond the scope of this book to show you how to install them; please refer to the database documentation at each projects' website.

If you are developing a simple project or something you

don't plan to deploy in a production environment, SQLite is generally the simplest option as it doesn't require running a separate server. However, SQLite has many differences from other databases, so if you are working on something substantial, it's recommended to develop with the same database as you plan on using in production.

In addition to a database backend, you'll need to make sure your Python database bindings are installed.

- If you're using PostgreSQL, you'll need the [postgresql_psycopg2](http://initd.org/psycopg/) (<http://initd.org/psycopg/>) package. You might want to refer to the PostgreSQL notes for further technical details specific to this database. If you're on Windows, check out the unofficial compiled Windows version (<http://stickpeople.com/projects/python/win-psycopg/>).
- If you're using MySQL, you'll need the [MySQL-python](#) package, version 1.2.1p2 or higher. You will also want to read the database-specific notes for the MySQL backend.
- If you're using SQLite, you might want to read the SQLite backend notes.
- If you're using Oracle, you'll need a copy of [cx_Oracle](http://cx-oracle.sourceforge.net/) (<http://cx-oracle.sourceforge.net/>), but please read the database-specific notes for the Oracle backend for important information regarding supported versions of both Oracle and [cx_Oracle](#).
- If you're using an unofficial 3rd party backend, please consult the documentation provided for any additional requirements.

If you plan to use Django's [manage.py migrate](#) command to automatically create database tables for your models (after first installing Django and creating a project), you'll need to ensure that Django has permission to create and alter tables in the database

you're using; if you plan to manually create the tables, you can simply grant Django `SELECT`, `INSERT`, `UPDATE` and `DELETE` permissions. After creating a database user with these permissions, you'll specify the details in your project's settings file, see [Databases](#) for details.

If you're using Django's testing framework to test database queries, Django will need permission to create a test database.

Installing Django manually

1. Download the latest release from the Django Project download page (<https://www.djangoproject.com/download/>).
2. Untar the downloaded file (for example, `tar xzvf Django-X.Y.tar.gz`, where `X.Y` is the version number of the latest release).
If you're using Windows, you can download the command-line tool `bsdtar` to do this, or you can use a GUI-based tool such as 7-zip (<http://www.7-zip.org/>).
3. Change into the directory created in step 2 (for example, `cd Django-X.Y`).
4. If you're using Linux, Mac OS X or some other flavor of Unix, enter the command `sudo python setup.py install` at the shell prompt.
If you're using Windows, start a command shell with administrator privileges and run the command `python setup.py install`. This will install Django in your Python installation's `site-packages` directory.

NOTE

Removing an old version

If you use this installation technique, it is particularly important that you remove any existing installations of Django first (see below). Otherwise, you can end up with a broken installation that includes files from previous versions that have since been removed from Django.

Upgrading Django

Remove any old versions of Django

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

If you installed Django using `pip` or `easy_install` previously, installing with `pip` or `easy_install` again will automatically take care of the old version, so you don't need to do it yourself.

If you previously installed Django manually, uninstalling is as simple as deleting the `django` directory from your Python `site-packages`. To find the directory you need to remove, you can run the following at your shell prompt (not the interactive Python prompt):

```
python -c "import sys; sys.path = sys.path[1:]; import django; print(django.__path__)"
```

Installing a Distribution-specific package

Check the distribution specific notes to see if your platform/distribution provides official Django packages/installers. Distribution-provided packages will typically allow for automatic installation of dependencies and easy upgrade paths; however, these packages will rarely contain the latest release of Django.

Installing the development version

If you decide to use the latest development version of Django, you'll want to pay close attention to the development timeline, and you'll want to keep an eye on the release notes for the upcoming release. This will help you stay on top of any new features you might want to use, as well as any changes you'll need to make to your code when updating your copy of Django. (For stable releases, any necessary changes are documented in the release notes.)

If you'd like to be able to update your Django code occasionally with the latest bug fixes and improvements, follow these instructions:

1. Make sure that you have Git installed and that you can run its commands from a shell. (Enter `git help` at a shell prompt to test this.)
2. Check out Django's main development branch (the *trunk* or *master*) like so:

```
git clone
```

```
git://github.com/django/django.git  
django-trunk
```

3. This will create a directory `django-trunk` in your current directory.
4. Make sure that the Python interpreter can load Django's code. The most convenient way to do this is via pip. Run the following command:

```
sudo pip install -e django-
```

trunk/

5. (If using a [virtualenv](#), or running Windows, you can omit [sudo](#).)

This will make Django's code importable, and will also make the [django-admin](#) utility command available. In other words, you're all set!

NOTE

Don't run `sudo python setup.py install`, because you've already carried out the equivalent actions in step 3.

When you want to update your copy of the Django source code, just run the command [git pull](#) from within the [django-trunk](#) directory. When you do this, Git will automatically download any changes.

What's next?

In the next chapter, we will be covering addition information specific to running Django with particular databases

Chapter 21. Advanced Database Management

This chapter provides additional information on each of the supported relational databases in Django, as well as notes and tips and tricks for connecting to legacy databases.

General notes

Django attempts to support as many features as possible on all database backends. However, not all database backends are alike, and the Django developers had to make design decisions on which features to support and which assumptions could be made safely.

This file describes some of the features that might be relevant to Django usage. Of course, it is not intended as a replacement for server-specific documentation or reference manuals.

Persistent connections

Persistent connections avoid the overhead of re-establishing a connection to the database in each request. They're controlled by the `CONN_MAX_AGE` parameter which defines the maximum lifetime of a connection. It can be set independently for each

database. The default value is 0, preserving the historical behavior of closing the database connection at the end of each request. To enable persistent connections, set `CONN_MAX_AGE` to a positive number of seconds. For unlimited persistent connections, set it to `None`.

CONNECTION MANAGEMENT

Django opens a connection to the database when it first makes a database query. It keeps this connection open and reuses it in subsequent requests. Django closes the connection once it exceeds the maximum age defined by `CONN_MAX_AGE` or when it isn't usable any longer.

In detail, Django automatically opens a connection to the database whenever it needs one and doesn't have one already-either because this is the first connection, or because the previous connection was closed.

At the beginning of each request, Django closes the connection if it has reached its maximum age. If your database terminates idle connections after some time, you should set `CONN_MAX_AGE` to a lower value, so that Django doesn't attempt to use a connection that has been terminated by the database server. (This problem may only affect very low traffic sites.)

At the end of each request, Django closes the connection if it has reached its maximum age or if it is in an unrecoverable error state. If any database errors

have occurred while processing the requests, Django checks whether the connection still works, and closes it if it doesn't. Thus, database errors affect at most one request; if the connection becomes unusable, the next request gets a fresh connection.

CAVEATS

Since each thread maintains its own connection, your database must support at least as many simultaneous connections as you have worker threads.

Sometimes a database won't be accessed by the majority of your views, for example, because it's the database of an external system, or thanks to caching. In such cases, you should set `CONN_MAX_AGE` to a low value or even `0`, because it doesn't make sense to maintain a connection that's unlikely to be reused. This will help keep the number of simultaneous connections to this database small.

The development server creates a new thread for each request it handles, negating the effect of persistent connections. Don't enable them during development.

When Django establishes a connection to the database, it sets up appropriate parameters, depending on the backend being used. If you enable persistent connections, this setup is no longer repeated every request. If you modify parameters such as the connection's isolation level or time zone, you should

either restore Django's defaults at the end of each request, force an appropriate value at the beginning of each request, or disable persistent connections.

Encoding

Django assumes that all databases use UTF-8 encoding. Using other encodings may result in unexpected behavior such as value too long errors from your database for data that is valid in Django. See the following database specific notes for information on how to set up your database correctly.

postgreSQL notes

Django supports PostgreSQL 9.0 and higher. It requires the use of Psycopg2 2.0.9 or higher.

Optimizing PostgreSQL's configuration

Django needs the following parameters for its database connections:

- `client_encoding: 'UTF8'`,
- `default_transaction_isolation: 'read committed'` by default, or the value set in the connection options (see here),
- `timezone: 'UTC'` when `USE_TZ` is `True`, value of `TIME_ZONE` otherwise.

If these parameters already have the correct values, Django won't set them for every new connection, which improves performance slightly. You can configure them directly in `postgresql.conf` or more conveniently per database user with `ALTER ROLE`.

Django will work just fine without this optimization, but each new connection will do some additional queries to set these parameters.

Isolation level

Like PostgreSQL itself, Django defaults to the `READ COMMITTED` isolation level. If you need a higher isolation level such as `REPEATABLE READ` or `SERIALIZABLE`, set it in the `OPTIONS` part of your database configuration in `DATABASES`:

```
import psycopg2.extensions

DATABASES = {
    # ...
    'OPTIONS': {
        'isolation_level':
            psycopg2.extensions.ISOLATION_LEVEL_SERIAL
            IZABLE,
    },
}
```

Under higher isolation levels, your application should be prepared to handle exceptions raised on serialization failures. This option is designed for advanced uses.

Indexes for `varchar` and `text` columns

When specifying `db_index=True` on your model fields, Django typically outputs a single `CREATE INDEX` statement. However, if the database type for the field is either `varchar` or `text` (for example, used by `CharField`, `FileField`, and `TextField`), then Django will create an additional index that uses an appropriate PostgreSQL operator class for the column. The extra index is necessary to correctly perform

lookups that use the `LIKE` operator in their SQL, as is done with the `contains` and `startswith` lookup types.

MySQL notes

Version support

Django supports MySQL 5.5 and higher.

Django's [inspectdb](#) feature uses the [information_schema](#) database, which contains detailed data on all database schemas.

Django expects the database to support Unicode (UTF-8 encoding) and delegates to it the task of enforcing transactions and referential integrity. It is important to be aware of the fact that the two latter ones aren't actually enforced by MySQL when using the MyISAM storage engine, see the next section.

Storage engines

MySQL has several storage engines. You can change the default storage engine in the server configuration.

Until MySQL 5.5.4, the default engine was MyISAM. The main drawbacks of MyISAM are that it doesn't support transactions or enforce foreign-key constraints. On the plus side, it was the only engine that supported full-text indexing and searching until MySQL 5.6.4.

Since MySQL 5.5.5, the default storage engine is

InnoDB. This engine is fully transactional and supports foreign key references. It's probably the best choice at this point. However, note that the InnoDB autoincrement counter is lost on a MySQL restart because it does not remember the `AUTO_INCREMENT` value, instead recreating it as `max(id)+1`. This may result in an inadvertent reuse of `AutoField` values.

If you upgrade an existing project to MySQL 5.5.5 and subsequently add some tables, ensure that your tables are using the same storage engine (that is MyISAM vs. InnoDB). Specifically, if tables that have a `ForeignKey` between them use different storage engines, you may see an error like the following when running `migrate`:

```
mysqlexceptions.OperationalError: (
    1005, "Can't create table
'\\db_name\\.#sql-4a8_ab' (errno: 150)"
)
```

MySQL DB API drivers

The Python Database API is described in PEP 249. MySQL has three prominent drivers that implement this API:

- MySQLdb (<https://pypi.python.org/pypi/MySQL-python/1.2.4>) is a native driver that has been developed and supported for over a decade by Andy Dustman.
- mySQLclient (<https://pypi.python.org/pypi/mysqlclient>) is a fork of `MySQLdb` which notably supports Python 3 and can be used as a drop-in replacement for MySQLdb. At the time of this writing, this is the recommended choice for using MySQL with Django.

- MySQL Connector/Python
(<http://dev.mysql.com/downloads/connector/python>) is a pure Python driver from Oracle that does not require the MySQL client library or any Python modules outside the standard library.

All these drivers are thread-safe and provide connection pooling. **MySQLdb** is the only one not supporting Python 3 currently.

In addition to a DB API driver, Django needs an adapter to access the database drivers from its ORM. Django provides an adapter for MySQLdb/mysqlclient while MySQL Connector/Python includes its own.

MySQLDB

Django requires MySQLdb version 1.2.1p2 or later.

If you see `ImportError: cannot import name
ImmutableSet` when trying to use Django, your MySQLdb installation may contain an outdated `sets.py` file that conflicts with the built-in module of the same name from Python 2.4 and later. To fix this, verify that you have installed MySQLdb version 1.2.1p2 or newer, then delete the `sets.py` file in the MySQLdb directory that was left by an earlier version.

There are known issues with the way MySQLdb converts date strings into datetime objects. Specifically, date strings with value `0000-00-00` are valid for MySQL but will be converted into `None` by MySQLdb.

This means you should be careful while using

`loaddata/dumpdata` with rows that may have `0000-00-00` values, as they will be converted to `None`.

At the time of writing, the latest release of MySQLdb (1.2.4) doesn't support Python 3. In order to use MySQLdb under Python 3, you'll have to install [mysqlclient](#).

MYSQLCLIENT

Django requires mysqlclient 1.3.3 or later. Note that Python 3.2 is not supported. Except for the Python 3.3+ support, mysqlclient should mostly behave the same as MySQLdb.

MYSQL CONNECTOR/PYTHON

MySQL Connector/Python is available from the download page. The Django adapter is available in versions 1.1.X and later. It may not support the most recent releases of Django.

Timezone definitions

If you plan on using Django's timezone support, use [mysql_tzinfo_to_sql](#) to load time zone tables into the MySQL database. This needs to be done just once for your MySQL server, not per database.

Creating your database

You can create your database using the command-line

tools and this SQL:

```
CREATE DATABASE <dbname> CHARACTER SET  
utf8;
```

This ensures all tables and columns will use UTF-8 by default.

COLLATION SETTINGS

The collation setting for a column controls the order in which data is sorted as well as what strings compare as equal. It can be set on a database-wide level and also per-table and per-column. This is documented thoroughly in the MySQL documentation. In all cases, you set the collation by directly manipulating the database tables; Django doesn't provide a way to set this on the model definition.

By default, with a UTF-8 database, MySQL will use the [utf8_general_ci](#) collation. This results in all string equality comparisons being done in a *case-insensitive* manner. That is, "Fred" and "freD" are considered equal at the database level. If you have a unique constraint on a field, it would be illegal to try to insert both "aa" and "AA" into the same column, since they compare as equal (and, hence, non-unique) with the default collation.

In many cases, this default will not be a problem. However, if you really want case-sensitive comparisons on a particular column or table, you would change the

column or table to use the `utf8_bin` collation. The main thing to be aware of in this case is that if you are using MySQLdb 1.2.2, the database backend in Django will then return bytestrings (instead of Unicode strings) for any character fields it receives from the database. This is a strong variation from Django's normal practice of *always* returning Unicode strings.

It is up to you, the developer, to handle the fact that you will receive bytestrings if you configure your table(s) to use `utf8_bin` collation. Django itself should mostly work smoothly with such columns (except for the `contrib.sessions.Session` and `contrib.admin.LogEntry` tables described here), but your code must be prepared to call `django.utils.encoding.smart_text()` at times if it really wants to work with consistent data-Django will not do this for you (the database backend layer and the model population layer are separated internally so the database layer doesn't know it needs to make this conversion in this one particular case).

If you're using MySQLdb 1.2.1p2, Django's standard `CharField` class will return Unicode strings even with `utf8_bin` collation. However, `TextField` fields will be returned as an `array.array` instance (from Python's standard `array` module). There isn't a lot Django can do about that, since, again, the information needed to make the necessary conversions isn't available when the data is read in from the database. This problem was fixed in

MySQLdb 1.2.2, so if you want to use `TextField` with `utf8_bin` collation, upgrading to version 1.2.2 and then dealing with the byte strings (which shouldn't be too difficult) as described before is the recommended solution.

Should you decide to use `utf8_bin` collation for some of your tables with MySQLdb 1.2.1p2 or 1.2.2, you should still use `utf8_general_ci` (the default) collation for the

`django.contrib.sessions.models.Session` table (usually called `django_session`) and the `django.contrib.admin.models.LogEntry` table (usually called `django_admin_log`). Please note that according to MySQL Unicode Character Sets, comparisons for the `utf8_general_ci` collation are faster, but slightly less correct, than comparisons for `utf8_unicode_ci`. If this is acceptable for your application, you should use `utf8_general_ci` because it is faster. If this is not acceptable (for example, if you require German dictionary order), use `utf8_unicode_ci` because it is more accurate.

NOTE

Model formsets validate unique fields in a case-sensitive manner. Thus when using a case-insensitive collation, a formset with unique field values that differ only by case will pass validation, but upon calling `save()`, an `IntegrityError` will be raised.

Connecting to the database

Connection settings are used in this order:

- `OPTIONS`
- `NAME`, `USER`, `PASSWORD`, `HOST`, `PORT`
- MySQL option files

In other words, if you set the name of the database in `OPTIONS`, this will take precedence over `NAME`, which would override anything in a MySQL option file. Here's a sample configuration which uses a MySQL option file:

```
# settings.py
DATABASES = {
    'default': {
        'ENGINE':
'django.db.backends.mysql',
        'OPTIONS': {'read_default_file':
'path/to/my.cnf',},
    }
}

# my.cnf
[client]
database = NAME
user = USER
password = PASSWORD
default-character-set = utf8
```

Several other MySQLdb connection options may be useful, such as `ssl`, `init_command`, and `sql_mode`. Consult the MySQLdb documentation for more details.

Creating your tables

When Django generates the schema, it doesn't specify a storage engine, so tables will be created with whatever default storage engine your database server is

configured for.

The easiest solution is to set your database server's default storage engine to the desired engine.

If you're using a hosting service and can't change your server's default storage engine, you have a couple of options.

- After the tables are created, execute an `ALTER TABLE` statement to convert a table to a new storage engine (such as InnoDB):

```
ALTER TABLE <tablename> ENGINE=INNODB;
```

- This can be tedious if you have a lot of tables.
- Another option is to use the `init_command` option for MySQLdb prior to creating your tables:

```
'OPTIONS': {  
    'init_command': 'SET  
storage_engine=INNODB',  
}
```

This sets the default storage engine upon connecting to the database. After your tables have been created, you should remove this option as it adds a query that is only needed during table creation to each database connection.

Table names

There are known issues in even the latest versions of MySQL that can cause the case of a table name to be

altered when certain SQL statements are executed under certain conditions. It is recommended that you use lowercase table names, if possible, to avoid any problems that might arise from this behavior. Django uses lowercase table names when it auto-generates table names from models, so this is mainly a consideration if you are overriding the table name via the `db_table` parameter.

Savepoints

Both the Django ORM and MySQL (when using the InnoDB storage engine) support database savepoints.

If you use the MyISAM storage engine, please be aware of the fact that you will receive database-generated errors if you try to use the savepoint-related methods of the transactions API. The reason for this is that detecting the storage engine of a MySQL database/table is an expensive operation so it was decided it isn't worth to dynamically convert these methods in no-op's based in the results of such detection.

Notes on specific fields

CHARACTER FIELDS

Any fields that are stored with `VARCHAR` column types have their `max_length` restricted to 255 characters if you are using `unique=True` for the field. This affects `CharField`, `SlugField` and

`CommaSeparatedIntegerField`.

FRACTIONAL SECONDS SUPPORT FOR TIME AND DATETIME FIELDS

MySQL 5.6.4 and later can store fractional seconds, provided that the column definition includes a fractional indication (for example, `DATETIME(6)`). Earlier versions do not support them at all. In addition, versions of MySQLdb older than 1.2.5 have a bug that also prevents the use of fractional seconds with MySQL.

Django will not upgrade existing columns to include fractional seconds if the database server supports it. If you want to enable them on an existing database, it's up to you to either manually update the column on the target database, by executing a command such as:

```
ALTER TABLE `your_table` MODIFY  
  `your_datetime_column` DATETIME(6)
```

or using a `RunSQL` operation in a `data migration`.

By default, new `DateTimeField` or `TimeField` columns are now created with fractional seconds support on MySQL 5.6.4 or later with either mysqlclient or MySQLdb 1.2.5 or later.

TIMESTAMP COLUMNS

If you are using a legacy database that contains `TIMESTAMP` columns, you must set `USE_TZ = False`

to avoid data corruption. `inspectdb` maps these columns to `DateTimeField` and if you enable timezone support, both MySQL and Django will attempt to convert the values from UTC to local time.

ROW LOCKING WITH `QUERYSET.SELECT_FOR_UPDATE()`

MySQL does not support the `NOWAIT` option to the `SELECT ... FOR UPDATE` statement. If `select_for_update()` is used with `nowait=True` then a `DatabaseError` will be raised.

AUTOMATIC TYPECASTING CAN CAUSE UNEXPECTED RESULTS

When performing a query on a string type, but with an integer value, MySQL will coerce the types of all values in the table to an integer before performing the comparison. If your table contains the values "`abc`", "`def`" and you query for `WHERE mycolumn=0`, both rows will match. Similarly, `WHERE mycolumn=1` will match the value "`abc1`". Therefore, string type fields included in Django will always cast the value to a string before using it in a query.

If you implement custom model fields that inherit from `Field` directly, are overriding `get_prep_value()`, or use `extra()` or `raw()`, you should ensure that you perform the appropriate typecasting.

SQLite notes

SQLite provides an excellent development alternative for applications that are predominantly read-only or require a smaller installation footprint. As with all database servers, though, there are some differences that are specific to SQLite that you should be aware of.

Substring matching and case sensitivity

For all SQLite versions, there is some slightly counter-intuitive behavior when attempting to match some types of strings. These are triggered when using the `iexact` or `contains` filters in Querysets. The behavior splits into two cases:

1. For substring matching, all matches are done case-insensitively. That is a filter such as `filter(name__contains="aa")` will match a name of "Aabb".
2. For strings containing characters outside the ASCII range, all exact string matches are performed case-sensitively, even when the case-insensitive options are passed into the query. So the `iexact` filter will behave exactly the same as the exact filter in these cases.

Some possible workarounds for this are documented at sqlite.org, but they aren't utilized by the default SQLite backend in Django, as incorporating them would be fairly difficult to do robustly. Thus, Django exposes the default SQLite behavior and you should be aware of this when

doing case-insensitive or substring filtering.

Old SQLite and CASE expressions

SQLite 3.6.23.1 and older contains a bug when handling query parameters in a [CASE](#) expression that contains an [ELSE](#) and arithmetic.

SQLite 3.6.23.1 was released in March 2010, and most current binary distributions for different platforms include a newer version of SQLite, with the notable exception of the Python 2.7 installers for Windows.

As of this writing, the latest release for Windows-Python 2.7.10-includes SQLite 3.6.21. You can install [pysqlite2](#) or replace [sqlite3.dll](#) (by default installed in [C:\Python27\DLLs](#)) with a newer version from [sqlite.org](#) to remedy this issue.

Using newer versions of the SQLite DB-API 2.0 driver

Django will use a [pysqlite2](#) module in preference to [sqlite3](#) as shipped with the Python standard library if it finds one is available.

This provides the ability to upgrade both the DB-API 2.0 interface or SQLite 3 itself to versions newer than the ones included with your particular Python binary distribution, if needed.

Database is locked errors

SQLite is meant to be a lightweight database, and thus can't support a high level of concurrency.

`OperationalError: database is locked` errors indicate that your application is experiencing more concurrency than `sqlite` can handle in default configuration. This error means that one thread or process has an exclusive lock on the database connection and another thread timed out waiting for the lock to be released.

Python's SQLite wrapper has a default timeout value that determines how long the second thread is allowed to wait on the lock before it times out and raises the `OperationalError: database is locked` error.

If you're getting this error, you can solve it by:

- Switching to another database backend. At a certain point SQLite becomes too light for real-world applications, and these sorts of concurrency errors indicate you've reached that point.
- Rewriting your code to reduce concurrency and ensure that database transactions are short-lived.
- Increase the default timeout value by setting the `timeout` database options:

```
'OPTIONS': { # ... 'timeout': 20, # ... }
```

This will simply make SQLite wait a bit longer before throwing database is locked errors; it won't really do anything to solve them.

QUERYSET.SELECT_FOR_UPDATE() NOT SUPPORTED

SQLite does not support the `SELECT ... FOR UPDATE` syntax. Calling it will have no effect.

PYFORMAT PARAMETER STYLE IN RAW QUERIES NOT SUPPORTED

For most backends, raw queries (`Manager.raw()` or `cursor.execute()`) can use the pyformat parameter style, where placeholders in the query are given as '`%(name)s`' and the parameters are passed as a dictionary rather than a list. SQLite does not support this.

PARAMETERS NOT QUOTED IN CONNECTION.QUERIES

`sqlite3` does not provide a way to retrieve the SQL after quoting and substituting the parameters. Instead, the SQL in `connection.queries` is rebuilt with a simple string interpolation. It may be incorrect. Make sure you add quotes where necessary before copying a query into an SQLite shell.

Oracle notes

Django supports Oracle Database Server versions 11.1 and higher. Version 4.3.1 or higher of the [cx_Oracle](http://cx-oracle.sourceforge.net/) (<http://cx-oracle.sourceforge.net/>) Python driver is required, although we recommend version 5.1.3 or later as these versions support Python 3.

Note that due to a Unicode-corruption bug in [cx_Oracle](#) 5.0, that version of the driver should **not** be used with Django; [cx_Oracle](#) 5.0.1 resolved this issue, so if you'd like to use a more recent [cx_Oracle](#), use version 5.0.1.

[cx_Oracle](#) 5.0.1 or greater can optionally be compiled with the `WITH_UNICODE` environment variable. This is recommended but not required.

In order for the `python manage.py migrate` command to work, your Oracle database user must have privileges to run the following commands:

- `CREATE TABLE`
- `CREATE SEQUENCE`
- `CREATE PROCEDURE`
- `CREATE TRIGGER`

To run a project's test suite, the user usually needs these *additional* privileges:

- `CREATE USER`
- `DROP USER`
- `CREATE TABLESPACE`
- `DROP TABLESPACE`
- `CREATE SESSION WITH ADMIN OPTION`
- `CREATE TABLE WITH ADMIN OPTION`
- `CREATE SEQUENCE WITH ADMIN OPTION`
- `CREATE PROCEDURE WITH ADMIN OPTION`
- `CREATE TRIGGER WITH ADMIN OPTION`

Note that, while the `RESOURCE` role has the required `CREATE TABLE`, `CREATE SEQUENCE`, `CREATE PROCEDURE` and `CREATE TRIGGER` privileges, and a user granted `RESOURCE WITH ADMIN OPTION` can grant `RESOURCE`, such a user cannot grant the individual privileges (for example, `CREATE TABLE`), and thus `RESOURCE WITH ADMIN OPTION` is not usually sufficient for running tests.

Some test suites also create views; to run these, the user also needs the `CREATE VIEW WITH ADMIN OPTION` privilege. In particular, this is needed for Django's own test suite.

All of these privileges are included in the DBA role, which is appropriate for use on a private developer's database.

The Oracle database backend uses the `SYS.DBMS_LOB` package, so your user will require execute permissions

on it. It's normally accessible to all users by default, but in case it is not, you'll need to grant permissions like so:

```
GRANT EXECUTE ON SYS.DBMS_LOB TO user;
```

Connecting to the database

To connect using the service name of your Oracle database, your `settings.py` file should look something like this:

```
DATABASES = {
    'default': {
        'ENGINE':
'django.db.backends.oracle',
        'NAME': 'xe',
        'USER': 'a_user',
        'PASSWORD': 'a_password',
        'HOST': '',
        'PORT': '',
    }
}
```

In this case, you should leave both `HOST` and `PORT` empty. However, if you don't use a `tnsnames.ora` file or a similar naming method and want to connect using the SID (`xe` in this example), then fill in both `HOST` and `PORT` like so:

```
DATABASES = {
    'default': {
        'ENGINE':
'django.db.backends.oracle',
        'NAME': 'xe',
        'USER': 'a_user',
```

```
        'PASSWORD': 'a_password',
        'HOST':
    'dbprod01ned.mycompany.com',
        'PORT': '1540',
    }
}
```

You should either supply both `HOST` and `PORT`, or leave both as empty strings. Django will use a different connect descriptor depending on that choice.

Threaded option

If you plan to run Django in a multithreaded environment (for example, Apache using the default MPM module on any modern operating system), then you **must** set the `threaded` option of your Oracle database configuration to True:

```
'OPTIONS': {
    'threaded': True,
},
```

Failure to do this may result in crashes and other odd behavior.

INSERT ... RETURNING INTO

By default, the Oracle backend uses a `RETURNING INTO` clause to efficiently retrieve the value of an `AutoField` when inserting new rows. This behavior may result in a `DatabaseError` in certain unusual setups, such as when inserting into a remote table, or

into a view with an `INSTEAD OF` trigger.

The `RETURNING INTO` clause can be disabled by setting the `use_returning_into` option of the database configuration to `False`:

```
'OPTIONS': {  
    'use_returning_into': False,  
},
```

In this case, the Oracle backend will use a separate `SELECT` query to retrieve `AutoField` values.

Naming issues

Oracle imposes a name length limit of 30 characters.

To accommodate this, the backend truncates database identifiers to fit, replacing the final four characters of the truncated name with a repeatable MD5 hash value. Additionally, the backend turns database identifiers to all-uppercase.

To prevent these transformations (this is usually required only when dealing with legacy databases or accessing tables which belong to other users), use a quoted name as the value for `db_table`:

```
class LegacyModel(models.Model):  
    class Meta:  
        db_table =  
            '"name_left_in_lowercase"'
```

```
class ForeignModel(models.Model):
    class Meta:
        db_table =
        '"OTHER_USER", "NAME_ONLY_SEEMS_OVER_30"'
```

Quoted names can also be used with Django's other supported database backends; except for Oracle, however, the quotes have no effect.

When running `migrate`, an `ORA-06552` error may be encountered if certain Oracle keywords are used as the name of a model field or the value of a `db_column` option. Django quotes all identifiers used in queries to prevent most such problems, but this error can still occur when an Oracle datatype is used as a column name. In particular, take care to avoid using the names `date`, `timestamp`, `number` or `float` as a field name.

NULL and empty strings

Django generally prefers to use the empty string (" ") rather than `NULL`, but Oracle treats both identically. To get around this, the Oracle backend ignores an explicit `null` option on fields that have the empty string as a possible value and generates DDL as if `null=True`. When fetching from the database, it is assumed that a `NULL` value in one of these fields really means the empty string, and the data is silently converted to reflect this assumption.

Textfield limitations

The Oracle backend stores `TextField`s as `NLOB` columns. Oracle imposes some limitations on the usage of such LOB columns in general:

- LOB columns may not be used as primary keys.
- LOB columns may not be used in indexes.
- LOB columns may not be used in a `SELECT DISTINCT` list. This means that attempting to use the `QuerySet.distinct` method on a model that includes `TextField` columns will result in an error when run against Oracle. As a workaround, use the `QuerySet.defer` method in conjunction with `distinct()` to prevent `TextField` columns from being included in the `SELECT DISTINCT` list.

Using a 3rd-Party database backend

In addition to the officially supported databases, there are backends provided by 3rd parties that allow you to use other databases with Django:

- SAP SQL Anywhere
- IBM DB2
- Microsoft SQL Server
- Firebird
- ODBC
- ADSDB

The Django versions and ORM features supported by these unofficial backends vary considerably. Queries regarding the specific capabilities of these unofficial backends, along with any support queries, should be directed to the support channels provided by each 3rd party project.

Integrating Django with a legacy database

While Django is best suited for developing new applications, it's quite possible to integrate it into legacy databases. Django includes a couple of utilities to automate as much of this process as possible.

Once you've got Django set up, you'll follow this general process to integrate with an existing database.

Give Django your database parameters

You'll need to tell Django what your database connection parameters are, and what the name of the database is. Do that by editing the `DATABASES` setting and assigning values to the following keys for the '`default`' connection:

- `NAME`
- `ENGINE <DATABASE-ENGINE>`
- `USER`
- `PASSWORD`
- `HOST`
- `PORT`

Auto-generate the models

Django comes with a utility called `inspectdb` that can create models by introspecting an existing database. You can view the output by running this command:

```
python manage.py inspectdb
```

Save this as a file by using standard Unix output redirection:

```
python manage.py inspectdb > models.py
```

This feature is meant as a shortcut, not as definitive model generation. See the documentation of `inspectdb` for more information.

Once you've cleaned up your models, name the file `models.py` and put it in the Python package that holds your app. Then add the app to your `INSTALLED_APPS` setting.

By default, `inspectdb` creates unmanaged models. That is, `managed = False` in the model's `Meta` class tells Django not to manage each table's creation, modification, and deletion:

```
class Person(models.Model):
    id =
        models.IntegerField(primary_key=True)
    first_name =
        models.CharField(max_length=70)
    class Meta:
```

```
managed = False  
db_table = 'CENSUS_PERSONS'
```

If you do want to allow Django to manage the table's lifecycle, you'll need to change the `managed` option preceding to `True` (or simply remove it because `True` is its default value).

Install the core Django tables

Next, run the `migrate` command to install any extra needed database records such as admin permissions and content types:

```
python manage.py migrate
```

Cleaning up generated models

As you might expect, the database introspection isn't perfect, and you'll need to do some light clean-up of the resulting model code. Here are a few pointers for dealing with the generated models:

- Each database table is converted to a model class (that is, there is a one-to-one mapping between database tables and model classes). This means that you'll need to refactor the models for any many-to-many join tables into `ManyToManyField` objects.
- Each generated model has an attribute for every field, including id primary key fields. However, recall that Django automatically adds an id primary key field if a model doesn't have a primary key. Thus, you'll want to remove any lines that look like this:

```
id = models.IntegerField(primary_key=True)
```

- Not only are these lines redundant, but also they can cause problems if your application will be adding *new* records to these tables.
- Each field's type (for example, `CharField`, `DateField`) is determined by looking at the database column type (for example, `VARCHAR`, `DATE`). If `inspectdb` cannot map a column's type to a model field type, it will use `TextField` and will insert the Python comment `'This field type is a guess.'` next to the field in the generated model. Keep an eye out for that, and change the field type accordingly if needed.
- If a field in your database has no good Django equivalent, you can safely leave it off. The Django model layer is not required to include every field in your table(s).
- If a database column name is a Python reserved word (such as `pass`, `class`, or `for`), `inspectdb` will append `"_field"` to the attribute name and set the `db_column` attribute to the real field name (for example, `pass`, `class`, or `for`).
- For example, if a table has an `INT` column called `for`, the generated model will have a field like this:

```
for_field =
    models.IntegerField(db_column='for')
```

- `inspectdb` will insert the Python comment `'Field renamed because it was a Python reserved word.'` next to the field.
- If your database contains tables that refer to other tables (as most databases do), you might need to rearrange the order of the generated models so that models that refer to other models are ordered properly. For example, if model `Book` has a `ForeignKey` to model `Author`, model `Author` should be defined before model `Book`. If you need to create a relationship on a model that has not yet been defined, you can use a string containing the name of the model, rather than the model object itself.
- `inspectdb` detects primary keys for PostgreSQL, MySQL, and SQLite. That is, it inserts `primary_key=True` where appropriate. For other databases, you'll need to insert `primary_key=True` for at least one field in each model, because Django models are required to have a `primary_key=True` field.

- Foreign-key detection only works with PostgreSQL and with certain types of MySQL tables. In other cases, foreign-key fields will be generated as `IntegerField`'s, assuming the foreign-key column was an `INT` column.

Test and tweak

Those are the basic steps-from here you'll want to tweak the models Django generated until they work the way you'd like. Try accessing your data via the Django database API, and try editing objects via Django's admin site, and edit the models file accordingly.

What's next?

That's all folks!

I hope you have enjoyed reading *Mastering Django: Core* and have learned a lot from the book. While this book will serve you as a complete reference to Django, there is still no substitute for plain old practice-so get coding and all the best with your Django career!

The remaining chapters are purely for your reference. They include appendices and quick references for all of the functions and fields in Django

Appendix A. Model Definition Reference

Chapter 4, *Models*, explains the basics of defining models, and we use them throughout the rest of the book. There is, however, a huge range of model options available not covered elsewhere. This appendix explains each possible model definition option.

Fields

The most important part of a model-and the only required part of a model-is the list of database fields it defines.

Field name restrictions

Django places only two restrictions on model field names:

1. A field name cannot be a Python reserved word, because that would result in a Python syntax error. For example:
`Example(models.Model): pass = models.IntegerField() # 'pass' is a reserved word!`
2. A field name cannot contain more than one underscore in a row, due to the way Django's query lookup syntax works. For example:

```
class Example(models.Model):
    # 'foo__bar' has two
    underscores!
    foo__bar =
```

```
models.IntegerField()
```

Each field in your model should be an instance of the appropriate [Field](#) class. Django uses the field class types to determine a few things:

- The database column type (for example, [INTEGER](#), [VARCHAR](#))
- The widget to use in Django's forms and admin site, if you care to use it (for example, `<input type="text">`, `<select>`)
- The minimal validation requirements, which are used in Django's admin interface and by forms

Each field class can be passed a list of option arguments, for example when we were building the book model in [Chapter 4, Models](#), our `num_pages` field looked like this: `num_pages = models.IntegerField(blank=True, null=True)`

In this case, we are setting the `blank` and `null` options for the field class. *Table A.2* lists all the field options in Django.

A number of fields also define additional options specific to that class, for example the [CharField](#) class has a required option `max_length` which defaults to `None`. For example: `title = models.CharField(max_length=100)`

In this case we are setting the `max_length` field option to 100 to limit our book titles to 100 characters.

A complete list of field classes is in *Table A.1*, sorted alphabetically.

Field	Default Width	Description
AutoField	N/A	An IntegerField that automatically increments according to available IDs.
BigIntegerField	Number Input	A 64-bit integer, much like an IntegerField except that it is guaranteed to fit numbers from -9223372036854775808 to 9223372036854775807
BinaryField	N/A	A field to store raw binary data. It only supports bytes assignment. Be aware that this field has limited functionality.
BooleanField	Checkboxes Input	A true/false field. If you need to accept null values then use NullBooleanField instead.
CharField	Text Input	A string field, for small-to large-sized strings. For large amounts of text, use TextField . CharField has one extra required argument: max_length . The maximum length (in characters) of the field.

<code>DateField</code>	<code>DateInput</code>	A date, represented in Python by a <code>datetime.date</code> instance. Has two extra, optional arguments: <code>auto_now</code> which automatically set the field to now every time the object is saved, and <code>auto_now_add</code> which automatically set the field to now when the object is first created.
<code>DateTimeField</code>	<code>DatetimeInput</code>	A date and time, represented in Python by a <code>datetime.datetime</code> instance. Takes the same extra arguments as <code>DateField</code> .
<code>DecimalField</code>	<code>TextInput</code>	A fixed-precision decimal number, represented in Python by a <code>Decimal</code> instance. Has two required arguments: <code>max_digits</code> and <code>decimal_places</code> .
<code>DurationField</code>	<code>TextInput</code>	A field for storing periods of time-modeled in Python by <code>timedelta</code> .
<code>EmailField</code>	<code>TextInput</code>	A <code>CharField</code> that uses <code>EmailValidator</code> to validate the input. <code>max_length</code> defaults to <code>254</code> .
<code>FileField</code>	<code>ClearableFileInput</code>	A file upload field. For more information on <code>FileField</code> , see the next section.

<code>FileField</code>	<code>Select</code>	A <code>CharField</code> whose choices are limited to the filenames in a certain directory on the filesystem.
<code>FloatField</code>	<code>NumberInput</code>	A floating-point number represented in Python by a <code>float</code> instance. Note when <code>field.localize</code> is <code>False</code> , the default widget is <code>TextInput</code>
<code>ImageField</code>	<code>ClearableFileInput</code>	Inherits all attributes and methods from <code>FileField</code> , but also validates that the uploaded object is a valid image. Additional <code>height</code> and <code>width</code> attributes. Requires the Pillow library available at http://pillow.readthedocs.org/en/latest/ .
<code>IntegerField</code>	<code>NumberInput</code>	An integer. Values from <code>-2147483648</code> to <code>2147483647</code> are safe in all databases supported by Django.
<code>GenericIPAddressField</code>	<code>TextInput</code>	An IPv4 or IPv6 address, in string format (for example, <code>192.0.2.30</code> or <code>2a02:42fe::4</code>).
<code>...</code>	<code>Null</code>	

<code>NullBooleanField</code>	<code>BooleanField</code>	Like a <code>BooleanField</code> , but allows <code>NULL</code> as one of the options.
<code>PositiveIntegerField</code>	<code>NumberInput</code>	An integer. Values from <code>0</code> to <code>2147483647</code> are safe in all databases supported by Django.
<code>SlugField</code>	<code>TextInput</code>	Slug is a newspaper term. A slug is a short label for something, containing only letters, numbers, underscores or hyphens.
<code>SmallIntegerField</code>	<code>NumberInput</code>	Like an <code>IntegerField</code> , but only allows values under a certain point. Values from <code>-32768</code> to <code>32767</code> are safe in all databases supported by Django.
<code>TextField</code>	<code>Textarea</code>	A large text field. If you specify a <code>max_length</code> attribute, it will be reflected in the <code>Textarea</code> widget of the auto-generated form field.
<code>TimeField</code>	<code>TextInput</code>	A time, represented in Python by a <code>datetime.time</code> instance.
	<code>... ...</code>	

<code>URLField</code>	<code>URLInput</code>	A <code>CharField</code> for a URL. Optional <code>max_length</code> argument.
<code>UUIDField</code>	<code>TextInput</code>	A field for storing universally unique identifiers. Uses Python's <code>UUID</code> class.

Table A.1: Django model field reference

FileField notes

The `primary_key` and `unique` arguments are not supported, and will raise a `TypeError` if used.

- Has two optional arguments: `FileField.upload_to`
- `FileField.storage`

FILEFIELD FILEFIELD.UPLOAD_TO

A local filesystem path that will be appended to your `MEDIA_ROOT` setting to determine the value of the `url` attribute. This path may contain `strftime()` formatting, which will be replaced by the date/time of the file upload (so that uploaded files don't fill up the given directory). This may also be a callable, such as a function, which will be called to obtain the upload path, including the filename. This callable must be able to accept two arguments, and return a Unix-style path (with forward slashes) to be passed along to the storage system.

The two arguments that will be passed are:

- **Instance:** An instance of the model where the FileField is defined. More specifically, this is the particular instance where the current file is being attached. In most cases, this object will not have been saved to the database yet, so if it uses the default `AutoField`, it might not yet have a value for its primary key field.
- **Filename:** The filename that was originally given to the file. This may or may not be taken into account when determining the final destination path.

FILEFIELD.STORAGE

A storage object, which handles the storage and retrieval of your files. The default form widget for this field is a `ClearableFileInput`. Using a `FileField` or an `ImageField` (see below) in a model takes a few steps:

- In your settings file, you'll need to define `MEDIA_ROOT` as the full path to a directory where you'd like Django to store uploaded files. (For performance, these files are not stored in the database.) Define `MEDIA_URL` as the base public URL of that directory. Make sure that this directory is writable by the web server's user account.
- Add the `FileField` or `ImageField` to your model, defining the `upload_to` option to specify a subdirectory of `MEDIA_ROOT` to use for uploaded files.
- All that will be stored in your database is a path to the file (relative to `MEDIA_ROOT`). You'll most likely want to use the convenient `url` attribute provided by Django. For example, if your `ImageField` is called `mug_shot`, you can get the absolute path to your image in a template with `{{ object.mug_shot.url }}`.

Note that whenever you deal with uploaded files, you should pay close attention to where you're uploading them and what type of files they are, to avoid security holes. Validate all uploaded files so that you're sure the

files are what you think they are. For example, if you blindly let somebody upload files, without validation, to a directory that's within your web server's document root, then somebody could upload a CGI or PHP script and execute that script by visiting its URL on your site. Don't allow that.

Also note that even an uploaded HTML file, since it can be executed by the browser (though not by the server), can pose security threats that are equivalent to XSS or CSRF attacks. `FileField` instances are created in your database as `varchar` columns with a default max length of 100 characters. As with other fields, you can change the maximum length using the `max_length` argument.

FILEFIELD AND FIELDFILE

When you access a `FileField` on a model, you are given an instance of `FieldFile` as a proxy for accessing the underlying file. In addition to the functionality inherited from `django.core.files.File`, this class has several attributes and methods that can be used to interact with file data:

FieldFile.url

A read-only property to access the file's relative URL by calling the `url()` method of the underlying `Storage` class.

FieldFile.open(mode='rb')

Behaves like the standard Python `open()` method and opens the file associated with this instance in the mode specified by `mode`.

FieldFile.close()

Behaves like the standard Python `file.close()` method and closes the file associated with this instance.

FieldFile.save(name, content, save=True)

This method takes a filename and file contents and passes them to the storage class for the field, then associates the stored file with the model field. If you want to manually associate file data with `FileField` instances on your model, the `save()` method is used to persist that file data.

Takes two required arguments: `name` which is the name of the file, and `content` which is an object containing the file's contents. The optional `save` argument controls whether or not the model instance is saved after the file associated with this field has been altered. Defaults to `True`.

Note that the `content` argument should be an instance of `django.core.files.File`, not Python's built-in file object. You can construct a `File` from an existing Python file object like this:

```
from django.core.files import File
# Open an existing file using Python's built-in open()
f = open('tmpHello.world')
myfile = File(f)
```

Or you can construct one from a Python string like this:

```
from django.core.files.base import  
ContentFile  
myfile = ContentFile("hello world")
```

FieldFile.delete(save=True)

Deletes the file associated with this instance and clears all attributes on the field. This method will close the file if it happens to be open when `delete()` is called.

The optional `save` argument controls whether or not the model instance is saved after the file associated with this field has been deleted. Defaults to `True`.

Note that when a model is deleted, related files are not deleted. If you need to clean up orphaned files, you'll need to handle it yourself (for instance, with a custom management command that can be run manually or scheduled to run periodically via for example, [cron](#)).

Universal field options

Table A.2 lists all the optional field arguments in Django. They are available to all field types.

Option	Description
null	If <code>True</code> , Django will store empty values as <code>NULL</code> in the database. Default is <code>False</code> . Avoid using <code>null</code> on string-based fields such as <code>CharField</code> and <code>TextField</code> because empty string values will always be stored as empty strings, not as <code>NULL</code> . For both string-based and non-string-based fields, you will also need to set <code>blank=True</code> if you wish to permit empty values in forms. If you want to accept <code>null</code> values with <code>BooleanField</code> , use <code>NullBooleanField</code> instead.
blank	If <code>True</code> , the field is allowed to be blank. Default is <code>False</code> . Note that this is different than <code>null</code> . <code>null</code> is purely database-related, whereas <code>blank</code> is validation-related.
choices	An iterable (for example, a list or tuple) consisting itself of iterables of exactly two items (for example, <code>[(A, B), (A, B) ...]</code>) to use as choices for this field. If this is given, the default form widget will be a select box with these choices instead of the standard text field. The first element in each tuple is the actual value to be set on the model, and the second element is the human readable name.

s	The second element is the human-readable name.
d b — c o l u m n	The name of the database column to use for this field. If this isn't given, Django will use the field's name.
d b — i n d e x	If <code>True</code> , a database index will be created for this field.
d b — t a b l e s p a c e	The name of the database tablespace to use for this field's index, if this field is indexed. The default is the project's <code>DEFAULT_INDEX_TABLESPACE</code> setting, if set, or the <code>db_tablespace</code> of the model, if any. If the backend doesn't support tablespaces for indexes, this option is ignored.

d e f a u l t	<p>The default value for the field. This can be a value or a callable object. If callable it will be called every time a new object is created. The default cannot be a mutable object (model instance, list, set, and others.), as a reference to the same instance of that object would be used as the default value in all new model instances.</p>
e d i t a b l e	<p>If <code>False</code>, the field will not be displayed in the admin or any other <code>ModelForm</code>. They are also skipped during model validation. Default is <code>True</code>.</p>
e r r o r — m e s s a g e s	<p>The <code>error_messages</code> argument lets you override the default messages that the field will raise. Pass in a dictionary with keys matching the error messages you want to override. Error message keys include <code>null</code>, <code>blank</code>, <code>invalid</code>, <code>invalid_choice</code>, <code>unique</code>, and <code>unique_for_date</code>.</p>
h e	

<code>help_text</code>	Extra help text to be displayed with the form widget. It's useful for documentation even if your field isn't used on a form. Note that this value is <i>not</i> HTML-escaped in automatically-generated forms. This lets you include HTML in <code>help_text</code> if you so desire.
<code>primary_key</code>	If <code>True</code> , this field is the primary key for the model. If you don't specify <code>primary_key=True</code> for any field in your model, Django will automatically add an <code>AutoField</code> to hold the primary key, so you don't need to set <code>primary_key=True</code> on any of your fields unless you want to override the default primary-key behavior. The primary key field is read-only.
<code>unique</code>	If <code>True</code> , this field must be unique throughout the table. This is enforced at the database level and by model validation. This option is valid on all field types except <code>ManyToManyField</code> , <code>OneToOneField</code> , and <code>FileField</code> .
<code>unique_for_date</code>	Set this to the name of a <code>DateField</code> or <code>DateTimeField</code> to require that this field be unique for the value of the date field. For example, if you have a field <code>title</code> that has <code>unique_for_date="pub_date"</code> , then Django wouldn't allow the entry of two records with the same

<code>o</code>	<code>title</code> and <code>pub_date</code> . This is enforced by
<code>r</code>	<code>Model.validate_unique()</code> during model validation but not at the
<code>_</code>	database level.
<code>d</code>	
<code>a</code>	
<code>t</code>	
<code>e</code>	

`u`	
`n`	
`i`	
`q`	
`u`	
`e`	
`_`	Like `unique_for_date`, but requires the field to be unique with
`f`	respect to the month.
`r`	
`_`	
`m`	
`o`	
`n`	
`t`	
`h`	
`u`	
`n`	
`i`	
`q`	
`u`	
`e`	
`_`	Like `unique_for_date`, but requires the field to be unique with
`f`	respect to the year.
`o`	
`r`	
`_`	
`v`	

y e a r	
v e r b o s e — n a m e	A human-readable name for the field. If the verbose name isn't given, Django will automatically create it using the field's attribute name, converting underscores to spaces.
v a l i d a t o r s	A list of validators to run for this field.

Table A.2: Django universal field options

Field attribute reference

Every `Field` instance contains several attributes that allow introspecting its behavior. Use these attributes instead of `isinstance` checks when you need to write code that depends on a field's functionality. These attributes can be used together with the `Model._meta` API to narrow down a search for specific field types. Custom model fields should implement these flags.

Attributes for fields

`FIELD.AUTO_CREATED`

Boolean flag that indicates if the field was automatically created, such as the `OneToOneField` used by model inheritance.

`FIELD.CONCRETE`

Boolean flag that indicates if the field has a database column associated with it.

`FIELD.HIDDEN`

Boolean flag that indicates if a field is used to back another non-hidden field's functionality (for example, the `content_type` and `object_id` fields that make up a `GenericForeignKey`). The `hidden` flag is used to distinguish what constitutes the public subset of fields on

the model from all the fields on the model.

FIELD.IS_RELATION

Boolean flag that indicates if a field contains references to one or more other models for its functionality (for example, [ForeignKey](#), [ManyToManyField](#), [OneToOneField](#), and others).

FIELD.MODEL

Returns the model on which the field is defined. If a field is defined on a superclass of a model, [model](#) will refer to the superclass, not the class of the instance.

Attributes for fields with relations

These attributes are used to query for the cardinality and other details of a relation. These attribute are present on all fields; however, they will only have meaningful values if the field is a relation type ([Field.is_relation=True](#)).

FIELD.MANY_TO_MANY

Boolean flag that is [True](#) if the field has a many-to-many relation; [False](#) otherwise. The only field included with Django where this is [True](#) is [ManyToManyField](#).

FIELD.MANY_TO_ONE

Boolean flag that is [True](#) if the field has a many-to-one

relation, such as a `ForeignKey`; `False` otherwise.

FIELD.ONE_TO_MANY

Boolean flag that is `True` if the field has a one-to-many relation, such as a `GenericRelation` or the reverse of a `ForeignKey`; `False` otherwise.

FIELD.ONE_TO_ONE

Boolean flag that is `True` if the field has a one-to-one relation, such as a `OneToOneField`; `False` otherwise.

FIELD.RELATED_MODEL

Points to the model the field relates to. For example, `Author` in `ForeignKey(Author)`. If a field has a generic relation (such as a `GenericForeignKey` or a `GenericRelation`) then `related_model` will be `None`.

Relationships

Django also defines a set of fields that represent relations.

ForeignKey

A many-to-one relationship. Requires a positional argument: the class to which the model is related. To create a recursive relationship—an object that has a many-to-one relationship with itself—use `models.ForeignKey('self')`.

If you need to create a relationship on a model that has not yet been defined, you can use the name of the model, rather than the model object itself:

```
from django.db import models

class Car(models.Model):
    manufacturer =
        models.ForeignKey('Manufacturer')
    # ...

class Manufacturer(models.Model):
    # ...
    pass
```

To refer to models defined in another application, you can explicitly specify a model with the full application label. For example, if the `Manufacturer` model above

is defined in another application called `production`, you'd need to use:

```
class Car(models.Model):
    manufacturer =
        models.ForeignKey('production.Manufacturer'
    )
```

This sort of reference can be useful when resolving circular import dependencies between two applications. A database index is automatically created on the `ForeignKey`. You can disable this by setting `db_index` to `False`.

You may want to avoid the overhead of an index if you are creating a foreign key for consistency rather than joins, or if you will be creating an alternative index like a partial or multiple column index.

DATABASE REPRESENTATION

Behind the scenes, Django appends `"_id"` to the field name to create its database column name. In the above example, the database table for the `Car` model will have a `manufacturer_id` column.

You can change this explicitly by specifying `db_column`, however, your code should never have to deal with the database column name, unless you write custom SQL. You'll always deal with the field names of your model object.

ARGUMENTS

`ForeignKey` accepts an extra set of arguments-all optional-that define the details of how the relation works.

`limit_choices_to`

Sets a limit to the available choices for this field when this field is rendered using a `ModelForm` or the admin (by default, all objects in the queryset are available to choose). Either a dictionary, a `Q` object, or a callable returning a dictionary or `Q` object can be used. For example:

```
staff_member = models.ForeignKey(User,  
limit_choices_to={'is_staff': True})
```

causes the corresponding field on the `ModelForm` to list only `Users` that have `is_staff=True`. This may be helpful in the Django admin. The callable form can be helpful, for instance, when used in conjunction with the Python `datetime` module to limit selections by date range. For example:

```
def limit_pub_date_choices():  
    return {'pub_date_lte':  
        datetime.date.utcnow()  
    }  
limit_choices_to = limit_pub_date_choices
```

If `limit_choices_to` is or returns a `Q` object, which is useful for complex queries, then it will only have an effect on the choices available in the admin when the field is not listed in `raw_id_fields` in the `ModelAdmin`

for the model.

related_name

The name to use for the relation from the related object back to this one. It's also the default value for `related_query_name` (the name to use for the reverse filter name from the target model). See the related objects documentation for a full explanation and example. Note that you must set this value when defining relations on abstract models; and when you do so some special syntax is available. If you'd prefer Django not to create a backwards relation, set `related_name` to `'+'` or end it with `'+'`. For example, this will ensure that the `User` model won't have a backwards relation to this model:

```
user = models.ForeignKey(User,  
    related_name='+')
```

related_query_name

The name to use for the reverse filter name from the target model. Defaults to the value of `related_name` if it is set, otherwise it defaults to the name of the model:

```
# Declare the ForeignKey with  
related_query_name  
class Tag(models.Model):  
    article = models.ForeignKey(Article,  
        related_name="tags",  
        related_query_name="tag")  
    name =  
        models.CharField(max_length=255)
```

```
# That's now the name of the reverse  
filter  
Article.objects.filter(tag__name="importan  
t")
```

to_field

The field on the related object that the relation is to. By default, Django uses the primary key of the related object.

db_constraint

Controls whether or not a constraint should be created in the database for this foreign key. The default is `True`, and that's almost certainly what you want; setting this to `False` can be very bad for data integrity. That said, here are some scenarios where you might want to do this:

- You have legacy data that is not valid.
- You're sharding your database.

If this is set to `False`, accessing a related object that doesn't exist will raise its `DoesNotExist` exception.

on_delete

When an object referenced by a `ForeignKey` is deleted, Django by default emulates the behavior of the SQL constraint `ON DELETE CASCADE` and also deletes the object containing the `ForeignKey`. This behavior can be overridden by specifying the `on_delete` argument. For example, if you have a nullable `ForeignKey` and you want it to be set null when the

referenced object is deleted:

```
user = models.ForeignKey(User, blank=True,  
null=True, on_delete=models.SET_NULL)
```

The possible values for `on_delete` are found in `django.db.models`:

- `CASCADE`: Cascade deletes; the default
- `PROTECT`: Prevent deletion of the referenced object by raising `ProtectedError`, a subclass of `django.db.IntegrityError`
- `SET_NULL`: Set the `ForeignKey` null; this is only possible if `null` is `True`
- `SET_DEFAULT`: Set the `ForeignKey` to its default value; a default for the `ForeignKey` must be set

swappable

Controls the migration framework's reaction if this `ForeignKey` is pointing at a swappable model. If it is `True`-the default-then if the `ForeignKey` is pointing at a model which matches the current value of `settings.AUTH_USER_MODEL` (or another swappable model setting) the relationship will be stored in the migration using a reference to the setting, not to the model directly.

You only want to override this to be `False` if you are sure your model should always point towards the swapped-in model-for example, if it is a profile model designed specifically for your custom user model. Setting it to `False` does not mean you can reference a swappable model even if it is swapped out-`False` just

means that the migrations made with this [ForeignKey](#) will always reference the exact model you specify (so it will fail hard if the user tries to run with a User model you don't support, for example). If in doubt, leave it to its default of [True](#).

ManyToManyField

A many-to-many relationship. Requires a positional argument: the class to which the model is related, which works exactly the same as it does for [ForeignKey](#), including recursive and lazy relationships. Related objects can be added, removed, or created with the field's [RelatedManager](#).

DATABASE REPRESENTATION

Behind the scenes, Django creates an intermediary join table to represent the many-to-many relationship. By default, this table name is generated using the name of the many-to-many field and the name of the table for the model that contains it.

Since some databases don't support table names above a certain length, these table names will be automatically truncated to 64 characters and a uniqueness hash will be used. This means you might see table names like [author_books_9cdf4](#); this is perfectly normal. You can manually provide the name of the join table using the [db_table](#) option.

ARGUMENTS

`ManyToManyField` accepts an extra set of arguments-all optional-that control how the relationship functions.

related_name

Same as `ForeignKey.related_name`.

related_query_name

Same as `ForeignKey.related_query_name`.

limit_choices_to

Same as `ForeignKey.limit_choices_to`.

`limit_choices_to` has no effect when used on a `ManyToManyField` with a custom intermediate table specified using the `through` parameter.

symmetrical

Only used in the definition of `ManyToManyFields` on self. Consider the following model:

```
from django.db import models

class Person(models.Model):
    friends =
        models.ManyToManyField("self")
```

When Django processes this model, it identifies that it has a `ManyToManyField` on itself, and as a result, it doesn't add a `person_set` attribute to the `Person` class. Instead, the `ManyToManyField` is assumed to be

symmetrical—that is, if I am your friend, then you are my friend.

If you do not want symmetry in many-to-many relationships with `self`, set `symmetrical` to `False`.

This will force Django to add the descriptor for the reverse relationship, allowing `ManyToManyField` relationships to be non-symmetrical.

through

Django will automatically generate a table to manage many-to-many relationships. However, if you want to manually specify the intermediary table, you can use the `through` option to specify the Django model that represents the intermediate table that you want to use.

The most common use for this option is when you want to associate extra data with a many-to-many relationship. If you don't specify an explicit `through` model, there is still an implicit `through` model class you can use to directly access the table created to hold the association. It has three fields:

- `id`: The primary key of the relation
- `<containing_model>_id`: The `id` of the model that declares the `ManyToManyField`
- `<other_model>_id`: The `id` of the model that the `ManyToManyField` points to

This class can be used to query associated records for a given model instance like a normal model.

through_fields

Only used when a custom intermediary model is specified. Django will normally determine which fields of the intermediary model to use in order to establish a many-to-many relationship automatically.

db_table

The name of the table to create for storing the many-to-many data. If this is not provided, Django will assume a default name based upon the names of the table for the model defining the relationship and the name of the field itself.

db_constraint

Controls whether or not constraints should be created in the database for the foreign keys in the intermediary table. The default is `True`, and that's almost certainly what you want; setting this to `False` can be very bad for data integrity.

That said, here are some scenarios where you might want to do this:

- You have legacy data that is not valid
- You're sharding your database

It is an error to pass both `db_constraint` and `through`.

swappable

Controls the migration framework's reaction if this `ManyToManyField` is pointing at a swappable model. If it is `True`--the default--then if the `ManyToManyField` is pointing at a model which matches the current value of `settings.AUTH_USER_MODEL` (or another swappable model setting) the relationship will be stored in the migration using a reference to the setting, not to the model directly.

You only want to override this to be `False` if you are sure your model should always point towards the swapped-in model--for example, if it is a profile model designed specifically for your custom user model. If in doubt, leave it to its default of `True`. `ManyToManyField` does not support `validators`. `null` has no effect since there is no way to require a relationship at the database level.

OneToOneField

A one-to-one relationship. Conceptually, this is similar to a `ForeignKey` with `unique=True`, but the reverse side of the relation will directly return a single object. This is most useful as the primary key of a model which extends another model in some way; multi table inheritance is implemented by adding an implicit one-to-one relation from the child model to the parent model, for example.

One positional argument is required: the class to which the model will be related. This works exactly the same as it does for `ForeignKey`, including all the options

regarding recursive and lazy relationships. If you do not specify the `related_name` argument for the `OneToOneField`, Django will use the lower-case name of the current model as default value. With the following example:

```
from django.conf import settings
from django.db import models

class MySpecialUser(models.Model):
    user =
        models.OneToOneField(settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE)
    supervisor =
        models.OneToOneField(settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
        related_name='supervisor_of')
```

your resulting `User` model will have the following attributes:

```
>>> user = User.objects.get(pk=1)
>>> hasattr(user, 'myspecialuser')
True
>>> hasattr(user, 'supervisor_of')
True
```

A `DoesNotExist` exception is raised when accessing the reverse relationship if an entry in the related table doesn't exist. For example, if a user doesn't have a supervisor designated by `MySpecialUser`:

```
>>> user.supervisor_of
Traceback (most recent call last):
...

```

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

Additionally, [OneToOneField](#) accepts all of the extra arguments accepted by [ForeignKey](#), plus one extra argument:

PARENT_LINK

When [True](#) and used in a model which inherits from another concrete model, indicates that this field should be used as the link back to the parent class, rather than the extra [OneToOneField](#) which would normally be implicitly created by subclassing. See *One-to-one relationships* in the next chapter for usage examples of [OneToOneField](#).

Model metadata options

Table A.3 is a complete list of model meta options you can give your model in its internal `class Meta`. For more detail on each meta option as well as examples, see the Django documentation at <https://docs.djangoproject.com/en/1.8/ref/models/options/>.

Option	Notes
<code>abstract</code>	If <code>abstract = True</code> , this model will be an abstract base class.
<code>app_label</code>	If a model is defined outside of an application in <code>INSTALLED_APPS</code> , it must declare which app it belongs to.
<code>db_table</code>	The name of the database table to use for the model.
<code>db_tablespace</code>	The name of the database tablespace to use for this model. The default is the project's <code>DEFAULT_TABLESPACE</code> setting, if set. If the backend doesn't support tablespaces, this option is ignored.
<code>default_related_name</code>	The name that will be used by default for the relation from a

<code>related_name</code>	related object back to this one. The default is <code><model_name>_set</code> .
<code>get_latest_by</code>	The name of an orderable field in the model, typically a <code>DateField</code> , <code>DateTimeField</code> , or <code>IntegerField</code> .
<code>managed</code>	Defaults to <code>True</code> , meaning Django will create the appropriate database tables in <code>migrate</code> or as part of migrations and remove them as part of a <code>flush</code> management command.
<code>order_with_respect_to</code>	Marks this object as orderable with respect to the given field.
<code>ordering</code>	The default ordering for the object, for use when obtaining lists of objects.
<code>permissions</code>	Extra permissions to enter into the permissions table when creating this object.
<code>default_permissions</code>	Defaults to <code>('add', 'change', 'delete')</code> .
	If <code>proxy = True</code> , a model which subclasses another model will

<code>proxy</code>	be treated as a proxy model.
<code>select_on_save</code>	Determines if Django will use the pre-1.6 <code>django.db.models.Model.save()</code> algorithm.
<code>unique_together</code>	Sets of field names that, taken together, must be unique.
<code>index_together</code>	Sets of field names that, taken together, are indexed.
<code>verbose_name</code>	A human-readable name for the object, singular.
<code>verbose_name_plural</code>	The plural name for the object.

Table A.3: Model metadata options

Appendix B. Database API Reference

Django's database API is the other half of the model API discussed in Appendix A. Once you've defined a model, you'll use this API any time you need to access the database. You've seen examples of this API in use throughout the book; this appendix explains all the various options in detail.

Throughout this appendix I'll refer to the following models, which comprise a Weblog application:

```
from django.db import models
class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()
    def __str__(self):
        return self.name
class Author(models.Model):
    name = models.CharField(max_length=50)
    email = models.EmailField()
    def __str__(self):
        return self.name
class Entry(models.Model):
    blog = models.ForeignKey(Blog)
    headline = models.CharField(max_length=255)
    body_text = models.TextField()
    pub_date = models.DateField()
    mod_date = models.DateField()
    authors = models.ManyToManyField(Author)
    n_comments = models.IntegerField()
    n_pingbacks = models.IntegerField()
    rating = models.IntegerField()
    def __str__(self):
        return self.headline
```

Creating objects

To represent database-table data in Python objects, Django uses an intuitive system: a model class represents a database table, and an instance of that class represents a particular record in the database table.

To create an object, instantiate it using keyword arguments to the model class, then call `save()` to save it to the database.

Assuming models live in a file `mysite/blog/models.py`, here's an example:

```
>>> from blog.models import Blog >>> b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.') >>> b.save()
```

This performs an `INSERT` SQL statement behind the scenes. Django doesn't hit the database until you explicitly call `save()`.

The `save()` method has no return value.

To create and save an object in a single step, use the `create()` method.

Saving changes to objects

To save changes to an object that's already in the database, use `save()`.

Given a `Blog` instance `b5` that has already been saved to the database, this example changes its name and updates its record in the database: `>>> b5.name = 'New name' >>> b5.save()`

This performs an `UPDATE` SQL statement behind the scenes. Django doesn't hit the database until you explicitly call `save()`.

Saving ForeignKey and ManyToManyField fields

Updating a `ForeignKey` field works exactly the same way as saving a normal field—simply assign an object of the right type to the field in question. This example updates the `blog` attribute of an `Entry` instance `entry`, assuming appropriate instances of `Entry`, and `Blog` are already saved to the database (so we can retrieve them below): `>>> from blog.models import Entry >>> entry = Entry.objects.get(pk=1) >>> cheese_blog = Blog.objects.get(name="Cheddar Talk") >>> entry.blog = cheese_blog >>> entry.save()`

Updating a [ManyToManyField](#) works a little differently—use the `add()` method on the field to add a record to the relation. This example adds the `Author` instance `joe` to the `entry` object:

```
>>> from blog.models import Author
>>> joe = Author.objects.create(name="Joe")
>>> entry.authors.add(joe)
```

To add multiple records to a [ManyToManyField](#) in one go, include multiple arguments in the call to `add()`, like this:

```
>>> john = Author.objects.create(name="John")
>>> paul = Author.objects.create(name="Paul") >>>
george = Author.objects.create(name="George") >>>
ringo = Author.objects.create(name="Ringo") >>>
entry.authors.add(john, paul, george, ringo)
```

Django will complain if you try to assign or add an object of the wrong type.

Retrieving objects

To retrieve objects from your database, construct a [QuerySet](#) via a [Manager](#) on your model class.

A [QuerySet](#) represents a collection of objects from your database. It can have zero, one or many filters. Filters narrow down the query results based on the given parameters. In SQL terms, a [QuerySet](#) equates to a [SELECT](#) statement, and a filter is a limiting clause such as [WHERE](#) or [LIMIT](#).

You get a [QuerySet](#) by using your model's [Manager](#). Each model has at least one [Manager](#), and it's called [objects](#) by default. Access it directly via the model class, like so:

```
>>> Blog.objects
<django.db.models.manager.Manager object
at ...>
>>> b = Blog(name='Foo', tagline='Bar')
>>> b.objects
Traceback:
...
AttributeError: "Manager isn't accessible
via Blog instances."
```

Retrieving all objects

The simplest way to retrieve objects from a table is to get all of them. To do this, use the [all\(\)](#) method on a

Manager:

```
>>> all_entries = Entry.objects.all()
```

The `all()` method returns a `QuerySet` of all the objects in the database.

Retrieving specific objects with filters

The `QuerySet` returned by `all()` describes all objects in the database table. Usually, though, you'll need to select only a subset of the complete set of objects.

To create such a subset, you refine the initial `QuerySet`, adding filter conditions. The two most common ways to refine a `QuerySet` are:

- `filter(**kwargs)`. Returns a new `QuerySet` containing objects that match the given lookup parameters.
- `exclude(**kwargs)`. Returns a new `QuerySet` containing objects that do not match the given lookup parameters.

The lookup parameters (`**kwargs` in the above function definitions) should be in the format described in *Field lookups* later in this chapter.

CHAINING FILTERS

The result of refining a `QuerySet` is itself a `QuerySet`, so it's possible to chain refinements together. For example:

```
>>> Entry.objects.filter(
...     headline__startswith='What'
... ).exclude(
...
pub_date__gte=datetime.date.today()
... ).filter(pub_date__gte=datetime(2005,
1, 30)
... )
```

This takes the initial `QuerySet` of all entries in the database, adds a filter, then an exclusion, then another filter. The final result is a `QuerySet` containing all entries with a headline that starts with `What`, that were published between January 30, 2005, and the current day.

Filtered querysets are unique

Each time you refine a `QuerySet`, you get a brand-new `QuerySet` that is in no way bound to the previous `QuerySet`. Each refinement creates a separate and distinct `QuerySet` that can be stored, used, and reused.

Example:

```
>>> q1 =
Entry.objects.filter(headline__startswith=
"What")
>>> q2 =
q1.exclude(pub_date__gte=datetime.date.tod
ay())
>>> q3 =
q1.filter(pub_date__gte=datetime.date.toda
y())
```

These three `QuerySets` are separate. The first is a base `QuerySet` containing all entries that contain a headline starting with What. The second is a subset of the first, with an additional criterion that excludes records whose `pub_date` is today or in the future. The third is a subset of the first, with an additional criterion that selects only the records whose `pub_date` is today or in the future. The initial `QuerySet` (`q1`) is unaffected by the refinement process.

QUERYSETS ARE LAZY

`QuerySets` are lazy—the act of creating a `QuerySet` doesn't involve any database activity. You can stack filters together all day long, and Django won't actually run the query until the `QuerySet` is evaluated. Take a look at this example:

```
>>> q =
Entry.objects.filter(headline__startswith=
"What")
>>> q =
q.filter(pub_date__lte=datetime.date.today()
())
>>> q =
q.exclude(body_text__icontains="food")
>>> print(q)
```

Though this looks like three database hits, in fact it hits the database only once, at the last line (`print(q)`). In general, the results of a `QuerySet` aren't fetched from the database until you ask for them. When you do, the `QuerySet` is evaluated by accessing the database.

Retrieving a single object with get

`filter()` will always give you a `QuerySet`, even if only a single object matches the query-in this case, it will be a `QuerySet` containing a single element.

If you know there is only one object that matches your query, you can use the `get()` method on a `Manager` which returns the object directly:

```
>>> one_entry = Entry.objects.get(pk=1)
```

You can use any query expression with `get()`, just like with `filter()`-again, see *Field lookups* in the next section of this chapter.

Note that there is a difference between using `get()`, and using `filter()` with a slice of `[0]`. If there are no results that match the query, `get()` will raise a `DoesNotExist` exception. This exception is an attribute of the model class that the query is being performed on-so in the code above, if there is no `Entry` object with a primary key of 1, Django will raise `Entry.DoesNotExist`.

Similarly, Django will complain if more than one item matches the `get()` query. In this case, it will raise `MultipleObjectsReturned`, which again is an attribute of the model class itself.

Other queryset methods

Most of the time you'll use `all()`, `get()`, `filter()`, and `exclude()` when you need to look up objects from the database. However, that's far from all there is; see the [QuerySet API Reference](#) at <https://docs.djangoproject.com/en/1.8/ref/models/querysets/>, for a complete list of all the various `QuerySet` methods.

Limiting querysets

Use a subset of Python's array-slicing syntax to limit your `QuerySet` to a certain number of results. This is the equivalent of SQL's `LIMIT` and `OFFSET` clauses.

For example, this returns the first 5 objects (`LIMIT 5`):

```
>>> Entry.objects.all()[:5]
```

This returns the sixth through tenth objects (`OFFSET 5 LIMIT 5`):

```
>>> Entry.objects.all()[5:10]
```

Negative indexing (that is, `Entry.objects.all()[-1]`) is not supported.

Generally, slicing a `QuerySet` returns a new `QuerySet`-it doesn't evaluate the query. An exception is if you use the step parameter of Python slice syntax. For

example, this would actually execute the query in order to return a list of every *second* object of the first 10:

```
>>> Entry.objects.all()[::2]
```

To retrieve a *single* object rather than a list (for example, `SELECT foo FROM bar LIMIT 1`), use a simple index instead of a slice.

For example, this returns the first `Entry` in the database, after ordering entries alphabetically by headline:

```
>>> Entry.objects.order_by('headline')[0]
```

This is roughly equivalent to:

```
>>> Entry.objects.order_by('headline')[0:1].get()
```

Note, however, that the first of these will raise `IndexError` while the second will raise `DoesNotExist` if no objects match the given criteria. See `get()` for more details.

Field lookups

Field lookups are how you specify the meat of an SQL `WHERE` clause. They're specified as keyword arguments to the `QuerySet` methods `filter()`, `exclude()`, and `get()`. Basic lookups keyword arguments take the form

`field__lookuptype=value`. (That's a double-underscore). For example:

```
>>>  
Entry.objects.filter(pub_date__lte='2006-  
01-01')
```

translates (roughly) into the following SQL:

```
SELECT * FROM blog_entry WHERE pub_date <=  
'2006-01-01';
```

The field specified in a lookup has to be the name of a model field. There's one exception though, in case of a [ForeignKey](#) you can specify the field name suffixed with `_id`. In this case, the value parameter is expected to contain the raw value of the foreign model's primary key. For example:

```
>>> Entry.objects.filter(blog_id=4)
```

If you pass an invalid keyword argument, a lookup function will raise [TypeError](#).

The complete list of field lookups are:

- `exact`
- `iexact`
- `contains`
- `icontains`
- `in`
- `gt`

- `gte`
- `lt`
- `lte`
- `startswith`
- `istartswith`
- `endswith`
- `iendswith`
- `range`
- `year`
- `month`
- `day`
- `week_day`
- `hour`
- `minute`
- `second`
- `isnull`
- `search`
- `regex`
- `iregex`

A complete reference, including examples for each field lookup can be found in the field lookup reference at <https://docs.djangoproject.com/en/1.8/ref/models/querysets/#field-lookups>.

Lookups that span relationships

Django offers a powerful and intuitive way to follow relationships in lookups, taking care of the SQL `JOINS`

for you automatically, behind the scenes. To span a relationship, just use the field name of related fields across models, separated by double underscores, until you get to the field you want.

This example retrieves all `Entry` objects with a `Blog` whose `name` is '`Beatles Blog`':

```
>>>
Entry.objects.filter(blog__name='Beatles
Blog')
```

This spanning can be as deep as you'd like.

It works backwards, too. To refer to a reverse relationship, just use the lowercase name of the model.

This example retrieves all `Blog` objects which have at least one `Entry` whose `headline` contains '`Lennon`':

```
>>>
Blog.objects.filter(entry__headline__conta
ins='Lennon')
```

If you are filtering across multiple relationships and one of the intermediate models doesn't have a value that meets the filter condition, Django will treat it as if there is an empty (all values are `NULL`), but valid, object there. All this means is that no error will be raised. For example, in this filter:

```
Blog.objects.filter(entry__authors__name='
Lennon')
```

(if there was a related [Author](#) model), if there was no [author](#) associated with an entry, it would be treated as if there was also no [name](#) attached, rather than raising an error because of the missing [author](#). Usually this is exactly what you want to have happen. The only case where it might be confusing is if you are using [isnull](#).

Thus:

```
Blog.objects.filter(entry_authors_name__  
    isnull=True)
```

will return [Blog](#) objects that have an empty [name](#) on the [author](#) and also those which have an empty [author](#) on the [entry](#). If you don't want those latter objects, you could write:

```
Blog.objects.filter(entry_authors_isnull  
    =False,  
        entry_authors_name_isnull=True)
```

SPANNING MULTI-VALUED RELATIONSHIPS

When you are filtering an object based on a [ManyToManyField](#) or a reverse [ForeignKey](#), there are two different sorts of filter you may be interested in. Consider the [Blog/Entry](#) relationship ([Blog](#) to [Entry](#) is a one-to-many relation). We might be interested in finding blogs that have an entry which has both [Lennon](#) in the headline and was published in 2008.

Or we might want to find blogs that have an entry with `Lennon` in the headline as well as an entry that was published in 2008. Since there are multiple entries associated with a single `Blog`, both of these queries are possible and make sense in some situations.

The same type of situation arises with a `ManyToManyField`. For example, if an `Entry` has a `ManyToManyField` called `tags`, we might want to find entries linked to tags called `music` and `bands` or we might want an entry that contains a tag with a name of `music` and a status of `public`.

To handle both of these situations, Django has a consistent way of processing `filter()` and `exclude()` calls. Everything inside a single `filter()` call is applied simultaneously to filter out items matching all those requirements.

Successive `filter()` calls further restrict the set of objects, but for multi-valued relations, they apply to any object linked to the primary model, not necessarily those objects that were selected by an earlier `filter()` call.

That may sound a bit confusing, so hopefully an example will clarify. To select all blogs that contain entries with both `Lennon` in the headline and that were published in 2008 (the same entry satisfying both conditions), we would write:

```
Blog.objects.filter(entry_headline__conta  
+====+====+
```

```
    this='Lennon',
    entry__pub_date__year=2008)
```

To select all blogs that contain an entry with `Lennon` in the headline as well as an entry that was published in 2008, we would write:

```
Blog.objects.filter(entry__headline__contains='Lennon').filter(
    entry__pub_date__year=2008)
```

Suppose there is only one blog that had both entries containing `Lennon` and entries from 2008, but that none of the entries from 2008 contained `Lennon`. The first query would not return any blogs, but the second query would return that one blog.

In the second example, the first filter restricts the queryset to all those blogs linked to entries with `Lennon` in the headline. The second filter restricts the set of blogs *further* to those that are also linked to entries that were published in 2008.

The entries selected by the second filter may or may not be the same as the entries in the first filter. We are filtering the `Blog` items with each filter statement, not the `Entry` items.

All of this behavior also applies to `exclude()`: all the conditions in a single `exclude()` statement apply to a single instance (if those conditions are talking about the same multi-valued relation). Conditions in subsequent

`filter()` or `exclude()` calls that refer to the same relation may end up filtering on different linked objects.

Filters can reference fields on the model

In the examples given so far, we have constructed filters that compare the value of a model field with a constant. But what if you want to compare the value of a model field with another field on the same model?

Django provides `F expressions` to allow such comparisons. Instances of `F()` act as a reference to a model field within a query. These references can then be used in query filters to compare the values of two different fields on the same model instance.

For example, to find a list of all blog entries that have had more comments than pingbacks, we construct an `F()` object to reference the pingback count, and use that `F()` object in the query:

```
>>> from django.db.models import F
>>>
Entry.objects.filter(n_comments__gt=F('n_pingbacks'))
```

Django supports the use of addition, subtraction, multiplication, division, modulo, and power arithmetic with `F()` objects, both with constants and with other `F()` objects. To find all the blog entries with more than *twice*

as many comments as pingbacks, we modify the query:

```
>>>
Entry.objects.filter(n_comments__gt=F('n_pingbacks') * 2)
```

To find all the entries where the rating of the entry is less than the sum of the pingback count and comment count, we would issue the query:

```
>>>
Entry.objects.filter(rating__lt=F('n_comments') + F('n_pingbacks'))
```

You can also use the double underscore notation to span relationships in an `F()` object. An `F()` object with a double underscore will introduce any joins needed to access the related object.

For example, to retrieve all the entries where the author's name is the same as the blog name, we could issue the query:

```
>>>
Entry.objects.filter(authors__name=F('blog__name'))
```

For date and date/time fields, you can add or subtract a `timedelta` object. The following would return all entries that were modified more than 3 days after they were published:

```
>>> from datetime import timedelta
```

```
>>>
Entry.objects.filter(mod_date__gt=F('pub_date') + timedelta(days=3))
```

The `F()` objects support bitwise operations by `.bitand()` and `.bitor()`, for example:

```
>>> F('somefield').bitand(16)
```

The pk lookup shortcut

For convenience, Django provides a `pk` lookup shortcut, which stands for primary key.

In the example `Blog` model, the primary key is the `id` field, so these three statements are equivalent:

```
>>> Blog.objects.get(id__exact=14) #
Explicit form
>>> Blog.objects.get(id=14) # __exact is implied
>>> Blog.objects.get(pk=14) # pk implies id__exact
```

The use of `pk` isn't limited to `__exact` queries-any query term can be combined with `pk` to perform a query on the primary key of a model:

```
# Get blogs entries with id 1, 4 and 7
>>> Blog.objects.filter(pk__in=[1,4,7])
# Get all blog entries with id > 14
>>> Blog.objects.filter(pk__gt=14)
```

`pk` lookups also work across joins. For example, these

three statements are equivalent:

```
>>>
Entry.objects.filter(blog_id_exact=3) #
Explicit form
>>> Entry.objects.filter(blog_id=3)
# __exact is implied
>>> Entry.objects.filter(blog_pk=3)
# __pk implies __id__exact
```

Escaping percent signs and underscores in LIKE statements

The field lookups that equate to `LIKE` SQL statements (`iexact`, `contains`, `icontains`, `startswith`, `istartswith`, `endswith`, and `iendswith`) will automatically escape the two special characters used in `LIKE` statements—the percent sign and the underscore. (In a `LIKE` statement, the percent sign signifies a multiple-character wildcard and the underscore signifies a single-character wildcard.)

This means things should work intuitively, so the abstraction doesn't leak. For example, to retrieve all the entries that contain a percent sign, just use the percent sign as any other character:

```
>>>
Entry.objects.filter(headline_contains='%
')
```

Django takes care of the quoting for you; the resulting SQL will look something like this:

```
SELECT ... WHERE headline LIKE '%\%%';
```

Same goes for underscores. Both percentage signs and underscores are handled for you transparently.

Caching and querysets

Each `QuerySet` contains a cache to minimize database access. Understanding how it works will allow you to write the most efficient code.

In a newly created `QuerySet`, the cache is empty. The first time a `QuerySet` is evaluated-and, hence, a database query happens-Django saves the query results in the `QuerySet` class' cache and returns the results that have been explicitly requested (for example, the next element, if the `QuerySet` is being iterated over). Subsequent evaluations of the `QuerySet` reuse the cached results.

Keep this caching behavior in mind, because it may bite you if you don't use your `QuerySet` correctly. For example, the following will create two `QuerySet`, evaluate them, and throw them away:

```
>>> print([e.headline for e in
Entry.objects.all()])
>>> print([e.pub_date for e in
Entry.objects.all()])
```

That means the same database query will be executed twice, effectively doubling your database load. Also,

there's a possibility the two lists may not include the same database records, because an `Entry` may have been added or deleted in the split second between the two requests.

To avoid this problem, simply save the `QuerySet` and reuse it:

```
>>> queryset = Entry.objects.all()
>>> print([p.headline for p in queryset])
# Evaluate the query set.
>>> print([p.pub_date for p in queryset])
# Re-use the cache from the evaluation.
```

WHEN QUERYSETS ARE NOT CACHED

Querysets do not always cache their results. When evaluating only *part* of the queryset, the cache is checked, but if it is not populated then the items returned by the subsequent query are not cached. Specifically, this means that limiting the queryset using an array slice or an index will not populate the cache.

For example, repeatedly getting a certain index in a queryset object will query the database each time:

```
>>> queryset = Entry.objects.all()
>>> print queryset[5] # Queries the
database
>>> print queryset[5] # Queries the
database again
```

However, if the entire queryset has already been evaluated, the cache will be checked instead:

```
>>> queryset = Entry.objects.all()
>>> [entry for entry in queryset] #
    Queries the database
>>> print queryset[5] # Uses cache
>>> print queryset[5] # Uses cache
```

Here are some examples of other actions that will result in the entire queryset being evaluated and therefore populate the cache:

```
>>> [entry for entry in queryset]
>>> bool(queryset)
>>> entry in queryset
>>> list(queryset)
```

Complex lookups with Q objects

Keyword argument queries-in `filter()`, and others.- are ANDed together. If you need to execute more complex queries (for example, queries with `OR` statements), you can use `Q objects`.

A `Q object (django.db.models.Q)` is an object used to encapsulate a collection of keyword arguments. These keyword arguments are specified as in Field lookups above.

For example, this `Q` object encapsulates a single `LIKE` query:

```
from django.db.models import Q
Q(question__startswith='What')
```

`Q` objects can be combined using the `&` and `|` operators. When an operator is used on two `Q` objects, it yields a new `Q` object.

For example, this statement yields a single `Q` object that represents the OR of two "`question__startswith`" queries:

```
Q(question__startswith='Who') |
Q(question__startswith='What')
```

This is equivalent to the following SQL `WHERE` clause:

```
WHERE question LIKE 'Who%' OR question  
LIKE 'what%'
```

You can compose statements of arbitrary complexity by combining `Q` objects with the `&` and `|` operators and use parenthetical grouping. Also, `Q` objects can be negated using the `~` operator, allowing for combined lookups that combine both a normal query and a negated (`NOT`) query:

```
Q(question__startswith='Who') |  
~Q(pub_date__year=2005)
```

Each lookup function that takes keyword-arguments (for example, `filter()`, `exclude()`, `get()`) can also be passed one or more `Q` objects as positional (not-named) arguments. If you provide multiple `Q` object arguments to a lookup function, the arguments will be ANDed together. For example:

```
Poll.objects.get(  
Q(question__startswith='Who'), Q(pub_date=date(2005,  
5, 2)) | Q(pub_date=date(2005, 5, 6)) )
```

... roughly translates into the SQL:

```
SELECT * from polls WHERE question LIKE  
'Who%'  
    AND (pub_date = '2005-05-02' OR  
          pub_date = '2005-05-06')
```

Lookup functions can mix the use of `Q` objects and keyword arguments. All arguments provided to a lookup function (be the keyword arguments or `Q` objects) are

ANDed together. However, if a `Q` object is provided, it must precede the definition of any keyword arguments. For example: `Poll.objects.get(Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)), question__startswith='Who')`

... would be a valid query, equivalent to the previous example; but:

```
# INVALID QUERY
Poll.objects.get(
    question__startswith='Who',
    Q(pub_date=date(2005, 5, 2)) |
    Q(pub_date=date(2005, 5, 6)))
```

... would not be valid.

Comparing objects

To compare two model instances, just use the standard Python comparison operator, the double equals sign: `==`. Behind the scenes, that compares the primary key values of two models.

Using the [Entry](#) example above, the following two statements are equivalent:

```
>>> some_entry == other_entry  
>>> some_entry.id == other_entry.id
```

If a model's primary key isn't called `id`, no problem. Comparisons will always use the primary key, whatever it's called. For example, if a model's primary key field is called `name`, these two statements are equivalent:

```
>>> some_obj == other_obj  
>>> some_obj.name == other_obj.name
```

Deleting objects

The delete method, conveniently, is named `delete()`. This method immediately deletes the object and has no return value. Example:

```
e.delete()
```

You can also delete objects in bulk. Every `QuerySet` has a `delete()` method, which deletes all members of that `QuerySet`.

For example, this deletes all `Entry` objects with a `pub_date` year of 2005:

```
Entry.objects.filter(pub_date__year=2005).  
delete()
```

Keep in mind that this will, whenever possible, be executed purely in SQL, and so the `delete()` methods of individual object instances will not necessarily be called during the process. If you've provided a custom `delete()` method on a model class and want to ensure that it is called, you will need to manually delete instances of that model (for example, by iterating over a `QuerySet` and calling `delete()` on each object individually) rather than using the bulk `delete()` method of a `QuerySet`.

When Django deletes an object, by default it emulates the behavior of the SQL constraint `ON DELETE CASCADE`-in other words, any objects which had foreign keys pointing at the object to be deleted will be deleted along with it. For example:

```
b = Blog.objects.get(pk=1)
# This will delete the Blog and all of its
Entry objects.
b.delete()
```

This cascade behavior is customizable via the `on_delete` argument to the `ForeignKey`.

Note that `delete()` is the only `QuerySet` method that is not exposed on a `Manager` itself. This is a safety mechanism to prevent you from accidentally requesting `Entry.objects.delete()`, and deleting *all* the entries. If you *do* want to delete all the objects, then you have to explicitly request a complete query set:

```
Entry.objects.all().delete()
```

Copying model instances

Although there is no built-in method for copying model instances, it is possible to easily create new instance with all fields' values copied. In the simplest case, you can just set `pk` to `None`. Using our blog example:

```
blog = Blog(name='My blog', tagline='Blogging is easy')
blog.save() # blog.pk == 1
blog.pk = None
blog.save() #
blog.pk == 2
```

Things get more complicated if you use inheritance.
Consider a subclass of `Blog`:

```
class ThemeBlog(Blog):
    theme =
    models.CharField(max_length=200)

    django_blog = ThemeBlog(name='Django',
                           tagline='Django is easy',
                           theme='python')
    django_blog.save() # django_blog.pk == 3
```

Due to how inheritance works, you have to set both `pk` and `id` to `None`:

```
djang_blog.pk = None
djang_blog.id = None
djang_blog.save() # djang_blog.pk == 4
```

This process does not copy related objects. If you want to copy relations, you have to write a little bit more code.

In our example, `Entry` has a many to many field to `Author`:

```
entry = Entry.objects.all()[0] # some previous entry
old_authors = entry.authors.all()
entry.pk = None
entry.save()
entry.authors = old_authors # saves new many2many relations
```

Updating multiple objects at once

Sometimes you want to set a field to a particular value for all the objects in a [QuerySet](#). You can do this with the `update()` method. For example:

```
# Update all the headlines with pub_date
# in 2007.
Entry.objects.filter(pub_date__year=2007).
update(headline='Everything is the same')
```

You can only set non-relation fields and [ForeignKey](#) fields using this method. To update a non-relation field, provide the new value as a constant. To update [ForeignKey](#) fields, set the new value to be the new model instance you want to point to. For example:

```
>>> b = Blog.objects.get(pk=1)
# Change every Entry so that it belongs to
# this Blog.
>>> Entry.objects.all().update(blog=b)
```

The `update()` method is applied instantly and returns the number of rows matched by the query (which may not be equal to the number of rows updated if some rows already have the new value).

The only restriction on the [QuerySet](#) that is updated is that it can only access one database table, the model's

main table. You can filter based on related fields, but you can only update columns in the model's main table.

Example:

```
>>> b = Blog.objects.get(pk=1)
# Update all the headlines belonging to
this Blog.
>>>
Entry.objects.select_related().filter(blog
=b).update
(headline='Everything is the same')
```

Be aware that the `update()` method is converted directly to an SQL statement. It is a bulk operation for direct updates. It doesn't run any `save()` methods on your models, or emit the `pre_save` or `post_save` signals (which are a consequence of calling `save()`), or honor the `auto_now` field option. If you want to save every item in a `QuerySet` and make sure that the `save()` method is called on each instance, you don't need any special function to handle that. Just loop over them and call `save()`:

```
for item in my_queryset:
    item.save()
```

Calls to update can also use `F expressions` to update one field based on the value of another field in the model. This is especially useful for incrementing counters based upon their current value. For example, to increment the pingback count for every entry in the blog:

```
>>>
```

```
Entry.objects.all().update(n_pingbacks=F('n_pingbacks') + 1)
```

However, unlike `F()` objects in filter and exclude clauses, you can't introduce joins when you use `F()` objects in an update—you can only reference fields local to the model being updated. If you attempt to introduce a join with an `F()` object, a `FieldError` will be raised:

```
# THIS WILL RAISE A FieldError
>>>
Entry.objects.update(headline=F('blog__name'))
```

Related objects

When you define a relationship in a model (that is, a `ForeignKey`, `OneToOneField`, or `ManyToManyField`), instances of that model will have a convenient API to access the related object(s).

Using the models at the top of this page, for example, an `Entry` object `e` can get its associated `Blog` object by accessing the `blog` attribute: `e.blog`.

(Behind the scenes, this functionality is implemented by Python descriptors. This shouldn't really matter to you, but I point it out here for the curious.)

Django also creates API accessors for the other side of the relationship—the link from the related model to the model that defines the relationship. For example, a `Blog` object `b` has access to a list of all related `Entry` objects via the `entry_set` attribute: `b.entry_set.all()`.

All examples in this section use the sample `Blog`, `Author` and `Entry` models defined at the top of this page.

One-to-many relationships

FORWARD

If a model has a `ForeignKey`, instances of that model will have access to the related (foreign) object via a simple attribute of the model. For example:

```
>>> e = Entry.objects.get(id=2)
>>> e.blog # Returns the related Blog
object.
```

You can get and set via a foreign-key attribute. As you may expect, changes to the foreign key aren't saved to the database until you call `save()`. Example:

```
>>> e = Entry.objects.get(id=2)
>>> e.blog = some_blog
>>> e.save()
```

If a `ForeignKey` field has `null=True` set (that is, it allows `NULL` values), you can assign `None` to remove the relation. Example:

```
>>> e = Entry.objects.get(id=2)
>>> e.blog = None
>>> e.save() # "UPDATE blog_entry SET
blog_id = NULL ...;"
```

Forward access to one-to-many relationships is cached the first time the related object is accessed. Subsequent accesses to the foreign key on the same object instance are cached. Example:

```
>>> e = Entry.objects.get(id=2)
>>> print(e.blog) # Hits the database to
retrieve the associated Blog.
>>> print(e.blog) # Doesn't hit the
```

```
database; uses cached version.
```

Note that the `select_related()` `QuerySet` method recursively prepopulates the cache of all one-to-many relationships ahead of time. Example:

```
>>> e =
Entry.objects.select_related().get(id=2)
>>> print(e.blog) # Doesn't hit the
database; uses cached version.
>>> print(e.blog) # Doesn't hit the
database; uses cached version.
```

FOLLOWING RELATIONSHIPS BACKWARD

If a model has a `ForeignKey`, instances of the foreign-key model will have access to a `Manager` that returns all instances of the first model. By default, this `Manager` is named `foo_set`, where `foo` is the source model name, lowercased. This `Manager` returns `QuerySets`, which can be filtered and manipulated as described in the Retrieving objects section above.

Example:

```
>>> b = Blog.objects.get(id=1)
>>> b.entry_set.all() # Returns all Entry
objects related to Blog.
# b.entry_set is a Manager that returns
QuerySets.
>>>
b.entry_set.filter(headline__contains='Len
non')
>>> b.entry_set.count()
```

You can override the `foo_set` name by setting the `related_name` parameter in the `ForeignKey` definition. For example, if the `Entry` model was altered to `blog = ForeignKey(Blog, related_name='entries')`, the above example code would look like this:

```
>>> b = Blog.objects.get(id=1)
>>> b.entries.all() # Returns all Entry
objects related to Blog.
# b.entries is a Manager that returns
QuerySets.
>>>
b.entries.filter(headline__contains='Lenno
n')
>>> b.entries.count()
```

USING A CUSTOM REVERSE MANAGER

By default, the `RelatedManager` used for reverse relations is a subclass of the default manager for that model. If you would like to specify a different manager for a given query you can use the following syntax:

```
from django.db import models

class Entry(models.Model):
    ...
    objects = models.Manager() # Default
    Manager
    entries = EntryManager()    # Custom
    Manager

b = Blog.objects.get(id=1)
b.entry_set(manager='entries').all()
```

If `EntryManager` performed default filtering in its `get_queryset()` method, that filtering would apply to the `all()` call.

Of course, specifying a custom reverse manager also enables you to call its custom methods:

```
b.entry_set(manager='entries').is_published()
```

ADDITIONAL METHODS TO HANDLE RELATED OBJECTS

In addition to the `QuerySet` methods defined in *Retrieving objects* earlier, the `ForeignKey Manager` has additional methods used to handle the set of related objects. A synopsis of each is as follows (complete details can be found in the related objects reference at <https://docs.djangoproject.com/en/1.8/ref/models/relations/#related-objects-reference>):

- `add(obj1, obj2, ...)` Adds the specified model objects to the related object set
- `create(**kwargs)` Creates a new object, saves it and puts it in the related object set. Returns the newly created object
- `remove(obj1, obj2, ...)` Removes the specified model objects from the related object set
- `clear()` Removes all objects from the related object set
- `set(objs)` Replace the set of related objects

To assign the members of a related set in one fell swoop, just assign to it from any iterable object. The

iterable can contain object instances, or just a list of primary key values. For example:

```
b = Blog.objects.get(id=1)
b.entry_set = [e1, e2]
```

In this example, `e1` and `e2` can be full Entry instances, or integer primary key values.

If the `clear()` method is available, any pre-existing objects will be removed from the `entry_set` before all objects in the iterable (in this case, a list) are added to the set. If the `clear()` method is *not* available, all objects in the iterable will be added without removing any existing elements.

Each reverse operation described in this section has an immediate effect on the database. Every addition, creation and deletion is immediately and automatically saved to the database.

Many-to-many relationships

Both ends of a many-to-many relationship get automatic API access to the other end. The API works just as a backward one-to-many relationship, above.

The only difference is in the attribute naming: The model that defines the `ManyToManyField` uses the attribute name of that field itself, whereas the reverse model uses the lowercased model name of the original model, plus

'`_set`' (just like reverse one-to-many relationships).

An example makes this easier to understand:

```
e = Entry.objects.get(id=3)
e.authors.all() # Returns all Author
objects for this Entry.
e.authors.count()
e.authors.filter(name__contains='John')

a = Author.objects.get(id=5)
a.entry_set.all() # Returns all Entry
objects for this Author.
```

Like `ForeignKey`, `ManyToManyField` can specify `related_name`. In the above example, if the `ManyToManyField` in `Entry` had specified `related_name='entries'`, then each `Author` instance would have an `entries` attribute instead of `entry_set`.

One-to-one relationships

One-to-one relationships are very similar to many-to-one relationships. If you define a `OneToOneField` on your model, instances of that model will have access to the related object via a simple attribute of the model.

For example:

```
class EntryDetail(models.Model):
    entry = models.OneToOneField(Entry)
    details = models.TextField()
```

```
ed = EntryDetail.objects.get(id=2)
ed.entry # Returns the related Entry
object.
```

The difference comes in reverse queries. The related model in a one-to-one relationship also has access to a [Manager](#) object, but that [Manager](#) represents a single object, rather than a collection of objects:

```
e = Entry.objects.get(id=2)
e.entrydetail # returns the related
EntryDetail object
```

If no object has been assigned to this relationship, Django will raise a [DoesNotExist](#) exception.

Instances can be assigned to the reverse relationship in the same way as you would assign the forward relationship:

```
e.entrydetail = ed
```

Queries over related objects

Queries involving related objects follow the same rules as queries involving normal value fields. When specifying the value for a query to match, you may use either an object instance itself, or the primary key value for the object.

For example, if you have a Blog object `b` with `id=5`, the following three queries would be identical:

```
Entry.objects.filter(blog=b) # Query using
object instance
Entry.objects.filter(blog=b.id) # Query
using id from instance
Entry.objects.filter(blog=5) # Query using
id directly
```

Falling back to raw SQL

If you find yourself needing to write an SQL query that is too complex for Django's database-mapper to handle, you can fall back on writing SQL by hand.

Finally, it's important to note that the Django database layer is merely an interface to your database. You can access your database via other tools, programming languages or database frameworks; there's nothing Django-specific about your database.

Appendix C. Generic View Reference

Chapter 10, Generic Views, introduced generic views but left out some of the gory details. This appendix describes each generic view along with a summary of options each view can take. Be sure to read Chapter 10, Generic Views, before trying to understand the reference material that follows. You might want to refer back to the [Book](#), [Publisher](#), and [Author](#) objects defined in that chapter; the examples that follow use these models. If you want to dig deeper into more advanced generic view topics (like using mixins with the class-based views), see the Django Project website at <https://docs.djangoproject.com/en/1.8/topics/class-based-views/>.

Common arguments to generic views

Most of these views take a large number of arguments that can change the generic view's behavior. Many of these arguments work the same across multiple views. *Table C.1* describes each of these common arguments; anytime you see one of these arguments in a generic view's argument list, it will work as described in the table.

Argument	Description
<code>allow_empty</code>	A Boolean specifying whether to display the page if no objects are available. If this is <code>False</code> and no objects are available, the view will raise a 404 error instead of displaying an empty page. By default, this is <code>True</code> .
<code>context_processors</code>	A list of additional template-context processors (besides the defaults) to apply to the view's template. See Chapter 9, Advanced Models , for information on template context processors.
<code>extra_context</code>	A dictionary of values to add to the template context. By default, this is an empty dictionary. If a value in the dictionary is callable, the generic view will call it just before rendering the template.
<code>mime_type</code>	The MIME type to use for the resulting document. It defaults to the value of the <code>DEFAULT_MIME_TYPE</code> setting, which is <code>text/html</code> if you haven't changed it.
<code>queryset</code>	A <code>QuerySet</code> (that is, something like <code>Author.objects.all()</code>) to read objects from. See Appendix B for more information about <code>QuerySet</code> objects. Most generic views require this argument.
<code>template_loader</code>	The template loader to use when loading the template. By default, it's <code>django.template.loader</code> . See Chapter 9, Advanced

<code>_loader</code>	<i>Models</i> , for information on template loaders.
<code>template_name</code>	The full name of a template to use in rendering the page. This lets you override the default template name derived from the QuerySet .
<code>template_object_name</code>	The name of the template variable to use in the template context. By default, this is ' <code>object</code> '. Views that list more than one object (that is, <code>object_list</code> views and various objects-for-date views) will append ' <code>_list</code> ' to the value of this parameter.

Table C.1: Common Generic View Arguments

Simple generic views

The module `django.views.generic.base` contains simple views that handle a couple of common cases: rendering a template when no view logic is needed and issuing a redirect.

Rendering a template- TemplateView

This view renders a given template, passing it a context with keyword arguments captured in the URL.

Example:

Given the following URLconf:

```
from django.conf.urls import url

from myapp.views import HomePageView

urlpatterns = [
    url(r'^$', HomePageView.as_view(),
        name='home'),
]
```

And a sample `views.py`:

```
from django.views.generic.base import
TemplateView
from articles.models import Article
```

```
class HomePageView(TemplateView):

    template_name = "home.html"

    def get_context_data(self, **kwargs):
        context = super(HomePageView,
self).get_context_data(**kwargs)
        context['latest_articles'] =
Article.objects.all()[:5]
        return context
```

a request to `/` would render the template `home.html`, returning a context containing a list of the top 5 articles.

Redirecting to another URL

`django.views.generic.base.RedirectView()` redirects to a given URL.

The given URL may contain dictionary-style string formatting, which will be interpolated against the parameters captured in the URL. Because keyword interpolation is *always* done (even if no arguments are passed in), any `"%"` characters in the URL must be written as `"%%"` so that Python will convert them to a single percent sign on output.

If the given URL is `None`, Django will return an `HttpResponseGone` (410).

Example `views.py`:

```
from django.shortcuts import
get_object_or_404
```

```
from django.views.generic.base import
RedirectView

from articles.models import Article

class
ArticleCounterRedirectView(RedirectView):

    permanent = False
    query_string = True
    pattern_name = 'article-detail'

    def get_redirect_url(self, args,
*kwags):
        article =
get_object_or_404(Article,
pk=kwags['pk'])
        article.update_counter()
        return
super(ArticleCounterRedirectView,
self).get_redirect_url(args, *kwags)
```

Example urls.py:

```
from django.conf.urls import url
from django.views.generic.base import
RedirectView

from article.views import
ArticleCounterRedirectView, ArticleDetail

urlpatterns = [
    url(r'^counter/(?P<pk>[0-9]+)/$',
ArticleCounterRedirectView.as_view(),
        name='article-counter'),
    url(r'^details/(?P<pk>[0-9]+)/$',
        ArticleDetail.as_view(),
```

```
        name='article-detail'),
        url(r'^go-to-django/$',
            RedirectView.as_view(url='http://djangoproject.com'),
            name='go-to-django'),
    ]
```

ATTRIBUTES

url

The URL to redirect to, as a string. Or `None` to raise a 410 (Gone) HTTP error.

pattern_name

The name of the URL pattern to redirect to. Reversing will be done using the same `*args` and `**kwargs` as are passed in for this view.

permanent

Whether the redirect should be permanent. The only difference here is the HTTP status code returned. If `True`, then the redirect will use status code 301. If `False`, then the redirect will use status code 302. By default, `permanent` is `True`.

query_string

Whether to pass along the GET query string to the new location. If `True`, then the query string is appended to the URL. If `False`, then the query string is discarded. By default, `query_string` is `False`.

METHODS

`get_redirect_url(args, *kwargs)` constructs the target URL for redirection.

The default implementation uses `url` as a starting string and performs expansion of % named parameters in that string using the named groups captured in the URL.

If `url` is not set, `get_redirect_url()` tries to reverse the `pattern_name` using what was captured in the URL (both named and unnamed groups are used).

If requested by `query_string`, it will also append the query string to the generated URL. Subclasses may implement any behavior they wish, as long as the method returns a redirect-ready URL string.

List/detail generic views

The list/detail generic views handle the common case of displaying a list of items at one view and individual detail views of those items at another.

Lists of objects

```
django.views.generic.list.ListView
```

Use this view to display a page representing a list of objects.

Example `views.py`:

```
from django.views.generic.list import
ListView
from django.utils import timezone

from articles.models import Article

class ArticleListView(ListView):

    model = Article

    def get_context_data(self, **kwargs):
        context = super(ArticleListView,
self).get_context_data(**kwargs)
        context['now'] = timezone.now()
        return context
```

Example `myapp/urls.py`:

```
from django.conf.urls import url

from article.views import ArticleListView

urlpatterns = [
    url(r'^$', ArticleListView.as_view(),
name='article-list'),
]
```

Example myapp/article_list.html:

```
<h1>Articles</h1>
<ul>
{% for article in object_list %}
    <li>{{ article.pub_date|date }}-{{ article.headline }}</li>
{% empty %}
    <li>No articles yet.</li>
{% endfor %}
</ul>
```

Detail views

django.views.generic.detail.DetailView

This view provides a detail view of a single object.

Example myapp/views.py:

```
from django.views.generic.detail import
DetailView
from django.utils import timezone

from articles.models import Article

class ArticleDetailView(DetailView):
    ...
```

```
model = Article

def get_context_data(self, **kwargs):
    context = super(ArticleDetailView,
                    self).get_context_data(**kwargs)
    context['now'] = timezone.now()
    return context
```

Example myapp/urls.py:

```
from django.conf.urls import url

from article.views import
ArticleDetailView

urlpatterns = [
    url(r'^(?P<slug>[-\w]+)/$',,
        ArticleDetailView.as_view(),
        name='article-detail'),
]
```

Example myapp/article_detail.html:

```
<h1>{{ object.headline }}</h1>
<p>{{ object.content }}</p>
<p>Reporter: {{ object.reporter }}</p>
<p>Published: {{ object.pub_date|date }}</p>
<p>Date: {{ now|date }}</p>
```

Date-Based Generic Views

Date-based generic views, provided in `django.views.generic.dates`, are views for displaying drilldown pages for date-based data.

ArchiveIndexView

A top-level index page showing the latest objects, by date. Objects with a date in the *future* are not included unless you set `allow_future` to `True`.

Context

In addition to the context provided by `django.views.generic.list.MultipleObjectMixin` (via `django.views.generic.dates.BaseDateListView`), the template's context will be:

- `date_list`: A `DateQuerySet` object containing all years that have objects available according to `queryset`, represented as `datetime.datetime` objects, in descending order

Notes

- Uses a default `context_object_name` of `latest`.
- Uses a default `template_name_suffix` of `_archive`.
- Defaults to providing `date_list` by year, but this can be altered to month or day using the attribute `date_list_period`. This also applies to all subclass views:

```
Example myapp/urls.py:  
from django.conf.urls import url  
from django.views.generic.dates import  
ArchiveIndexView  
  
from myapp.models import Article  
  
urlpatterns = [  
    url(r'^archive/$',  
        ArchiveIndexView.as_view(model=Article,  
        date_field="pub_date"),  
        name="article_archive"),  
]
```

Example myapp/article_archive.html:

```
<ul>  
    {% for article in latest %}  
        <li>{{ article.pub_date }}: {{  
            article.title }}</li>  
    {% endfor %}  
</ul>
```

This will output all articles.

YearArchiveView

A yearly archive page showing all available months in a given year. Objects with a date in the *future* are not displayed unless you set `allow_future` to `True`.

Context

In addition to the context provided by `djongo.views.generic.list.MultipleObjectMi`

xin (via `django.views.generic.dates.BaseDateListView`), the template's context will be:

- `date_list`: A `DateQuerySet` object containing all months that have objects available according to `queryset`, represented as `datetime.datetime` objects, in ascending order
- `year`: A `date` object representing the given year
- `next_year`: A `date` object representing the first day of the next year, according to `allow_empty` and `allow_future`
- `previous_year`: A `date` object representing the first day of the previous year, according to `allow_empty` and `allow_future`

Notes

- Uses a default `template_name_suffix` of `archiveyear`

Example myapp/views.py:

```
from django.views.generic.dates import
YearArchiveView

from myapp.models import Article

class
ArticleYearArchiveView(YearArchiveView):
    queryset = Article.objects.all()
    date_field = "pub_date"
    make_object_list = True
    allow_future = True
```

Example myapp/urls.py:

```
from django.conf.urls import url

from myapp.views import
ArticleYearArchiveView
```

```
ARTICLEYEARARCHIVEVIEW

urlpatterns = [
    url(r'^(?P<year>[0-9]{4})/$',
        ArticleYearArchiveView.as_view(),
        name="article_year_archive"),
]
```

Example myapp/articlearchiveyear.html:

```
<ul>
    {% for date in date_list %}
        <li>{{ date|date }}</li>
    {% endfor %}
</ul>
<div>
    <h1>All Articles for {{ year|date:"Y" }}</h1>
    {% for obj in object_list %}
        <p>
            {{ obj.title }}-{{ obj.pub_date|date:"F j, Y" }}
        </p>
    {% endfor %}
</div>
```

MonthArchiveView

A monthly archive page showing all objects in a given month. Objects with a date in the *future* are not displayed unless you set `allow_future` to True.

Context

In addition to the context provided by `MultipleObjectMixin` (via `BaseDateListView`),

the template's context will be:

- `date_list`: A `DateQuerySet` object containing all days that have objects available in the given month, according to `queryset`, represented as `datetime.datetime` objects, in ascending order
- `month`: A `date` object representing the given month
- `next_month`: A `date` object representing the first day of the next month, according to `allow_empty` and `allow_future`
- `previous_month`: A `date` object representing the first day of the previous month, according to `allow_empty` and `allow_future`

Notes

- Uses a default `template_name_suffix` of `archivemonth`

Example myapp/views.py:

```
from django.views.generic.dates import
MonthArchiveView

from myapp.models import Article

class
ArticleMonthArchiveView(MonthArchiveView):
    queryset = Article.objects.all()
    date_field = "pub_date"
    make_object_list = True
    allow_future = True
```

Example myapp/urls.py:

```
from django.conf.urls import url

from myapp.views import
ArticleMonthArchiveView

urlpatterns = [
```

```

# Example: 2012aug/

url(r'^(?P<year>[0-9]{4})/(?P<month>[-\w]+
)/$', 
    ArticleMonthArchiveView.as_view(),
    name="archive_month"),
# Example: 201208/

url(r'^(?P<year>[0-9]{4})/(?P<month>[0-9]+
)/$', 
    ArticleMonthArchiveView.as_view(month_form
at='%m'),
    name="archive_month_numeric"),
]

```

Example myapp/articlearchivemonth.html:

```

<ul>
    {% for article in object_list %}
        <li>{{ article.pub_date|date:"F j,
Y" }}:
            {{ article.title }}
        </li>
    {% endfor %}
</ul>

<p>
    {% if previous_month %}
        Previous Month: {{ previous_month|date:"F Y" }}
    {% endif %}
    {% if next_month %}
        Next Month: {{ next_month|date:"F
Y" }}
    {% endif %}
</p>

```

WeekArchiveView

A weekly archive page showing all objects in a given week. Objects with a date in the *future* are not displayed unless you set `allow_future` to `True`.

Context

In addition to the context provided by `MultipleObjectMixin` (via `BaseDateListView`), the template's context will be:

- `week`: A `date` object representing the first day of the given week
- `next_week`: A `date` object representing the first day of the next week, according to `allow_empty` and `allow_future`
- `previous_week`: A `date` object representing the first day of the previous week, according to `allow_empty` and `allow_future`

Notes

- Uses a default `template_name_suffix` of `archiveweek`

Example myapp/views.py:

```
from django.views.generic.dates import  
WeekArchiveView  
  
from myapp.models import Article  
  
class  
ArticleWeekArchiveView(WeekArchiveView):  
    queryset = Article.objects.all()  
    date_field = "pub_date"  
    make_object_list = True  
    week_format = "%W"  
    allow_future = True
```

Example myapp/urls.py:

```
from django.conf.urls import url

from myapp.views import
ArticleWeekArchiveView

urlpatterns = [
    # Example: 2012week/23/

url(r'^(?P<year>[0-9]{4})/week/(?P<week>[0
-9]+)$',
    ArticleWeekArchiveView.as_view(),
    name="archive_week"),
]
```

Example myapp/articlearchiveweek.html:

```
<h1>Week {{ week|date:'W' }}</h1>

<ul>
    {% for article in object_list %}
        <li>{{ article.pub_date|date:"F j,
Y" }}: {{ article.title }}</li>
    {% endfor %}
</ul>

<p>
    {% if previous_week %}
        Previous Week: {{ previous_week|date:"F Y" }}
    {% endif %}
    {% if previous_week and next_week %}--
    {% endif %}
    {% if next_week %}
        Next week: {{ next_week|date:"F Y"
}}
    {% endif %}
</p>
```

In this example, you are outputting the week number. The default `week_format` in the `WeekArchiveView` uses week format "%U" which is based on the United States week system where the week begins on a Sunday. The "%W" format uses the ISO week format and its week begins on a Monday. The "%W" format is the same in both the `strftime()` and the `date`.

However, the `date` template filter does not have an equivalent output format that supports the US based week system. The `date` filter "%U" outputs the number of seconds since the Unix epoch.

DayArchiveView

A day archive page showing all objects in a given day. Days in the future throw a 404 error, regardless of whether any objects exist for future days, unless you set `allow_future` to `True`.

Context

In addition to the context provided by `MultipleObjectMixin` (via `BaseDateListView`), the template's context will be:

- `day`: A `date` object representing the given day
- `next_day`: A `date` object representing the next day, according to `allow_empty` and `allow_future`
- `previous_day`: A `date` object representing the previous day,

according to `allow_empty` and `allow_future`

- `next_month`: A `date` object representing the first day of the next month, according to `allow_empty` and `allow_future`
- `previous_month`: A `date` object representing the first day of the previous month, according to `allow_empty` and `allow_future`

Notes

- Uses a default `template_name_suffix` of `archiveday`

Example myapp/views.py:

```
from django.views.generic.dates import
DayArchiveView

from myapp.models import Article

class
ArticleDayArchiveView(DayArchiveView):
    queryset = Article.objects.all()
    date_field = "pub_date"
    make_object_list = True
    allow_future = True
```

Example myapp/urls.py:

```
from django.conf.urls import url

from myapp.views import
ArticleDayArchiveView

urlpatterns = [
    # Example: 2012nov/10/

    url(r'^(?P<year>[0-9]{4})/ (?P<month>[-\w]+
)/ (?P<day>[0-9]+)$',
        ArticleDayArchiveView.as_view(),
        name="archive_day"),
```

]

Example myapp/articlearchiveday.html:

```
<h1>{{ day }}</h1>

<ul>
    {% for article in object_list %}
        <li>
            {{ article.pub_date|date:"F j, Y" }}
        : {{ article.title }}
        </li>
    {% endfor %}
</ul>

<p>
    {% if previous_day %}
        Previous Day: {{ previous_day }}
    {% endif %}
    {% if previous_day and next_day %}--{%
    endif %}
    {% if next_day %}
        Next Day: {{ next_day }}
    {% endif %}
</p>
```

TodayArchiveView

A day archive page showing all objects for *today*. This is exactly the same as [django.views.generic.dates.DayArchiveView](#), except today's date is used instead of the `year/month/day` arguments.

Notes

- Uses a default `template_name_suffix` of `archivetoday`

Example myapp/views.py:

```
from django.views.generic.dates import
TodayArchiveView

from myapp.models import Article

class
ArticleTodayArchiveView(TodayArchiveView):
    queryset = Article.objects.all()
    date_field = "pub_date"
    make_object_list = True
    allow_future = True
```

Example myapp/urls.py:

```
from django.conf.urls import url

from myapp.views import
ArticleTodayArchiveView

urlpatterns = [
    url(r'^today/$',
        ArticleTodayArchiveView.as_view(),
        name="archive_today"),
]
```

Where is the example template for
`TodayArchiveView`?

This view uses by default the same template as the `DayArchiveView`, which is in the previous example. If you need a different template, set the `template_name` attribute to be the name of the new template.

DateDetailView

A page representing an individual object. If the object has a date value in the future, the view will throw a 404 error by default, unless you set `allow_future` to `True`.

Context

- Includes the single object associated with the `model` specified in the `DateDetailView`

Notes

- Uses a default `template_name_suffix` of `_detail`

```
Example myapp/urls.py:  
from django.conf.urls import url  
from django.views.generic.dates import  
DateDetailView  
  
urlpatterns = [  
    url(r'^(?P<year>[0-9]+)/(?P<month>[-\w]+)/(?P<day>[0-9]+)/  
        (?P<pk>[0-9]+)/$',  
  
        DateDetailView.as_view(model=Article,  
                               date_field="pub_date"),  
                               name="archive_date_detail"),  
]
```

Example myapp/article_detail.html:

```
<h1>{{ object.title }}</h1>
```

Form handling with class-based views

Form processing generally has 3 paths:

- Initial **GET** (blank or prepopulated form)
- **POST** with invalid data (typically redisplay form with errors)
- **POST** with valid data (process the data and typically redirect)

Implementing this yourself often results in a lot of repeated boilerplate code (see Using a form in a view). To help avoid this, Django provides a collection of generic class-based views for form processing.

Basic forms

Given a simple contact form:

```
# forms.py

from django import forms

class ContactForm(forms.Form):
    name = forms.CharField()
    message =
    forms.CharField(widget=forms.Textarea)

    def send_email(self):
        # send email using the
        self.cleaned_data dictionary
        pass
```

The view can be constructed using a [FormView](#):

```
# views.py

from myapp.forms import ContactForm
from django.views.generic.edit import
FormView

class ContactView(FormView):
    template_name = 'contact.html'
    form_class = ContactForm
    success_url = 'thanks'

    def form_valid(self, form):
        # This method is called when valid
form data has been POSTed.
        # It should return an HttpResponseRedirect.
        form.send_email()
        return super(ContactView,
self).form_valid(form)
```

Notes:

- `FormView` inherits `TemplateResponseMixin` so `template_name` can be used here
- The default implementation for `form_valid()` simply redirects to the `success_url`

Model forms

Generic views really shine when working with models. These generic views will automatically create a `ModelForm`, so long as they can work out which model class to use:

- If the `model` attribute is given, that model class will be used

- If `get_object()` returns an object, the class of that object will be used
- If a `queryset` is given, the model for that queryset will be used

Model form views provide a `form_valid()` implementation that saves the model automatically. You can override this if you have any special requirements; see below for examples.

You don't even need to provide a `success_url` for `CreateView` or `UpdateView`-they will use `get_absolute_url()` on the model object if available.

If you want to use a custom `ModelForm` (for instance to add extra validation) simply set `form_class` on your view.

NOTE

When specifying a custom form class, you must still specify the model, even though the `form_class` may be a `ModelForm`.

First we need to add `get_absolute_url()` to our `Author` class:

```
# models.py

from django.core.urlresolvers import
reverse
from django.db import models

class Author(models.Model):
    name =
models.CharField(max_length=200)

def get_absolute_url(self):
```

```
        return reverse('author-detail',
    kwargs={'pk': self.pk})
```

Then we can use [CreateView](#) and friends to do the actual work. Notice how we're just configuring the generic class-based views here; we don't have to write any logic ourselves:

```
# views.py

from django.views.generic.edit import
CreateView, UpdateView, DeleteView
from django.core.urlresolvers import
reverse_lazy
from myapp.models import Author

class AuthorCreate(CreateView):
    model = Author
    fields = ['name']

class AuthorUpdate(UpdateView):
    model = Author
    fields = ['name']

class AuthorDelete(DeleteView):
    model = Author
    success_url = reverse_lazy('author-
list')
```

We have to use `reverse_lazy()` here, not just `reverse` as the urls are not loaded when the file is imported.

The `fields` attribute works the same way as the `fields` attribute on the inner [Meta](#) class on [ModelForm](#). Unless you define the form class in another

way, the attribute is required and the view will raise an `ImproperlyConfigured` exception if it's not.

If you specify both the `fields` and `form_class` attributes, an `ImproperlyConfigured` exception will be raised.

Finally, we hook these new views into the URLconf:

```
# urls.py

from django.conf.urls import url
from myapp.views import AuthorCreate,
AuthorUpdate, AuthorDelete

urlpatterns = [
    #
    url(r'author/add/$',
AuthorCreate.as_view(),
name='author_add'),
    url(r'author/(?P<pk>[0-9]+)/$',
AuthorUpdate.as_view(),
        name='author_update'),
    url(r'author/(?P<pk>[0-9]+)/delete/$',
AuthorDelete.as_view(),
        name='author_delete'),
]
```

In this example:

- `CreateView` and `UpdateView` use `myapp/author_form.html`
- `DeleteView` uses `myapp/author_confirm_delete.html`

If you wish to have separate templates for `CreateView` and `UpdateView`, you can set either `template_name` or `template_name_suffix` on your view class.

Models and `request.user`

To track the user that created an object using a `CreateView`, you can use a custom `ModelForm` to do this. First, add the foreign key relation to the model:

```
# models.py

from django.contrib.auth.models import
User
from django.db import models

class Author(models.Model):
    name =
models.CharField(max_length=200)
    created_by = models.ForeignKey(User)

# ...
```

In the view, ensure that you don't include `created_by` in the list of fields to edit, and override `form_valid()` to add the user:

```
# views.py

from django.views.generic.edit import
CreateView
from myapp.models import Author

class AuthorCreate(CreateView):
    model = Author
    fields = ['name']

    def form_valid(self, form):
        form.instance.created_by =
self.request.user
        return super(AuthorCreate,
```

```
self).form_valid(form)
```

Note that you'll need to decorate this view using `login_required()`, or alternatively handle unauthorized users in the `form_valid()`.

AJAX example

Here is a simple example showing how you might go about implementing a form that works for AJAX requests as well as *normal* form `POST`:

```
from django.http import JsonResponse
from django.views.generic.edit import CreateView
from myapp.models import Author

class AjaxableResponseMixin(object):
    def form_invalid(self, form):
        response =
super(AjaxableResponseMixin,
self).form_invalid(form)
        if self.request.is_ajax():
            return
JsonResponse(form.errors, status=400)
        else:
            return response

    def form_valid(self, form):
        # We make sure to call the
parent's form_valid() method because
        # it might do some processing (in
the case of CreateView, it will
        # call form.save() for example).
        response =
super(AjaxableResponseMixin,
self).form_valid(form)
        ...
```

```
    if self.request.is_ajax():
        data = {
            'pk': self.object.pk,
        }
        return JsonResponse(data)
    else:
        return response

class AuthorCreate(AjaxableResponseMixin,
CreateView):
    model = Author
    fields = ['name']
```

Appendix D. Settings

Your Django settings file contains all the configuration of your Django installation. This appendix explains how settings work and which settings are available.

What's a settings file?

A settings file is just a Python module with module-level variables. Here are a couple of example settings:

```
ALLOWED_HOSTS = ['www.example.com'] DEBUG  
= False DEFAULT_FROM_EMAIL =  
'webmaster@example.com'
```

NOTE

If you set `DEBUG` to `False`, you also need to properly set the `ALLOWED_HOSTS` setting.

Because a settings file is a Python module, the following apply:

- It doesn't allow for Python syntax errors
- It can assign settings dynamically using normal Python syntax, for example:

```
MY_SETTING = [str(i) for i in  
range(30)]
```

- It can import values from other settings files

Default settings

A Django settings file doesn't have to define any settings if it doesn't need to. Each setting has a sensible default value. These defaults live in the module [django/conf/global_settings.py](#). Here's the algorithm Django uses in compiling settings:

- Load settings from [global_settings.py](#)
- Load settings from the specified settings file, overriding the global settings as necessary

Note that a settings file should *not* import from [global_settings](#), because that's redundant.

Seeing which settings you've changed

There's an easy way to view which of your settings deviate from the default settings. The command [python manage.py diffsettings](#) displays differences between the current settings file and Django's default settings. For more, see the [diffsettings](#) documentation.

Using settings in Python code

In your Django apps, use settings by importing the object `django.conf.settings`. Example:

```
from django.conf import settings
if settings.DEBUG:
    # Do something
```

Note that `django.conf.settings` isn't a module—it's an object. So importing individual settings is not possible:

```
from django.conf.settings import DEBUG  #
This won't work.
```

Also note that your code should *not* import from either `global_settings` or your own settings file.

`django.conf.settings` abstracts the concepts of default settings and site-specific settings; it presents a single interface. It also decouples the code that uses settings from the location of your settings.

Altering settings at runtime

You shouldn't alter settings in your applications at runtime. For example, don't do this in a view:

```
from django.conf import settings  
settings.DEBUG = True    # Don't do this!
```

The only place you should assign to settings is in a settings file.

Security

Because a settings file contains sensitive information, such as the database password, you should make every attempt to limit access to it. For example, change its file permissions so that only you and your web server's user can read it. This is especially important in a shared-hosting environment.

Creating your own settings

There's nothing stopping you from creating your own settings, for your own Django apps. Just follow these conventions:

- Setting names are in all uppercase
- Don't reinvent an already-existing setting

For settings that are sequences, Django itself uses tuples, rather than lists, but this is only a convention.

DJANGO_SETTINGS_MODULE

E

When you use Django, you have to tell it which settings you're using. Do this by using an environment variable, `DJANGO_SETTINGS_MODULE`. The value of `DJANGO_SETTINGS_MODULE` should be in Python path syntax, for example, `mysite.settings`.

The django-admin utility

When using `django-admin`, you can either set the environment variable once, or explicitly pass in the settings module each time you run the utility. Example (Unix Bash shell):

```
export  
DJANGO_SETTINGS_MODULE=mysite.settings  
django-admin runserver
```

Example (Windows shell):

```
set DJANGO_SETTINGS_MODULE=mysite.settings  
django-admin runserver
```

Use the `--settings` command-line argument to specify the settings manually:

```
django-admin runserver --  
settings=mysite.settings
```

On the server (mod_wsgi)

In your live server environment, you'll need to tell your WSGI application what settings file to use. Do that with `os.environ`:

```
import os
os.environ['DJANGO_SETTINGS_MODULE'] =
'mysite.settings'
```

Read [Chapter 13, *Deploying Django*](#), for more information and other common elements to a Django WSGI application.

Using settings without setting DJANGO_SETTINGS_MODULE

In some cases, you might want to bypass the `DJANGO_SETTINGS_MODULE` environment variable. For example, if you're using the template system by itself, you likely don't want to have to set up an environment variable pointing to a settings module. In these cases, you can configure Django's settings manually. Do this by calling:

```
django.conf.settings.configure(default_settings, **settings)
```

Example:

```
from django.conf import settings
settings.configure(DEBUG=True,
                   TEMPLATE_DEBUG=True)
```

Pass `configure()` as many keyword arguments as you'd like, with each keyword argument representing a setting and its value. Each argument name should be all uppercase, with the same name as the settings described above. If a particular setting is not passed to `configure()` and is needed at some later point,

Django will use the default setting value.

Configuring Django in this fashion is mostly necessary-and, indeed, recommended-when you're using a piece of the framework inside a larger application. Consequently, when configured via `settings.configure()`, Django will not make any modifications to the process environment variables (see the documentation of `TIME_ZONE` for why this would normally occur). It's assumed that you're already in full control of your environment in these cases.

Custom default settings

If you'd like default values to come from somewhere other than `django.conf.global_settings`, you can pass in a module or class that provides the default settings as the `default_settings` argument (or as the first positional argument) in the call to `configure()`. In this example, default settings are taken from `myapp_defaults`, and the `DEBUG` setting is set to `True`, regardless of its value in `myapp_defaults`:

```
from django.conf import settings
from myapp import myapp_defaults

settings.configure(default_settings=myapp_
defaults, DEBUG=True)
```

The following example, which uses `myapp_defaults` as a positional argument, is equivalent:

```
settings.configure(myapp_defaults,  
                  DEBUG=True)
```

Normally, you will not need to override the defaults in this fashion. The Django defaults are sufficiently tame that you can safely use them. Be aware that if you do pass in a new default module, it entirely *replaces* the Django defaults, so you must specify a value for every possible setting that might be used in that code you are importing. Check in

`django.conf.settings.global_settings` for the full list.

Either `configure()` or `DJANGO_SETTINGS_MODULE` is required

If you're not setting the `DJANGO_SETTINGS_MODULE` environment variable, you *must* call `configure()` at some point before using any code that reads settings. If you don't set `DJANGO_SETTINGS_MODULE` and don't call `configure()`, Django will raise an `ImportError` exception the first time a setting is accessed. If you set `DJANGO_SETTINGS_MODULE`, access settings values somehow, *then* call `configure()`, Django will raise a `RuntimeError` indicating that settings have already been configured. There is a property just for this purpose:

```
django.conf.settings.configured
```

For example:

```
from django.conf import settings
if not settings.configured:
    settings.configure(myapp_defaults,
DEBUG=True)
```

Also, it's an error to call `configure()` more than once, or to call `configure()` after any setting has been accessed. It boils down to this: Use exactly one of either `configure()` or `DJANGO_SETTINGS_MODULE`. Not both, and not neither.

Available settings

There are a large number of settings available in Django. For ease of reference, I have broken them up into six sections, each with a corresponding table in this Appendix:

- Core settings (*Table D.1*)
- Authentication settings (*Table D.2*)
- Message settings (*Table D.3*)
- Session settings (*Table D.4*)
- Django sites settings (*Table D.5*)
- Static files settings (*Table D.6*)

Each table lists the available setting and its default value. For additional information and use cases for each setting, see the Django Project website at

<https://docs.djangoproject.com/en/1.8/ref/settings/>.

NOTE

Be careful when you override settings, especially when the default value is a non-empty list or dictionary, such as `MIDDLEWARE_CLASSES` and `STATICFILES_FINDERS`. Make sure you keep the components required by the features of Django you wish to use.

Core settings

Setting	Default Value
<code>ABSOLUTE</code>	

<code>_URL_OVE RRIDES</code>	<code>{}</code> (Empty dictionary)
<code>ADMINS</code>	<code>[]</code> (Empty list)
<code>ALLOWED_ HOSTS</code>	<code>[]</code> (Empty list)
<code>APPEND_S LASH</code>	<code>True</code>
<code>CACHE_MI DDLEWARE _ALIAS</code>	<code>default</code>
<code>CACHES</code>	<code>{ 'default': { 'BACKEND': 'django.core.cache.backends.locmem.LocMemCache' , } }</code>
<code>CACHE_MI DDLEWARE _KEY_PRE FIX</code>	<code>''</code> (empty string)
<code>CACHE_MI DDLEWARE _SECONDS</code>	<code>600</code>

CSRF_COOKIE_AGE	31449600 (1 year, in seconds)
CSRF_COOKIE_DOMAIN_IN	None
CSRF_COOKIE_HTTP_ONLY	False
CSRF_COOKIE_NAME	Csrftoken
CSRF_COOKIE_PATH	'/'
CSRF_COOKIE_SECURE	False
DATE_INPUT_FORMATS	['%Y-%m-%d', '%m/%d/%Y', '%m/%d/%y', '%b %d %Y', '%b %d, %Y', '%d %b %Y', '%d %b, %Y', '%B %d %Y', '%B %d, %Y', '%d %B %Y', '%d %B, %Y',]
DATETIME_INPUT_FORMATS	'N j, Y, P' (for example, Feb. 4, 2003, 4

<u>FORMAT</u>	p.m.)
<u>DATETIME_FORMATS</u> <u>INPUT_F</u>	['%Y-%m-%d %H:%M:%S', '%Y-%m-%d %H:%M:%S.%f', '%Y-%m-%d %H:%M', '%Y-%m-%d', '%m/%d/%Y %H:%M:%S', '%m/%d/%Y %H:%M:%S.%f', '%m/%d/%Y %H:%M:%S', '%m/%d/%Y %H:%M:%S.%f', '%m/%d/%y %H:%M:%S', '%m/%d/%y %H:%M:%S.%f', '%m/%d/%y %H:%M', '%m/%d/%y',]
<u>DEBUG</u>	False
<u>DEBUG_PROPAGATE_EXCEPTIONS</u>	False
<u>DECIMAL_SEPARATOR</u>	'.' (Dot)
<u>DEFAULT_CHARSET</u>	'utf-8'
<u>DEFAULT_CONTENT_TYPE</u>	'text/html'

<code>DEFAULT_EXCEPTION_REPORT_FILTER</code>	<code>django.views.debug.SafeExceptionReporterFilter</code>
<code>DEFAULT_FILE_STORAGE</code>	<code>django.core.files.storage.FileSystemStorage</code>
<code>DEFAULT_FROM_EMAIL</code>	<code>'webmaster@localhost'.</code>
<code>DEFAULT_INDEX_TABLESPACE</code>	<code>'' (Empty string)</code>
<code>DEFAULT_TABLESPACE</code>	<code>'' (Empty string)</code>
<code>DISALLOWED_USER_AGENTS</code>	<code>[] (Empty list)</code>
<code>EMAIL_BACKEND</code>	<code>django.core.mail.backends.smtp.EmailBackend</code>

EMAIL_HOST	'localhost'
EMAIL_HOST_PASSWORD	'' (Empty string)
EMAIL_HOST_USER	'' (Empty string)
EMAIL_PORT	25
EMAIL_SUBJECT_PREFIX	'[Django] '
EMAIL_USE_TLS	False
EMAIL_USE_SSL	False
EMAIL_SSL_CERTFILE	None

EMAIL_SS L_KEYFILE	None
EMAIL_TIMOUT	None
FILE_CHARSET	'utf-8'
FILE_UPLOAD_HANDLERS	['django.core.files.uploadhandler.MemoryFileUploadHandler', 'django.core.files.uploadhandler.TemporaryFileUploadHandler']
FILE_UPLOAD_MAX_MEMORY_SIZE	2621440 (that is, 2.5 MB)
FILE_UPLOAD_DIRECTORY_PERMISSIONS	None
FILE_UPLOAD_D	

FILE_UPLOADED_PERMISSIONS	None
FILE_UPLOAD_TEMP_DIR	None
FIRST_DAY_OF_WEEK	0 (Sunday)
FIXTURE_DIRS	[] (Empty list)
FORCE_SCRIPT_NAME	None
FORMAT_MODULE_PATH	None
IGNOREABLE_404_URLS	[] (Empty list)
INSTALLED_APPS	[] (Empty list)

<code>INTERNAL_IPS</code>	<code>[]</code> (Empty list)
<code>LANGUAGE_CODE</code>	<code>'en-us'</code>
<code>LANGUAGE_COOKIE_AGE</code>	<code>None</code> (expires at browser close)
<code>LANGUAGE_COOKIE_DOMAIN</code>	<code>None</code>
<code>LANGUAGE_COOKIE_NAME</code>	<code>'django_language'</code>
<code>Languages</code>	A list of all available languages
<code>LOCALE_PATHS</code>	<code>[]</code> (Empty list)
<code>LOGGING</code>	A loading configuration dictionary

Setting		Description
LOGGING_CONFIG	'logging.config.dictConfig'	
MANAGERS	[] (Empty list)	
MEDIA_ROOT	'' (Empty string)	
MEDIA_URL	'' (Empty string)	
MIDDLEWARE_CLASSES	['django.middleware.common.CommonMiddleware', 'django.middleware.csrf.CsrfViewMiddleware']	
MIGRATION_MODULES	{ } (empty dictionary)	
MONTH_DAY_FORMAT	'F j'	
NUMBER_GROUPING	0	

PREPEND_WWW	False
ROOT_URL_CONF	Not defined
SECRET_KEY	'' (Empty string)
SECURE_BROWSER_XSS_FILTER	False
SECURE_CONTENT_TYPE_NOSNIFF	False
SECURE_HSTS_INCLUDING_SUBDOMAINS	False
SECURE_HSTS_SECONDS	0

SECURE_P ROXY_SSL _HEADER	None
SECURE_R EDIRECT_ EXEMPT	[] (Empty list)
SECURE_S SL_HOST	None
SECURE_S SL_REDIR ECT	False
SERIALIZ ATION_MO DULES	Not defined
SERVER_E MAIL	'root@localhost'
SHORT_DA TE_FORMA T	m/d/Y (for example, 12/31/2003)

<code>SHORT_DATE_FORMAT</code>	<code>m/d/Y P</code> (for example, 12/31/2003 4 p.m.)
<code>SIGNING_BACKEND</code>	<code>'django.core.signing.TimestampSigner'</code>
<code>SILENCED_SYSTEM_CHECKS</code>	<code>[]</code> (Empty list)
<code>TEMPLATES</code>	<code>[]</code> (Empty list)
<code>TEMPLATE_DEBUG</code>	<code>False</code>
<code>TEST_RUNNER</code>	<code>'django.test.runner.DiscoverRunner'</code>
<code>TEST_NON_SERIALIZED_APPS</code>	<code>[]</code> (Empty list)
<code>THOUSAND_SEPARATOR</code>	<code>,</code> (Comma)

TIME_FORMAT	'P' (for example, 4 p.m.)
TIME_INPUT_FORMAT_TS	['%H:%M:%S', '%H:%M:%S.%f', '%H:%M',]
TIME_ZONE	'America/Chicago'
USE_ETAGS	False
USE_I18N	True
USE_L10N	False
USE_THOUSAND_SEPARATOR	False
USE_TZ	False
use_v_en	

<code>USE_X_FORWARDED_HOST</code>	False
<code>WSGI_APPLICATION</code>	None
<code>YEAR_MONTH_FORMAT</code>	'F Y'
<code>X_FRAME_OPTIONS</code>	'SAMEORIGIN'

Table D.1: Django core settings

Auth

Setting	Default Value
<code>AUTHENTICATION_BACKENDS</code>	'django.contrib.auth.backends.ModelBackend'

AUT H_U SER _MO DEL	'auth.User'
LOG IN_ RED IRE CT_ URL	'accountsprofile/'
LOG IN_ URL	'accountslogin/'
LOG OUT _UR L	'accountslogout/'
PAS SWO RD_ RES ET_ 3 TIM EOU T_D AYS	

	['django.contrib.auth.hashers.PBKDF2PasswordHasher', 'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher', 'django.contrib.auth.hashers.BCryptPasswordHasher', 'django.contrib.auth.hashers.SHA1PasswordHasher', 'django.contrib.auth.hashers.MD5PasswordHasher', 'django.contrib.auth.hashers.UnsaltedMD5PasswordHasher', 'django.contrib.auth.hashers.CryptPasswordHasher']
--	--

Table D.2: Django authentication settings

Messages

Setting	Default Value
MESSAGE_LEVEL	messages
MESSAGE_STORAGE	'django.contrib.messages.storage.fallback.FallbackStorage'
MESSAGE_TAGS	{ messages.DEBUG: 'debug', messages.INFO: 'info', messages.SUCCESS: 'success', messages.WARNING: }

GS	'warning', messages.ERROR: 'error' }
----	--------------------------------------

Table D.3: Django messages settings

Sessions

Setting	Default Value
SESSION_CACHE_ALIAS	default
SESSION_COOKIE_AGE	1209600 (2 weeks, in seconds).
SESSION_COOKIE_DOMAIN	None
SESSION_COOKIE_HTTPONLY	True.
SESSION_COOKIE_NAME	'sessionid'
SESSION_COOKIE_PATH	'/'
SESSION_COOKIE_SECURE	False

SESSION_ENGINE	'django.contrib.sessions.backends.db'
SESSION_EXPIRE_AT_BROWSER_CLOSE	False
SESSION_FILE_PATH	None
SESSION_SAVE_EVERY_REQUEST	False
SESSION_SERIALIZER	'django.contrib.sessions.serializers.JSONSerializer'

Table D.4: Django sessions settings

Sites

Setting	Default Value
SITE_ID	Not defined

Table D.5: Django sites settings

Static files

Setting	Default Value
STATIC_ROOT	None
STATIC_URL	None
STATICFILES_DIRS	[] (Empty list)
STATICFILES_STORAGE	'django.contrib.staticfiles.storage.StaticFileStorage'
STATICFILE_FINDERS	["django.contrib.staticfiles.finders.FileSystemFinder", "django.contrib.staticfiles.finders.AppDirectoriesFinder"]

Table D.6: Django static files settings

Appendix E. Built-in Template Tags and Filters

Chapter 3, *Templates*, lists a number of the most useful builtin template tags and filters. However, Django ships with many more builtin tags and filters. This appendix provides a summary of all template tags and filters in Django. For more detailed information and use cases, see the Django Project website at <https://docs.djangoproject.com/en/1.8/ref/templates/builtins/>.

Builtin tags

autoescape

Controls the current autoescaping behavior. This tag takes either `on` or `off` as an argument and that determines whether autoescaping is in effect inside the block. The block is closed with an `endautoescape` ending tag.

When autoescaping is in effect, all variable content has HTML escaping applied to it before placing the result into the output (but after any filters have been applied). This is equivalent to manually applying the `escape` filter to each variable.

The only exceptions are variables that are already marked as safe from escaping, either by the code that populated the variable, or because it has had the `safe` or `escape` filters applied. Sample usage:

```
{% autoescape on %}  
    {{ body }}  
{% endautoescape %}
```

block

Defines a block that can be overridden by child templates. See "template inheritance" in [Chapter 3, Templates](#), for more information.

comment

Ignores everything between `{% comment %}` and `{% endcomment %}`. An optional note may be inserted in the first tag. For example, this is useful when commenting out code for documenting why the code was disabled.

`Comment` tags cannot be nested.

csrf_token

This tag is used for CSRF protection. For more information on **Cross Site Request Forgeries (CSRF)** see [Chapter 3, Templates](#), and [Chapter 19, Security in Django](#).

cycle

Produces one of its arguments each time this tag is encountered. The first argument is produced on the first encounter, the second argument on the second encounter, and so forth. Once all arguments are exhausted, the tag cycles to the first argument and produces it again. This tag is particularly useful in a loop:

```
{% for o in some_list %}
  <tr class="{% cycle 'row1' 'row2' %}">
    ...
  </tr>
{% endfor %}
```

The first iteration produces HTML that refers to class `row1`, the second to `row2`, the third to `row1` again, and so on for each iteration of the loop. You can use variables, too. For example, if you have two template variables, `rowvalue1` and `rowvalue2`, you can alternate between their values like this:

```
{% for o in some_list %}
  <tr class="{% cycle rowvalue1
rowvalue2 %}">
    ...
  </tr>
{% endfor %}
```

You can also mix variables and strings:

```
{% for o in some_list %}
  <tr class="{% cycle 'row1' rowvalue2
'row3' %}">
```

```
    ...
</tr>
{% endfor %}
```

You can use any number of values in a `cycle` tag, separated by spaces. Values enclosed in single quotes ('') or double quotes ("") are treated as string literals, while values without quotes are treated as template variables.

debug

Outputs a whole load of debugging information, including the current context and imported modules.

extends

Signals that this template extends a parent template.

This tag can be used in two ways:

- `{% extends "base.html" %}` (with quotes) uses the literal value "base.html" as the name of the parent template to extend.
- `{% extends variable %}` uses the value of `variable`. If the variable evaluates to a string, Django will use that string as the name of the parent template. If the variable evaluates to a `Template` object, Django will use that object as the parent template.

filter

Filters the contents of the block through one or more filters. See the `Builtin Filters` section later in this appendix for a list of filters in Django.

firstof

Outputs the first argument variable that is not `False`.

Outputs nothing if all the passed variables are `False`.

Sample usage:

```
{% firstof var1 var2 var3 %}
```

This is equivalent to:

```
{% if var1 %}
    {{ var1 }}
{% elif var2 %}
    {{ var2 }}
{% elif var3 %}
    {{ var3 }}
{% endif %}
```

for

Loops over each item in an array, making the item available in a context variable. For example, to display a list of athletes provided in `athlete_list`:

```
<ul>
  {% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
  {% endfor %}
</ul>
```

You can loop over a list in reverse by using `{% for obj in list reversed %}`. If you need to loop over a list of lists, you can unpack the values in each sub list

into individual variables. 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 %}
```

for... empty

The `for` tag can take an optional `{% empty %}` clause whose text is displayed if the given array is empty or could not be found:

```
<ul>  
    {% for athlete in athlete_list %}  
        <li>{{ athlete.name }}</li>  
    {% empty %}  
        <li>Sorry, no athletes in this list.  
    </li>  
    {% endfor %}  
</ul>
```

if

The `{% if %}` tag evaluates a variable, and if that variable is true (that is, exists, is not empty, and is not a false boolean value) the contents of the block are output:

```
{% if athlete_list %}  
    Number of athletes: {{  
        athlete_list|length }}  
    {% elif athlete_in_locker_room_list %}  
        Athlete in locker room: {{ athlete_in_locker_room }}
```

```
    Athletes should be out of the locker  
    room soon!  
    {% else %}  
        No athletes.  
    {% endif %}
```

In the above, if `athlete_list` is not empty, the number of athletes will be displayed by the `{{ athlete_list|length }}` variable. As you can see, the `if` tag may take one or several `{% elif %}` clauses, as well as an `{% else %}` clause that will be displayed if all previous conditions fail. These clauses are optional.

BOOLEAN OPERATORS

`if` tags may use `and`, `or`, or `not` to test a number of variables or to negate a given variable:

```
{% if athlete_list and coach_list %}  
    Both athletes and coaches are  
    available.  
{% endif %}  
  
{% if not athlete_list %}  
    There are no athletes.  
{% endif %}  
  
{% if athlete_list or coach_list %}  
    There are some athletes or some  
    coaches.  
{% endif %}
```

Use of both `and` and `or` clauses within the same tag is allowed, with `and` having higher precedence than `or` for example:

```
{% if athlete_list and coach_list or  
cheerleader_list %}
```

will be interpreted like:

```
if (athlete_list and coach_list) or  
cheerleader_list
```

Use of actual parentheses in the `if` tag is invalid syntax. If you need them to indicate precedence, you should use nested `if` tags.

`if` tags may also use the operators `==`, `!=`, `<`, `>`, `<=`, `>=`, and `in` which work as listed in *Table E.1*.

Operator	Example
<code>==</code>	<code>{% if somevar == "x" %} ...</code>
<code>!=</code>	<code>{% if somevar != "x" %} ...</code>
<code><</code>	<code>{% if somevar < 100 %} ...</code>
<code>></code>	<code>{% if somevar > 10 %} ...</code>

<=	{% if somevar <= 100 %} ...
>=	{% if somevar >= 10 %} ...
In	{% if "bc" in "abcdef" %}

Table E.1: Boolean operators in template tags

COMPLEX EXPRESSIONS

All of the above can be combined to form complex expressions. For such expressions, it can be important to know how the operators are grouped when the expression is evaluated—that is, the precedence rules. The precedence of the operators, from lowest to highest, is as follows:

- `or`
- `and`
- `not`
- `in`
- `==`, `!=`, `<`, `>`, `<=`, and `>=`

This order of precedence follows Python exactly.

FILTERS

You can also use filters in the `if` expression. For example:

```
{% if messages|length >= 100 %}  
    You have lots of messages today!  
{% endif %}
```

ifchanged

Check if a value has changed from the last iteration of a loop. The

`{% ifchanged %}` block tag is used within a loop. It has two possible uses:

- Checks its own rendered contents against its previous state and only displays the content if it has changed
- If given one or more variables, check whether any variable has changed

ifequal

Output the contents of the block if the two arguments equal each other. Example:

```
{% ifequal user.pk comment.user_id %}  
    ...  
{% endifequal %}
```

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

ifnotequal

Just like `ifequal`, except it tests that the two arguments are not equal. An alternative to the `ifnotequal` tag is

to use the `if` tag and the `!=` operator.

include

Loads a template and renders it with the current context. This is a way of including other templates within a template. The template name can either be a variable:

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

or a hard-coded (quoted) string:

```
{% include "foo/bar.html" %}
```

load

Loads a custom template tag set. For example, the following template would load all the tags and filters registered in `somelibrary` and `otherlibrary` located in package `package`:

```
{% load somelibrary package.otherlibrary %}
```

You can also selectively load individual filters or tags from a library, using the `from` argument.

In this example, the template tags/filters named `foo` and `bar` will be loaded from `somelibrary`:

```
{% load foo bar from somelibrary %}
```

See [Custom tag](#) and [Filter libraries](#) for more information.

lorem

Displays random lorem ipsum Latin text. This is useful for providing sample data in templates. Usage:

```
{% lorem [count] [method] [random] %}
```

The `{% lorem %}` tag can be used with zero, one, two or three arguments. The arguments are:

- **Count:** A number (or variable) containing the number of paragraphs or words to generate (default is 1).
- **Method:** Either w for words, p for HTML paragraphs or b for plain-text paragraph blocks (default is b).
- **Random:** The word random, which if given, does not use the common paragraph (Lorem ipsum dolor sit amet...) when generating text.

For example, `{% lorem 2 w random %}` will output two random Latin words.

now

Displays the current date and/or time, using a format according to the given string. Such string can contain format specifiers characters as described in the [date filter section](#). Example:

```
It is {% now "jS F Y H:i" %}
```

The format passed can also be one of the predefined

ones `DATE_FORMAT`, `DATETIME_FORMAT`,
`SHORT_DATE_FORMAT`, or `SHORT_DATETIME_FORMAT`.

The predefined formats may vary depending on the current locale and if format-localization is enabled, for example:

```
It is {% now "SHORT_DATETIME_FORMAT" %}
```

regroup

Regroups a list of alike objects by a common attribute.

`{% regroup %}` produces a list of *group objects*. Each group object has two attributes:

- `grouper`: The item that was grouped by (for example, the string India or Japan)
- `list`: A list of all items in this group (for example, a list of all cities with `country = "India"`)

Note that `{% regroup %}` does not order its input!

Any valid template lookup is a legal grouping attribute for the regroup tag, including methods, attributes, dictionary keys, and list items.

spaceless

Removes whitespace between HTML tags. This includes tab characters and newlines. Example usage:

```
{% spaceless %}  
<p>
```

```
<a href="foo/">Foo</a>
</p>
{%- endspaceless %}
```

This example would return this HTML:

```
<p><a href="foo/">Foo</a></p>
```

templatetag

Outputs one of the syntax characters used to compose template tags. Since the template system has no concept of escaping, to display one of the bits used in template tags, you must use the `{% templatetag %}` tag. The argument tells which template bit to output:

- `openblock` outputs: `{%`
- `closeblock` outputs: `%}`
- `openvariable` outputs: `{{`
- `closevariable` outputs: `}}`
- `openbrace` outputs: `{`
- `closebrace` outputs: `}`
- `opencomment` outputs: `{#`
- `closecomment` outputs: `#}`

Sample usage:

```
{% templatetag openblock %} url
'entry_list' {% templatetag closeblock %}
```

url

Returns an absolute path reference (a URL without the domain name) matching a given view function and optional parameters. Any special characters in the resulting path will be encoded using `iri_to_uri()`. This is a way to output links without violating the DRY principle by having to hard-code URLs in your templates:

```
{% url 'some-url-name' v1 v2 %}
```

The first argument is a path to a view function in the format `package.package.module.function`. It can be a quoted literal or any other context variable. Additional arguments are optional and should be space-separated values that will be used as arguments in the URL.

verbatim

Stops the template engine from rendering the contents of this block tag. A common use is to allow a Javascript template layer that collides with Django's syntax.

widtratio

For creating bar charts and such, this tag calculates the ratio of a given value to a maximum value, and then applies that ratio to a constant. For example:

```

```

with

Caches a complex variable under a simpler name. This is useful when accessing an expensive method (for example, one that hits the database) multiple times. For example:

```
{% with total=business.employees.count %}  
    {{ total }} employee{{ total|pluralize }}  
{% endwith %}
```

Builtin filters

add

Adds the argument to the value. For example:

```
{{ value|add:"2" }}
```

If `value` is 4, then the output will be 6.

addslashes

Adds slashes before quotes. Useful for escaping strings in CSV, for example. For example:

```
{{ value|addslashes }}
```

If `value` is I'm using Django, the output will be I'm using Django.

capfirst

Capitalizes the first character of the value. If the first character is not a letter, this filter has no effect.

center

Centers the value in a field of a given width. For example:

```
"{{ value|center:"14" }}"
```

If `value` is `Django`, the output will be `Django.`

cut

Removes all values of `arg` from the given string.

date

Formats a date according to the given format. Uses a similar format as PHP's `date()` function with some differences.

NOTE

These format characters are not used in Django outside of templates. They were designed to be compatible with PHP to ease transitioning for designers. For a full list of format strings see the Django Project website at <https://docs.djangoproject.com/en/dev/ref/templates/builtins/#date>.

For example:

```
{{ value|date:"D d M Y" }}
```

If `value` is a `datetime` object (for example, the result of `datetime.datetime.now()`), the output will be the string `Fri 01 Jul 2016`. The format passed can be one of the predefined ones `DATE_FORMAT`, `DATETIME_FORMAT`, `SHORT_DATE_FORMAT`, or `SHORT_DATETIME_FORMAT`, or a custom format that uses date format specifiers.

default

If value evaluates to `False`, uses the given default.

Otherwise, uses the value. For example:

```
{{ value|default:"nothing" }}
```

default_if_none

If (and only if) value is `None`, uses the given default.

Otherwise, uses the value.

dictsort

Takes a list of dictionaries and returns that list sorted by the key given in the argument. For example:

```
{{ value|dictsort:"name" }}
```

dictsortreversed

Takes a list of dictionaries and returns that list sorted in reverse order by the key given in the argument.

divisibleby

Returns `True` if the value is divisible by the argument.

For example:

```
{{ value|divisibleby:"3" }}
```

If `value` is `21`, the output would be `True`.

escape

Escapes a string's HTML. Specifically, it makes these replacements:

- `<` is converted to `<`;
- `>` is converted to `>`;
- `'` (single quote) is converted to `'`;
- `"` (double quote) is converted to `"`;
- `&` is converted to `&`;

The escaping is only applied when the string is output, so it does not matter where in a chained sequence of filters you put `escape`: it will always be applied as though it were the last filter.

escapejs

Escapes characters for use in JavaScript strings. This does *not* make the string safe for use in HTML, but does protect you from syntax errors when using templates to generate JavaScript/JSON.

filesizeformat

Formats the value like a 'human-readable' file size (that is, `'13 KB'`, `'4.1 MB'`, `'102 bytes'`, and more). For example:



```
 {{ value|filesizeformat }} 
```

If `value` is `123456789`, the output would be `117.7 MB`.

first

Returns the first item in a list.

floatformat

When used without an argument, rounds a floating-point number to one decimal place—but only if there's a decimal part to be displayed. If used with a numeric integer argument, `floatformat` rounds a number to that many decimal places.

For example, if `value` is `34.23234`, `{{ value|floatformat:3 }}` will output `34.232`.

get_digit

Given a whole number, returns the requested digit, where 1 is the right-most digit.

iriencode

Converts an **Internationalized Resource Identifier (IRI)** to a string that is suitable for including in a URL.

join

Joins a list with a string, like Python's `str.join(list)`.

last

Returns the last item in a list.

length

Returns the length of the value. This works for both strings and lists.

length_is

Returns `True` if the value's length is the argument, or `False` otherwise. For example:

```
 {{ value|length_is:"4" }}
```

linebreaks

Replaces line breaks in plain text with appropriate HTML; a single newline becomes an HTML line break (`
`) and a new line followed by a blank line becomes a paragraph break (`</p>`).

linebreaksbr

Converts all newlines in a piece of plain text to HTML line breaks (`
`).

linenumbers

Displays text with line numbers.

ljust

Left-aligns the value in a field of a given width. For example:

```
 {{ value|ljust:"10" }}
```

If `value` is `Django`, the output will be `Django`.

lower

Converts a string into all lowercase.

make_list

Returns the value turned into a list. For a string, it's a list of characters. For an integer, the argument is cast into an Unicode string before creating a list.

phone2numeric

Converts a phone number (possibly containing letters) to its numerical equivalent. The input doesn't have to be a valid phone number. This will happily convert any string. For example:

```
 {{ value|phone2numeric }}
```

If `value` is `800-COLLECT`, the output will be `800-2655328`.

pluralize

Returns a plural suffix if the value is not `1`. By default, this suffix is `s`.

For words that don't pluralize by simple suffix, you can specify both a singular and plural suffix, separated by a comma. Example:

```
You have {{ num_cherries }} cher{{  
    num_cherries|pluralize:"y,ies" }}.
```

pprint

A wrapper around `pprint.pprint()`--for debugging.

random

Returns a random item from the given list.

rjust

Right-aligns the value in a field of a given width. For example:

```
{{ value|rjust:"10" }}
```

If `value` is `Django`, the output will be `Django`.

safe

Marks a string as not requiring further HTML escaping prior to output. When autoescaping is off, this filter has no effect.

safeseq

Applies the `safe` filter to each element of a sequence. Useful in conjunction with other filters that operate on sequences, such as `join`. For example:

```
{% some_list|safeseq|join:", " %}
```

You couldn't use the `safe` filter directly in this case, as it would first convert the variable into a string, rather than working with the individual elements of the sequence.

slice

Returns a slice of the list. Uses the same syntax as Python's list slicing.

slugify

Converts to ASCII. Converts spaces to hyphens. Removes characters that aren't alphanumeric, underscores, or hyphens. Converts to lowercase. Also strips leading and trailing whitespace.

stringformat

Formats the variable according to the argument, a string formatting specifier. This specifier uses Python string formatting syntax, with the exception that the leading % is dropped.

striptags

Makes all possible efforts to strip all [X]HTML tags. For example:

```
 {{ value|striptags }} 
```

time

Formats a time according to the given format. Given format can be the predefined one [TIME_FORMAT](#), or a custom format, same as the [date](#) filter.

timesince

Formats a date as the time since that date (for example, 4 days, 6 hours). Takes an optional argument that is a variable containing the date to use as the comparison point (without the argument, the comparison point is [now](#)).

timeuntil

Measures the time from now until the given date or

[datetime](#).

title

Converts a string into title case by making words start with an uppercase character and the remaining characters lowercase.

truncatechars

Truncates a string if it is longer than the specified number of characters. Truncated strings will end with a translatable ellipsis sequence (...). For example:

```
 {{ value|truncatechars:9 }}
```

truncatechars_html

Similar to [truncatechars](#), except that it is aware of HTML tags.

truncatewords

Truncates a string after a certain number of words.

truncatewords_html

Similar to [truncatewords](#), except that it is aware of HTML tags.

unordered_list

Recursively takes a self-nested list and returns an HTML unordered list-without opening and closing tags.

upper

Converts a string into all uppercase.

urlencode

Escapes a value for use in a URL.

urlize

Converts URLs and email addresses in text into clickable links. This template tag works on links prefixed with `http://`, `https://`, or `www..`

urlizetrunc

Converts URLs and email addresses into clickable links just like `urlize`, but truncates URLs longer than the given character limit. For example:

```
{% value|urlizetrunc:15 %}
```

If `value` is `Check out www.djangoproject.com`, the output would be `Check out www.djangopr...`. As with

[urlize](#), this filter should only be applied to plain text.

wordcount

Returns the number of words.

wordwrap

Wraps words at specified line length.

yesno

Maps values for true, false and (optionally) None, to the strings yes, no, maybe, or a custom mapping passed as a comma-separated list, and returns one of those strings according to the value: For example:

```
 {{ value|yesno:"yeah,no,maybe" }}
```

Internationalization tags and filters

Django provides template tags and filters to control each aspect of internationalization in templates. They allow for granular control of translations, formatting, and time zone conversions.

i18n

This library allows specifying translatable text in templates. To enable it, set `USE_I18N` to `True`, then load it with `{% load i18n %}`.

l10n

This library provides control over the localization of values in templates. You only need to load the library using `{% load l10n %}`, but you'll often set `USE_L10N` to `True` so that localization is active by default.

tz

This library provides control over time zone conversions in templates. Like `l10n`, you only need to load the library using `{% load tz %}`, but you'll usually also set `USE_TZ` to `True` so that conversion to local time

happens by default. See [time-zones-in-templates](#).

Other tags and filters

libraries

static

To link to static files that are saved in `STATIC_ROOT` Django ships with a `static` template tag. You can use this regardless if you're using `RequestContext` or not.

```
{% load static %}
![Hi!]({% static )
```

It is also able to consume standard context variables, for example, assuming a `user_stylesheet` variable is passed to the template:

```
{% load static %}
<link href="{% static user_stylesheet %}" media="screen" rel="stylesheet" type="text/css" />
```

If you'd like to retrieve a static URL without displaying it, you can use a slightly different call:

```
{% load static %}
{{ static "images/hi.jpg" }} as myphoto
![Hi!]({{ myphoto }})
```

The `staticfiles` contrib app also ships with a

`static` template tag which uses `staticfiles STATICFILES_STORAGE` to build the URL of the given path (rather than simply using `urllib.parse.urljoin()` with the `STATIC_URL` setting and the given path). Use that instead if you have an advanced use case such as using a cloud service to serve static files: `{% load static from staticfiles %} `

get_static_prefix

You should prefer the `static` template tag, but if you need more control over exactly where and how `STATIC_URL` is injected into the template, you can use the `get_static_prefix` template tag: `{% load static %} `

There's also a second form you can use to avoid extra processing if you need the value multiple times:

```
{% load static %}
{% get_static_prefix as STATIC_PREFIX %}



```

get_media_prefix

Similar to the `get_static_prefix`,

`get_media_prefix` populates a template variable with the media prefix `MEDIA_URL`, for example:

```
<script type="text/javascript"
charset="utf-8">
var media_path = '{% get_media_prefix %}';
</script>
```

Django comes with a couple of other template-tag libraries that you have to enable explicitly in your `INSTALLED_APPS` setting and enable in your template with the `{% load %}` tag.

Appendix F. Request and Response Objects

Django uses request and response objects to pass state through the system.

When a page is requested, Django creates an [HttpRequest](#) object that contains metadata about the request. Then Django loads the appropriate view, passing the [HttpRequest](#) as the first argument to the view function. Each view is responsible for returning an [HttpResponse](#) object.

This document explains the APIs for [HttpRequest](#) and [HttpResponse](#) objects, which are defined in the `django.http` module.

HttpRequest objects

Attributes

All attributes should be considered read-only, unless stated otherwise below. [session](#) is a notable exception.

HttpRequest.scheme

A string representing the scheme of the request ([http](#) or [https](#) usually).

HttpRequest.body

The raw HTTP request body as a byte string. This is useful for processing data in different ways than conventional HTML forms: binary images, XML payload etc. For processing conventional form data, use [HttpRequest.POST](#).

You can also read from an `HttpRequest` using a file-like interface. See [HttpRequest.read\(\)](#).

HttpRequest.path

A string representing the full path to the requested page, not including the domain.

Example: [musicbands/the_beatles/](#)

HttpRequest.path_info

Under some web server configurations, the portion of the URL after the host name is split up into a script prefix portion and a path info portion. The [path_info](#) attribute always contains the path info portion of the path, no matter what web server is being used. Using this instead of [path](#) can make your code easier to move between test and deployment servers.

For example, if the [WSGIScriptAlias](#) for your application is set to [/minfo](#), then [path](#) might be [/minfo](#)[musicbands/the_beatles/](#) and [path_info](#)

would be `musicbands/the_beatles/`.

HttpRequest.method

A string representing the HTTP method used in the request. This is guaranteed to be uppercase. Example:

```
if request.method == 'GET':  
    do_something() elif request.method ==  
'POST':  
    do_something_else()
```

HttpRequest.encoding

A string representing the current encoding used to decode form submission data (or `None`, which means the `DEFAULT_CHARSET` setting is used). You can write to this attribute to change the encoding used when accessing the form data.

Any subsequent attribute accesses (such as reading from `GET` or `POST`) will use the new `encoding` value. Useful if you know the form data is not in the `DEFAULT_CHARSET` encoding.

HttpRequest.GET

A dictionary-like object containing all given HTTP `GET` parameters. See the [QueryDict](#) documentation below.

HttpRequest.POST

A dictionary-like object containing all given HTTP `POST` parameters, providing that the request contains form data. See the [QueryDict](#) documentation below.

If you need to access raw or non-form data posted in the request, access this through the `HttpRequest.body` attribute instead.

It's possible that a request can come in via `POST` with an empty `POST` dictionary-if, say, a form is requested via the `POST` HTTP method but does not include form data. Therefore, you shouldn't use `if request.POST` to check for use of the `POST` method; instead, use `if request.method == 'POST'` (see above).

Note: `POST` does *not* include file-upload information. See [FILES](#).

HttpRequest.COOKIES

A standard Python dictionary containing all cookies. Keys and values are strings.

HttpRequest.FILES

A dictionary-like object containing all uploaded files. Each key in `FILES` is the `name` from the `<input type="file" name="" />`. Each value in `FILES` is an [UploadedFile](#).

Note that `FILES` will only contain data if the request

method was `POST` and the `<form>` that posted to the request had `enctype="multipart/form-data"`. Otherwise, `FILES` will be a blank dictionary-like object.

HttpRequest.META

A standard Python dictionary containing all available HTTP headers. Available headers depend on the client and server, but here are some examples:

- `CONTENT_LENGTH`: The length of the request body (as a string)
- `CONTENT_TYPE`: The MIME type of the request body
- `HTTP_ACCEPT_ENCODING`: Acceptable encodings for the response
- `HTTP_ACCEPT_LANGUAGE`: Acceptable languages for the response
- `HTTP_HOST`: The HTTP Host header sent by the client
- `HTTP_REFERER`: The referring page, if any
- `HTTP_USER_AGENT`: The client's user-agent string
- `QUERY_STRING`: The query string, as a single (unparsed) string
- `REMOTE_ADDR`: The IP address of the client
- `REMOTE_HOST`: The hostname of the client
- `REMOTE_USER`: The user authenticated by the web server, if any
- `REQUEST_METHOD`: A string such as `"GET"` or `"POST"`
- `SERVER_NAME`: The hostname of the server
- `SERVER_PORT`: The port of the server (as a string)

With the exception of `CONTENT_LENGTH` and `CONTENT_TYPE`, as given above, any HTTP headers in the request are converted to `META` keys by converting all characters to uppercase, replacing any hyphens with underscores and adding an `HTTP_` prefix to the name.

So, for example, a header called `X-Bender` would be mapped to the `META` key `HTTP_X_BENDER`.

HttpRequest.user

An object of type `AUTH_USER_MODEL` representing the currently logged-in user. If the user isn't currently logged in, `user` will be set to an instance of `django.contrib.auth.models.AnonymousUser`. You can tell them apart with `is_authenticated()`, like so:

```
if request.user.is_authenticated():
    # Do something for logged-in users.
else:
    # Do something for anonymous users.
```

`user` is only available if your Django installation has the `AuthenticationMiddleware` activated.

HttpRequest.session

A readable-and-writable, dictionary-like object that represents the current session. This is only available if your Django installation has session support activated.

HttpRequest.urlconf

Not defined by Django itself, but will be read if other code (for example, a custom middleware class) sets it. When present, this will be used as the root URLconf for the current request, overriding the `ROOT_URLCONF`

setting.

HttpRequest.resolver_match

An instance of `ResolverMatch` representing the resolved url. This attribute is only set after url resolving took place, which means it's available in all views but not in middleware methods which are executed before url resolving takes place (like `process_request`, you can use `process_view` instead).

Methods

HttpRequest.get_host()

Returns the originating host of the request using information from the `HTTP_X_FORWARDED_HOST` (if `USE_X_FORWARDED_HOST` is enabled) and `HTTP_HOST` headers, in that order. If they don't provide a value, the method uses a combination of `SERVER_NAME` and `SERVER_PORT` as detailed in PEP 3333.

Example: `127.0.0.1:8000`

Note

The `get_host()` method fails when the host is behind multiple proxies. One solution is to use middleware to rewrite the proxy headers, as in the following example:

```
class MultipleProxyMiddleware(object):
    FORWARDED_FOR_FIELDS = [
```

```
'HTTP_X_FORWARDED_FOR',
'HTTP_X_FORWARDED_HOST',
'HTTP_X_FORWARDED_SERVER',
]
def process_request(self, request):
    """
        Rewrites the proxy headers so
        that only the most
        recent proxy is used.
    """
    for field in
self.FORWARDED_FOR_FIELDS:
        if field in request.META:
            if ',' in
request.META[field]:
                parts =
request.META[field].split(',')
                request.META[field] =
parts[-1].strip()
```

This middleware should be positioned before any other middleware that relies on the value of `get_host()`-for instance, [CommonMiddleware](#) or [CsrfViewMiddleware](#).

HttpRequest.get_full_path()

Returns the `path`, plus an appended query string, if applicable.

Example: `musicbands/the_beatles/?print=true`

HttpRequest.build_absolute_uri(location)

Returns the absolute URI form of `location`. If no location is provided, the location will be set to

```
request.get_full_path().
```

If the location is already an absolute URI, it will not be altered. Otherwise the absolute URI is built using the server variables available in this request.

Example:

```
http://example.com/musicbands/the_beatles/?  
print=true
```

HttpRequest.get_signed_cookie()

Returns a cookie value for a signed cookie, or raises a `django.core.signing.BadSignature` exception if the signature is no longer valid. If you provide the `default` argument the exception will be suppressed and that default value will be returned instead.

The optional `salt` argument can be used to provide extra protection against brute force attacks on your secret key. If supplied, the `max_age` argument will be checked against the signed timestamp attached to the cookie value to ensure the cookie is not older than `max_age` seconds.

For example:

```
>>> request.get_signed_cookie('name')  
'Tony'  
>>> request.get_signed_cookie('name',  
salt='name-salt')  
'Tony' # assuming cookie was set using the
```

```
same salt
>>> request.get_signed_cookie('non-
existing-cookie')
...
KeyError: 'non-existing-cookie'
>>> request.get_signed_cookie('non-
existing-cookie', False)
False
>>> request.get_signed_cookie('cookie-
that-was-tampered-with')
...
BadSignature: ...
>>> request.get_signed_cookie('name',
max_age=60)
...
SignatureExpired: Signature age
1677.3839159 > 60 seconds
>>> request.get_signed_cookie('name',
False, max_age=60)
False
```

HttpRequest.is_secure()

Returns `True` if the request is secure; that is, if it was made with HTTPS.

HttpRequest.is_ajax()

Returns `True` if the request was made via an `XMLHttpRequest`, by checking the `HTTP_X_REQUESTED_WITH` header for the string `"XMLHttpRequest"`. Most modern JavaScript libraries send this header. If you write your own `XMLHttpRequest` call (on the browser side), you'll have to set this header manually if you want `is_ajax()` to work.

If a response varies on whether or not it's requested via AJAX and you are using some form of caching like Django's [cache middleware](#), you should decorate the view with `vary_on_headers('HTTP_X_REQUESTED_WITH')` so that the responses are properly cached.

HttpRequest.read(size=None)

HttpRequest.readline()

HttpRequest.readlines()

HttpRequest.xreadlines()

HttpRequest.__iter__()

Methods implementing a file-like interface for reading from an [HttpRequest](#) instance. This makes it possible to consume an incoming request in a streaming fashion. A common use-case would be to process a big XML payload with iterative parser without constructing a whole XML tree in memory.

Given this standard interface, an [HttpRequest](#) instance can be passed directly to an XML parser such as [ElementTree](#):

```
import xml.etree.ElementTree as ET
for element in ET.iterparse(request):
    process(element)
```

QueryDict objects

In an `HttpRequest` object, the `GET` and `POST` attributes are instances of `django.http.QueryDict`, a dictionary-like class customized to deal with multiple values for the same key. This is necessary because some HTML form elements, notably `<select multiple>`, pass multiple values for the same key.

The `QueryDict`s at `request.POST` and `request.GET` will be immutable when accessed in a normal request/response cycle. To get a mutable version you need to use `.copy()`.

Methods

`QueryDict` implements all the standard dictionary methods because it's a subclass of `dict`, with the following exceptions.

`QueryDict.__init__()`

Instantiates a `QueryDict` object based on `query_string`.

```
>>> QueryDict('a=1&a=2&c=3')
<QueryDict: {'a': ['1', '2'], 'c': ['3']}>
```

If `query_string` is not passed in, the resulting

`QueryDict` will be empty (it will have no keys or values).

Most `QueryDict`s you encounter, and in particular those at `request.POST` and `request.GET`, will be immutable. If you are instantiating one yourself, you can make it mutable by passing `mutable=True` to its `__init__()`.

Strings for setting both keys and values will be converted from `encoding` to Unicode. If encoding is not set, it defaults to `DEFAULT_CHARSET`.

`QueryDict.__getitem__(key)`

Returns the value for the given key. If the key has more than one value, `__getitem__()` returns the last value.

Raises

`django.utils.datastructures.MultiValueDictKeyError` if the key does not exist.

`QueryDict.__setitem__(key, value)`

Sets the given key to `[value]` (a Python list whose single element is `value`). Note that this, as other dictionary functions that have side effects, can only be called on a mutable `QueryDict` (such as one that was created via `copy()`).

`QueryDict.__contains__(key)`

Returns `True` if the given key is set. This lets you do, for example, `if "foo" in request.GET`.

`QueryDict.get(key, default)`

Uses the same logic as `__getitem__()` above, with a hook for returning a default value if the key doesn't exist.

`QueryDict.setdefault(key, default)`

Just like the standard dictionary `setdefault()` method, except it uses `__setitem__()` internally.

`QueryDict.update(other_dict)`

Takes either a `QueryDict` or standard dictionary. Just like the standard dictionary `update()` method, except it appends to the current dictionary items rather than replacing them. For example:

```
>>> q = QueryDict('a=1', mutable=True)
>>> q.update({'a': '2'})
>>> q.getlist('a')
['1', '2']
>>> q['a'] # returns the last
['2']
```

`QueryDict.items()`

Just like the standard dictionary `items()` method, except this uses the same last-value logic as `__getitem__()`. For example:

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.items()
[('a', '3')]
```

QueryDict.iteritems()

Just like the standard dictionary `iteritems()` method.

Like `QueryDict.items()` this uses the same last-value logic as `QueryDict.__getitem__()`.

QueryDict.iterlists()

Like `QueryDict.iteritems()` except it includes all values, as a list, for each member of the dictionary.

QueryDict.values()

Just like the standard dictionary `values()` method, except this uses the same last-value logic as `__getitem__()`. For example:

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.values()
['3']
```

QueryDict.itervalues()

Just like `QueryDict.values()`, except an iterator.

In addition, `QueryDict` has the following methods:

QueryDict.copy()

Returns a copy of the object, using `copy.deepcopy()` from the Python standard library. This copy will be mutable even if the original was not.

QueryDict.getlist(key, default)

Returns the data with the requested key, as a Python list. Returns an empty list if the key doesn't exist and no default value was provided. It's guaranteed to return a list of some sort unless the default value was no list.

QueryDict.setlist(key, list)

Sets the given key to `list_` (unlike `__setitem__()`).

QueryDict.appendlist(key, item)

Appends an item to the internal list associated with key.

QueryDict.setlistdefault(key, default_list)

Just like `setdefault`, except it takes a list of values instead of a single value.

QueryDict.lists()

Like `items()`, except it includes all values, as a list, for each member of the dictionary. For example:

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.lists()
[('a', ['1', '2', '3'])]
```

QueryDict.pop(key)

Returns a list of values for the given key and removes them from the dictionary. Raises `KeyError` if the key does not exist. For example:

```
>>> q = QueryDict('a=1&a=2&a=3',
    mutable=True)
>>> q.pop('a')
['1', '2', '3']
```

QueryDict.popitem()

Removes an arbitrary member of the dictionary (since there's no concept of ordering), and returns a two value tuple containing the key and a list of all values for the key. Raises `KeyError` when called on an empty dictionary. For example:

```
>>> q = QueryDict('a=1&a=2&a=3',
    mutable=True)
>>> q.popitem()
('a', ['1', '2', '3'])
```

QueryDict.dict()

Returns `dict` representation of `QueryDict`. For every (key, list) pair in `QueryDict`, `dict` will have (key, item), where item is one element of the list, using same logic as `QueryDict.__getitem__()`:

```
>>> q = QueryDict('a=1&a=3&a=5')
>>> q.dict()
```

```
{'a': '5'}
```

QueryDict.urlencode([safe])

Returns a string of the data in query-string format.

Example:

```
>>> q = QueryDict('a=2&b=3&b=5')
>>> q.urlencode()
'a=2&b=3&b=5'
```

Optionally, urlencode can be passed characters which do not require encoding. For example:

```
>>> q = QueryDict(mutable=True)
>>> q['next'] = 'a&b'
>>> q.urlencode(safe=' ')
'next=a%26b/'
```

HttpResponse objects

In contrast to [HttpRequest](#) objects, which are created automatically by Django, [HttpResponse](#) objects are your responsibility. Each view you write is responsible for instantiating, populating and returning an [HttpResponse](#).

The [HttpResponse](#) class lives in the `django.http` module.

Usage

Passing strings

Typical usage is to pass the contents of the page, as a string, to the [HttpResponse](#) constructor:

```
>>> from  
django.http import HttpResponse  
>>> response =  
HttpResponse("Here's the text of the Web page.")  
>>> response = HttpResponse("Text only, please.",  
content_type="text/plain")
```

But if you want to add content incrementally, you can use `response` as a file-like object:

```
>>> response =  
HttpResponse()  
>>> response.write("<p>Here's the  
text of the Web page.</p>")  
>>> response.write("<p>Here's another paragraph.</p>")
```

Passing iterators

Finally, you can pass `HttpResponse` an iterator rather than strings. `HttpResponse` will consume the iterator immediately, store its content as a string, and discard it.

If you need the response to be streamed from the iterator to the client, you must use the `StreamingHttpResponse` class instead.

Setting header fields

To set or remove a header field in your response, treat it like a dictionary:

```
>>> response = HttpResponse()
>>> response['Age'] = 120
>>> del response['Age']
```

Note that unlike a dictionary, `del` doesn't raise `KeyError` if the header field doesn't exist.

For setting the `Cache-Control` and `Vary` header fields, it is recommended to use the `patch_cache_control()` and `patch_vary_headers()` methods from `django.utils.cache`, since these fields can have multiple, comma-separated values. The patch methods ensure that other values, for example, added by a middleware, are not removed.

HTTP header fields cannot contain newlines. An attempt to set a header field containing a newline character (CR or LF) will raise `BadHeaderError`.

Telling the browser to treat the response as a file attachment

To tell the browser to treat the response as a file attachment, use the `content_type` argument and set the `Content-Disposition` header. For example, this is how you might return a Microsoft Excel spreadsheet:

```
>>> response = HttpResponseRedirect(my_data,
content_type='application/vnd.ms-excel') >>>
response['Content-Disposition'] = 'attachment;
filename="foo.xls"'
```

There's nothing Django-specific about the `Content-Disposition` header, but it's easy to forget the syntax, so we've included it here.

Attributes

`HttpResponse.content`

A bytestring representing the content, encoded from a Unicode object if necessary.

`HttpResponse.charset`

A string denoting the charset in which the response will be encoded. If not given at `HttpResponse` instantiation time, it will be extracted from `content_type` and if that is unsuccessful, the `DEFAULT_CHARSET` setting will be used.

HttpResponse.status_code

The HTTP status code for the response.

HttpResponse.reason_phrase

The HTTP reason phrase for the response.

HttpResponse.streaming

This is always **False**.

This attribute exists so middleware can treat streaming responses differently from regular responses.

HttpResponse.closed

True if the response has been closed.

Methods

HttpResponse.__init__()

```
HttpResponse.__init__(content='',
                     content_type=None, status=200,
                     reason=None, charset=None)
```

Instantiates an **HttpResponse** object with the given page content and content type. **content** should be an iterator or a string. If it's an iterator, it should return strings, and those strings will be joined together to form the content of the response. If it is not an iterator or a

string, it will be converted to a string when accessed.

Has four parameters:

- `content_type` is the MIME type optionally completed by a character set encoding and is used to fill the HTTP `Content-Type` header. If not specified, it is formed by the `DEFAULT_CONTENT_TYPE` and `DEFAULT_CHARSET` settings, by default: `text/html; charset=utf-8`.
- `status` is the HTTP status code for the response.
- `reason` is the HTTP response phrase. If not provided, a default phrase will be used.
- `charset` is the charset in which the response will be encoded. If not given it will be extracted from `content_type`, and if that is unsuccessful, the `DEFAULT_CHARSET` setting will be used.

HttpResponse.__setitem__(header, value)

Sets the given header name to the given value. Both `header` and `value` should be strings.

HttpResponse.__delitem__(header)

Deletes the header with the given name. Fails silently if the header doesn't exist. Case-insensitive.

HttpResponse.__getitem__(header)

Returns the value for the given header name. Case-insensitive.

HttpResponse.has_header(header)

Returns `True` or `False` based on a case-insensitive check for a header with the given name.

`HttpResponse.setdefault(header, value)`

Sets a header unless it has already been set.

`HttpResponse.set_cookie()`

```
HttpResponse.set_cookie(key, value='',
    max_age=None, expires=None, path='/',
    domain=None, secure=None,
    httponly=False)
```

Sets a cookie. The parameters are the same as in the [Morsel](#) cookie object in the Python standard library.

- `max_age` should be a number of seconds, or `None` (default) if the cookie should last only as long as the client's browser session. If `expires` is not specified, it will be calculated.
- `expires` should either be a string in the format `"Wdy, DD-Mon-YY HH:MM:SS GMT"` or a `datetime.datetime` object in UTC. If `expires` is a `datetime` object, the `max_age` will be calculated.
- Use `domain` if you want to set a cross-domain cookie. For example, `domain=".lawrence.com"` will set a cookie that is readable by the domains `www.lawrence.com`, `blogs.lawrence.com` and `calendars.lawrence.com`. Otherwise, a cookie will only be readable by the domain that set it.
- Use `httponly=True` if you want to prevent client-side JavaScript from having access to the cookie.

`HTTPOnly` is a flag included in a Set-Cookie HTTP response header. It is not part of the RFC 2109 standard for cookies, and it isn't honored consistently by all browsers. However, when it is honored, it can be a useful way to mitigate the risk of client side script accessing the protected cookie data.

HttpResponse.set_signed_cookie()

Like [set_cookie\(\)](#), but cryptographic signing the cookie before setting it. Use in conjunction with [HttpRequest.get_signed_cookie\(\)](#). You can use the optional [salt](#) argument for added key strength, but you will need to remember to pass it to the corresponding [HttpRequest.get_signed_cookie\(\)](#) call.

HttpResponse.delete_cookie()

Deletes the cookie with the given key. Fails silently if the key doesn't exist.

Due to the way cookies work, [path](#) and [domain](#) should be the same values you used in [set_cookie\(\)](#)- otherwise the cookie may not be deleted.

[HttpResponse.write\(content\)](#)

HttpResponse.flush()

HttpResponse.tell()

These methods implement a file-like interface with an [HttpResponse](#). They work the same way as the corresponding Python file method.

HttpResponse.getvalue()

Returns the value of [HttpResponse.content](#). This

method makes an [HttpResponse](#) instance a stream-like object.

HttpResponse.writable()

Always [True](#). This method makes an [HttpResponse](#) instance a stream-like object.

HttpResponse.writelines(lines)

Writes a list of lines to the response. Line separators are not added. This method makes an [HttpResponse](#) instance a stream-like object.

HttpResponse subclasses

Django includes a number of [HttpResponse](#) subclasses that handle different types of HTTP responses. Like [HttpResponse](#), these subclasses live in [django.http](#).

HttpResponseRedirect

The first argument to the constructor is required—the path to redirect to. This can be a fully qualified URL (for example, <http://www.yahoo.comsearch>) or an absolute path with no domain (for example, [search](#)). See [HttpResponse](#) for other optional constructor arguments. Note that this returns an HTTP status code 302.

HttpResponsePermanentRedirect

Like [HttpResponseRedirect](#), but it returns a permanent redirect (HTTP status code 301) instead of a found redirect (status code 302).

HttpResponseNotModified

The constructor doesn't take any arguments and no content should be added to this response. Use this to designate that a page hasn't been modified since the user's last request (status code 304).

HttpResponseBadRequest

Acts just like [HttpResponse](#) but uses a 400 status code.

HttpResponseNotFound

Acts just like [HttpResponse](#) but uses a 404 status code.

HttpResponseForbidden

Acts just like [HttpResponse](#) but uses a 403 status code.

HttpResponseNotAllowed

Like [HttpResponse](#), but uses a 405 status code. The first argument to the constructor is required: a list of

permitted methods (for example, `['GET', 'POST']`).

HttpResponseGone

Acts just like [HttpResponse](#) but uses a 410 status code.

HttpResponseServerError

Acts just like [HttpResponse](#) but uses a 500 status code.

If a custom subclass of [HttpResponse](#) implements a `render` method, Django will treat it as emulating a [SimpleTemplateResponse](#), and the `render` method must itself return a valid response object.

JsonResponse Objects

```
class JsonResponse(data,  
encoder=DjangoJSONEncoder, safe=True,  
**kwargs)
```

An `HttpResponse` subclass that helps to create a JSON-encoded response. It inherits most behavior from its superclass with some differences:

- Its default `Content-Type` header is set to `application/json`.
- The first parameter, `data`, should be a `dict` instance. If the `safe` parameter is set to `False` (see below) it can be any JSON-serializable object.
- The `encoder`, which defaults to `djangocore.serializers.json.DjangoJSONEncoder`, will be used to serialize the data.

The `safe` boolean parameter defaults to `True`. If it's set to `False`, any object can be passed for serialization (otherwise only `dict` instances are allowed). If `safe` is `True` and a non-`dict` object is passed as the first argument, a `TypeError` will be raised.

Usage

Typical usage could look like:

```
>>> from django.http import JsonResponse  
>>> response = JsonResponse({'foo':  
'bar'}) >>> response.content '{"foo":
```

```
"bar"}'
```

Serializing non-dictionary objects

In order to serialize objects other than `dict` you must set the `safe` parameter to `False`:

```
response = JsonResponse([1, 2, 3],  
safe=False)
```

Without passing `safe=False`, a `TypeError` will be raised.

Changing the default JSON encoder

If you need to use a different JSON encoder class, you can pass the `encoder` parameter to the constructor method:

```
response = JsonResponse(data,  
encoder=MyJSONEncoder)
```

StreamingHttpResponse objects

The `StreamingHttpResponse` class is used to stream a response from Django to the browser. You might want to do this if generating the response takes too long or uses too much memory. For instance, it's useful for generating large CSV files.

Performance considerations

Django is designed for short-lived requests. Streaming responses will tie a worker process for the entire duration of the response. This may result in poor performance.

Generally speaking, you should perform expensive tasks outside of the request-response cycle, rather than resorting to a streamed response.

The `StreamingHttpResponse` is not a subclass of `HttpResponse`, because it features a slightly different API. However, it is almost identical, with the following notable differences:

- It should be given an iterator that yields strings as content.
- You cannot access its content, except by iterating the response object itself. This should only occur when the response is returned to the client.

- It has no `content` attribute. Instead, it has a `streaming_content` attribute.
- You cannot use the file-like object `tell()` or `write()` methods.
Doing so will raise an exception.

`StreamingHttpResponse` should only be used in situations where it is absolutely required that the whole content isn't iterated before transferring the data to the client. Because the content can't be accessed, many middlewares can't function normally. For example, the `ETag` and `Content-Length` headers can't be generated for streaming responses.

Attributes

`StreamingHttpResponse` has the following attributes:

- `* * .streaming_content`. An iterator of strings representing the content.
- `* * .status_code`. The HTTP status code for the response.
- `* * .reason_phrase`. The HTTP reason phrase for the response.
- `* * .streaming`. This is always `True`.

FileResponse objects

`FileResponse` is a subclass of `StreamingHttpResponse` optimized for binary files. It uses `wsgi.file_wrapper` if provided by the wsgi server, otherwise it streams the file out in small chunks.

`FileResponse` expects a file open in binary mode like so:

```
>>> from django.http import FileResponse  
>>> response =  
FileResponse(open('myfile.png', 'rb'))
```

Error views

Django comes with a few views by default for handling HTTP errors. To override these with your own custom views, see [customizing-error-views](#).

The 404 (page not found) view

```
defaults.page_not_found(request,  
template_name='404.html')
```

When you raise `Http404` from within a view, Django loads a special view devoted to handling 404 errors. By default, it's the view

`django.views.defaults.page_not_found()`, which either produces a very simple Not Found message or loads and renders the template `404.html` if you created it in your root template directory.

The default 404 view will pass one variable to the template: `request_path`, which is the URL that resulted in the error.

Three things to note about 404 views:

- The 404 view is also called if Django doesn't find a match after checking every regular expression in the URLconf.
- The 404 view is passed a `RequestContext` and will have access to variables supplied by your template context processors (for example, `MEDIA_URL`).

- If `DEBUG` is set to `True` (in your settings module), then your 404 view will never be used, and your URLconf will be displayed instead, with some debug information.

The 500 (server error) view

```
defaults.server_error(request,  
template_name='500.html')
```

Similarly, Django executes special-case behavior in the case of runtime errors in view code. If a view results in an exception, Django will, by default, call the view `django.views.defaults.server_error`, which either produces a very simple Server Error message or loads and renders the template `500.html` if you created it in your root template directory.

The default 500 view passes no variables to the `500.html` template and is rendered with an empty `Context` to lessen the chance of additional errors.

If `DEBUG` is set to `True` (in your settings module), then your 500 view will never be used, and the traceback will be displayed instead, with some debug information.

The 403 (HTTP Forbidden) view

```
defaults.permission_denied(request,  
template_name='403.html')
```

In the same vein as the 404 and 500 views, Django has a view to handle 403 Forbidden errors. If a view results

in a 403 exception then Django will, by default, call the view

`django.views.defaults.permission_denied`.

This view loads and renders the template `403.html` in your root template directory, or if this file does not exist, instead serves the text 403 Forbidden, as per RFC 2616 (the HTTP 1.1 Specification).

`django.views.defaults.permission_denied` is triggered by a `PermissionDenied` exception. To deny access in a view you can use code like this:

```
from django.core.exceptions import  
PermissionDenied  
  
def edit(request, pk):  
    if not request.user.is_staff:  
        raise PermissionDenied  
    # ...
```

The 400 (bad request) view

```
defaults.bad_request(request,  
template_name='400.html')
```

When a `SuspiciousOperation` is raised in Django, it may be handled by a component of Django (for example resetting the session data). If not specifically handled, Django will consider the current request a 'bad request' instead of a server error.

`django.views.defaults.bad_request`, is

otherwise very similar to the `server_error` view, but returns with the status code 400 indicating that the error condition was the result of a client operation.

`bad_request` views are also only used when `DEBUG` is `False`.

Customizing error views

The default error views in Django should suffice for most web applications, but can easily be overridden if you need any custom behavior. Simply specify the handlers as seen below in your URLconf (setting them anywhere else will have no effect).

The `page_not_found()` view is overridden by `handler404`:

```
handler404 =
'mysite.views.my_custom_page_not_found_view'
```

The `server_error()` view is overridden by `handler500`:

```
handler500 =
'mysite.views.my_custom_error_view'
```

The `permission_denied()` view is overridden by `handler403`:

```
handler403 =
'mysite.views.my_custom_permission_denied_view'
```

The `bad_request()` view is overridden by `handler400`:

```
handler400 =  
'mysite.views.my_custom_bad_request_view'
```

Appendix G. Developing Django with Visual Studio

Regardless of what you might hear trolling around the Internet, Microsoft Visual Studio (VS) has always been an extremely capable and powerful Integrated Development Environment (IDE). As a developer for multiple platforms, I have dabbled in just about everything else out there and have always ended up back with VS.

The biggest barriers to wider uptake of VS in the past have been (in my opinion):

- Lack of good support for languages outside of Microsoft's ecosystem (C++, C# and VB)
- Cost of the fully featured IDE. Previous incarnations of Microsoft 'free' IDE's have fallen a bit short of being useful for professional development

With the release of Visual Studio Community Editions a few years ago and the more recent release of Python Tools for Visual Studio (PTVS), this situation has changed dramatically for the better. So much so that I now do all my development in VS-both Microsoft technologies and Python and Django.

I am not going to go on with the virtues of VS, lest I begin to sound like a commercial for Microsoft, so let's assume that you have at least decided to give VS and

PTVS a go.

Firstly, I will explain how to install VS and PTVS on your Windows box and then I will give you a quick overview of all the cool Django and Python tools that you have at your disposal.

Installing Visual Studio

NOTE

Before you start

Because it's still Microsoft, we can't get past the fact that VS is a big install.

To minimize the chances of grief, please:

1. Turn off your antivirus for the duration of the install
2. Make sure you have a good Internet connection. Wired is better than wireless
3. Pause other memory/disk hogs like OneDrive and Dropbox
4. Close every application that doesn't have to be open

Once you have taken careful note of the preceding warning, jump on to the Visual Studio website (<https://www.visualstudio.com/>) and download the free Visual Studio Community Edition 2015 (*Figure G.1*):



Figure G.1: Visual Studio Downloads

Launch the downloaded installer file, make sure the default install option is selected (*Figure G.2*) and click install:

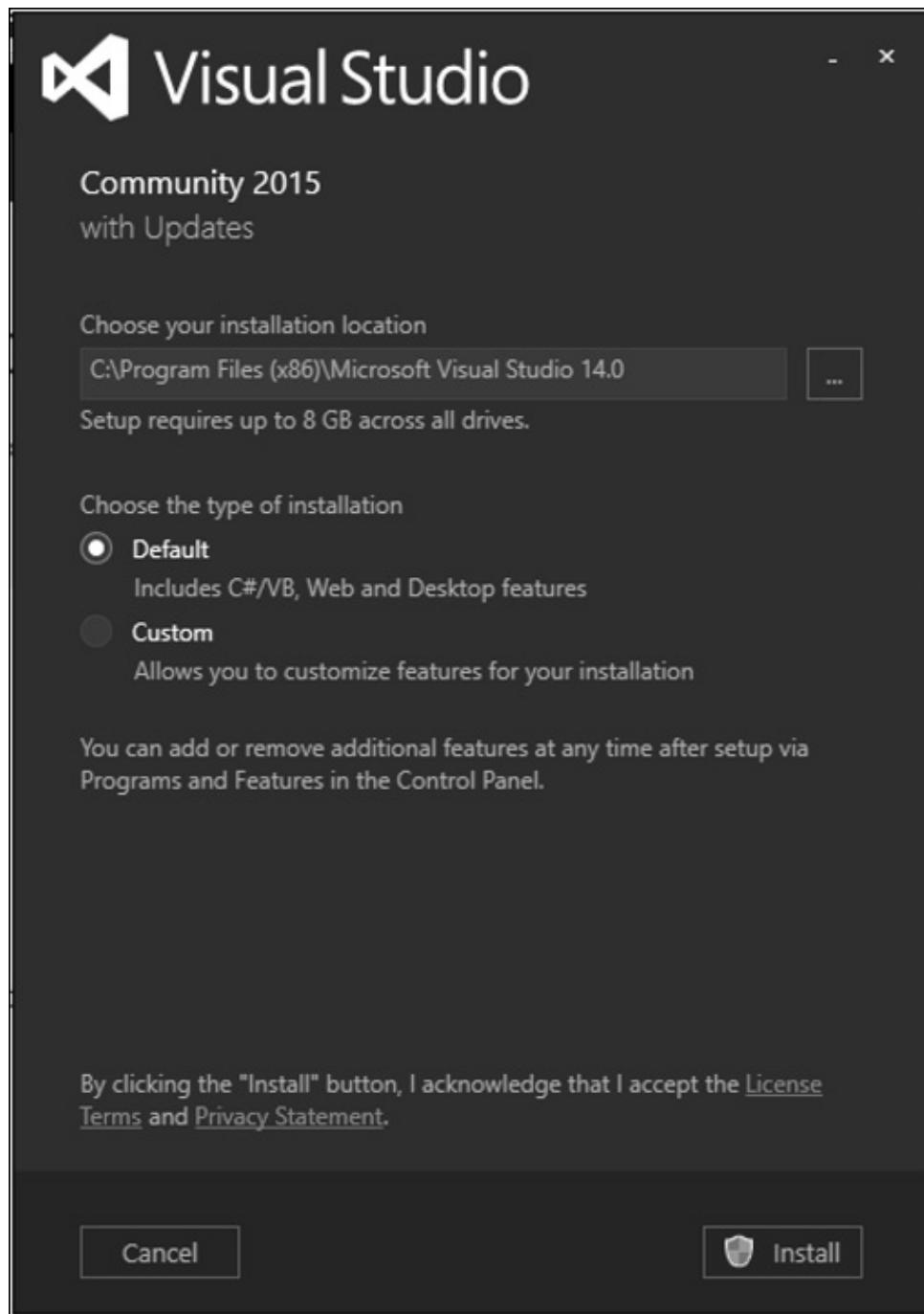


Figure G.2: Visual Studio's default install

Now's the time to go make yourself a coffee. Or seven.
Microsoft, remember-it's going to take a while.
Depending on your Internet connection this can take
anywhere from 15 minutes to more than an hour.

In a few rare cases it will fail. This is always (in my
experience) either forgetting to turn antivirus off or a
momentary dropout in your Internet connection. Luckily
VS's recovery process is pretty robust and I have found
rebooting and restarting the install after a failure works
every time. VS will even remember where it's up to, so
you don't have to start all over again.

Install PTVS and Web Essentials

Once you have installed VS, it's time to add Python
Tools for Visual Studio (PTVS) and Visual Studio Web
Essentials. From the top menu, select [Tools >
Extensions and Updates](#) (*Figure G.3*):

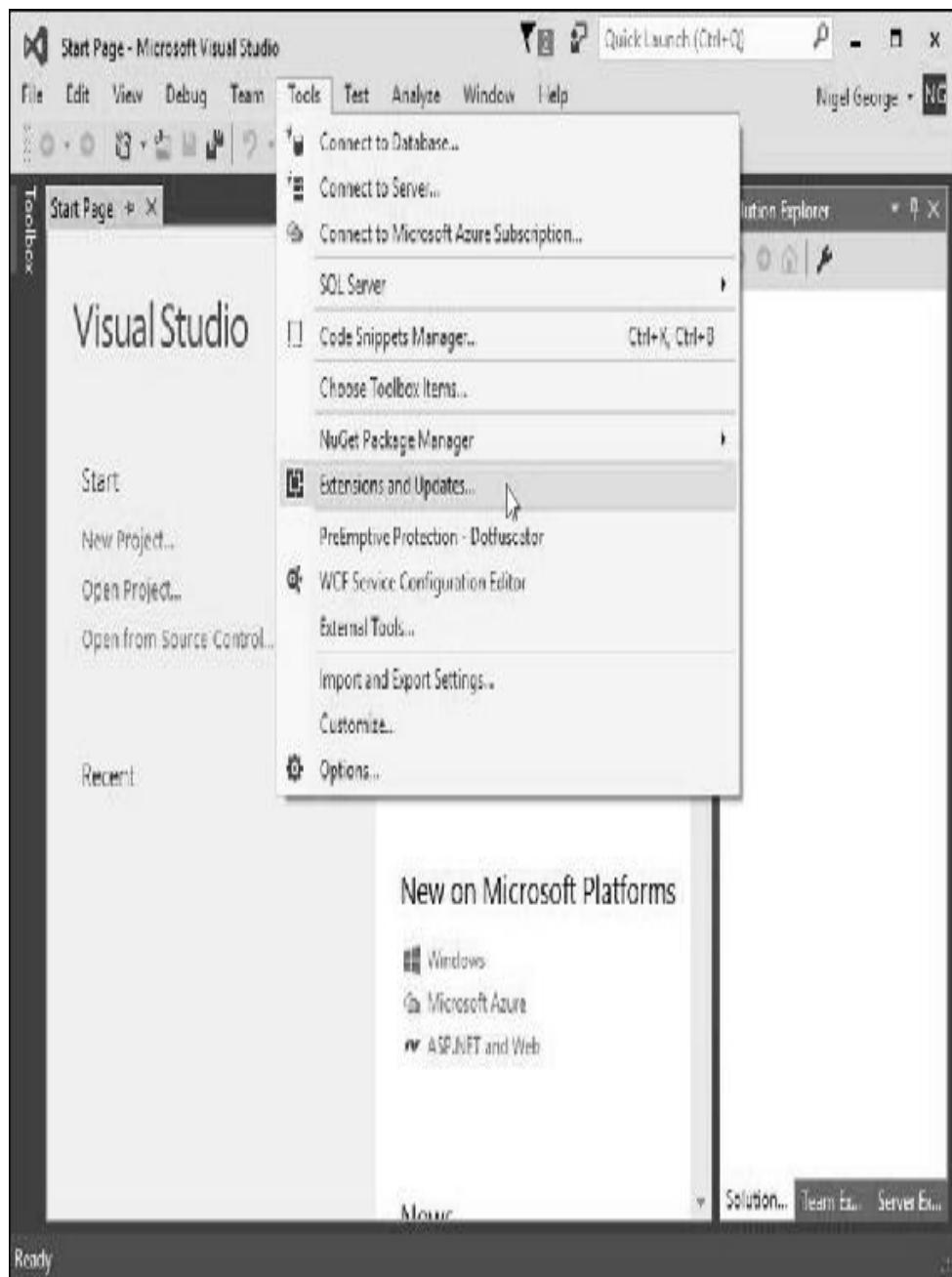


Figure G.3: Install Visual Studio's extension

Once the Extensions and Updates window opens, select **Online** from the dropdown on the left to go to the VS online application gallery. Type **python** in the search box on the top right and the PTVS extension should

appear on the top of the list (*Figure G.4*):



Figure G.4: Install PTVS extension

Repeat the same process for VS Web Essentials (*Figure G.5*). Note that, depending on the VS build and what

extensions have been installed previously, Web Essentials may already be installed. If this is the case, the **Download** button will be replaced with a green tick icon:



Figure G.5: Install Web Essentials extension

Creating A Django project

One of the great things about using VS for Django development is that the only thing you need to install other than VS is Python. So if you followed the instructions in [Chapter 1, Introduction to Django and Getting Started](#), and have installed Python, there is nothing else to do-VS takes care of the virtual environment, installing any Python modules you need and even has all of Django's management commands built in to the IDE.

To demonstrate these capabilities, lets create our `mysite` project from [Chapter 1, Introduction to Django and Getting Started](#), but this time we will do it all from inside VS.

Start a Django project

Select `File > New > Project` from the top menu and then select a Python web project from the dropdown on the left. You should see something like *Figure G.6*. Select a Blank Django Web Project, give your project a name and then click OK:



Figure G.6: Create a blank Django project

Visual Studio will then display a popup window saying that this project requires external packages (*Figure G.7*). The simplest option here is to install directly into a virtual environment (option 1), but this will install the latest version of Django, which at the time of writing is 1.9.7. As this book is for the 1.8 LTS version we want to select option 3 **I will install them myself** so we can make the necessary changes to the `requirements.txt` file:



Figure G.7: Install external packages

Once the project has installed, you will notice in Solution

Explorer on the right of the VS screen the complete Django project structure has been created for you. Next step is to add a virtual environment running Django 1.8. At the time of writing the latest version is 1.8.13, so we have to edit our `requirements.txt` file so the first line reads:

```
django==1.8.13
```

Save the file and then right click **Python Environments** in your Solution Explorer and select **Add Virtual Environment...** (*Figure G.8*):

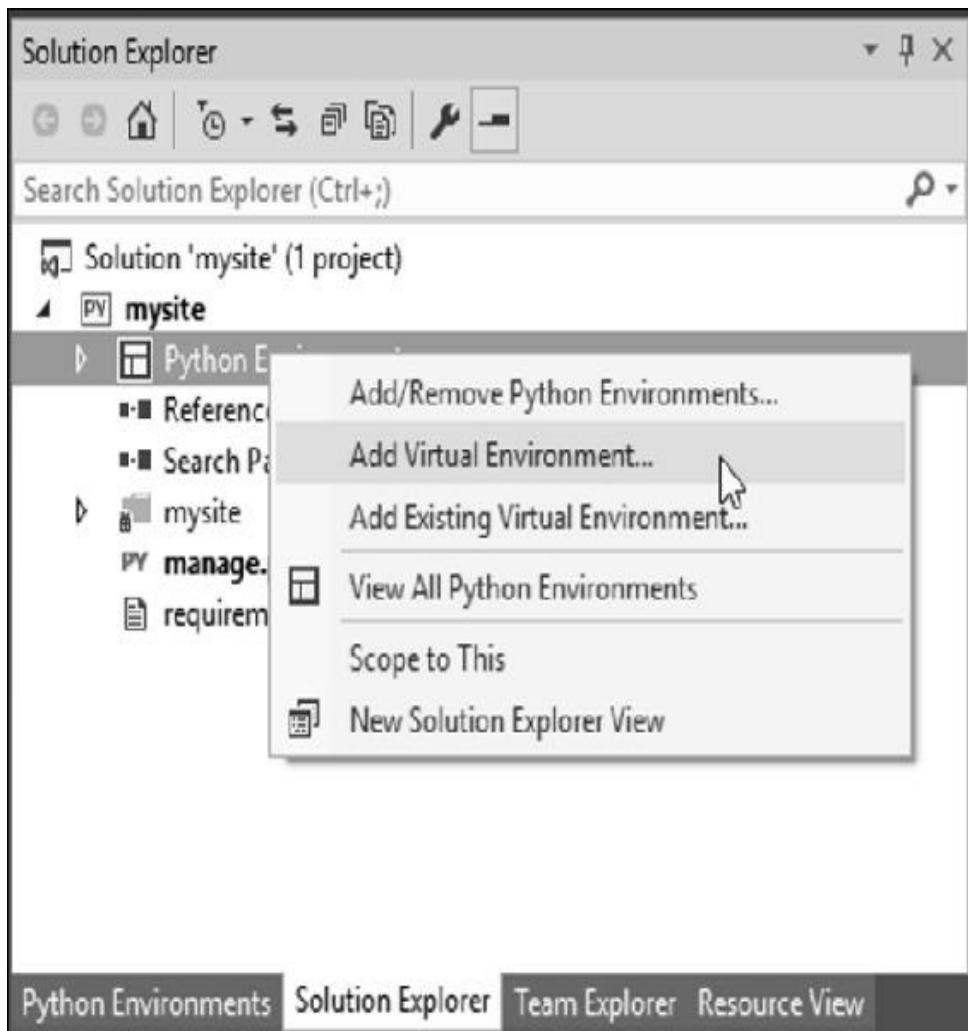


Figure G.8: Add virtual environment

In the popup window, change the default environment name from **env** to something more meaningful (if you are following on from the example in Chapter 1, *Introduction to Django and Getting Started*, use [env_mysite](#)). Click **Create** and VS will create a virtual environment for you (*Figure G.9*):

NOTE

You don't have to explicitly activate a virtual environment when using VS-any code you run

will automatically run in the active virtual environment in Solution Explorer.

This is really useful for cases like testing code against Python 2.7 and 3.4—you just have to right click and activate whichever environment you want to run.

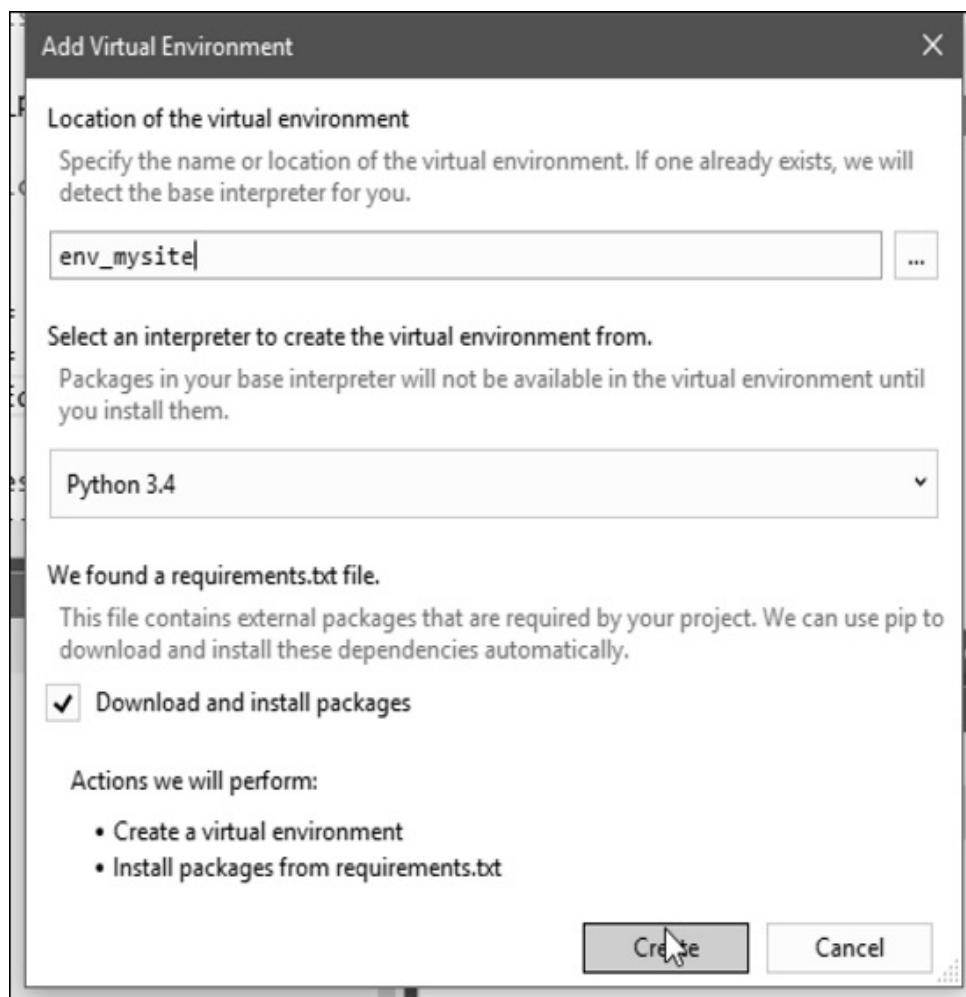


Figure G.9: Create the virtual environment

Django development in Visual Studio

Microsoft have put a lot of effort into ensuring developing Python applications in VS is as simple and headache free as possible. The killer feature for beginning programmers is full IntelliSense for all Python and Django modules. This will accelerate your learning more than any other feature as you don't have to go through documentation looking for module implementations.

The other major aspects of Python/Django programming that VS makes really simple are:

- Integration of Django management commands
- Easy installation of Python packages
- Easy installation of new Django apps

Integration of Django management commands

All of Django's common management commands are available from the Project menu (*Figure G.10*):

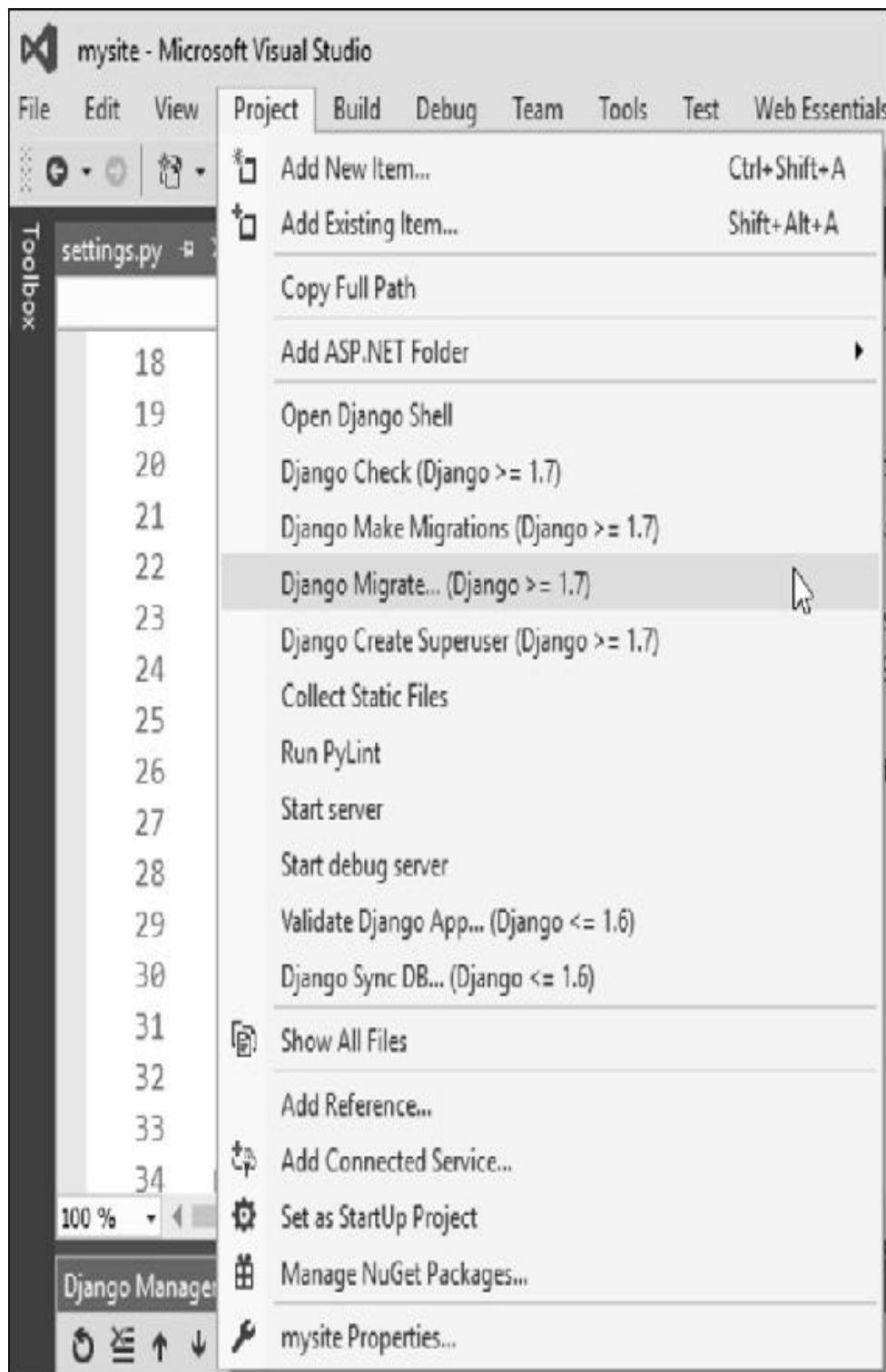


Figure G.10: Common Django commands on Project menu

From this menu you can run migrations, create superusers, open the Django shell and run the development server.

Easy installation of Python packages

Python packages can be installed directly into any virtual environment from Solution Explorer, just right click on the environment and select **Install Python Package...** (*Figure G.11*).

Packages can be installed with either [pip](#) or [easy_install](#).

Easy installation of new Django apps

And finally, adding a new Django app to your project is as simple as right clicking on your project and selecting [Add > Django app...](#) (*Figure G.12*). Give your app a name and click **OK** and VS will add a new app to your project:

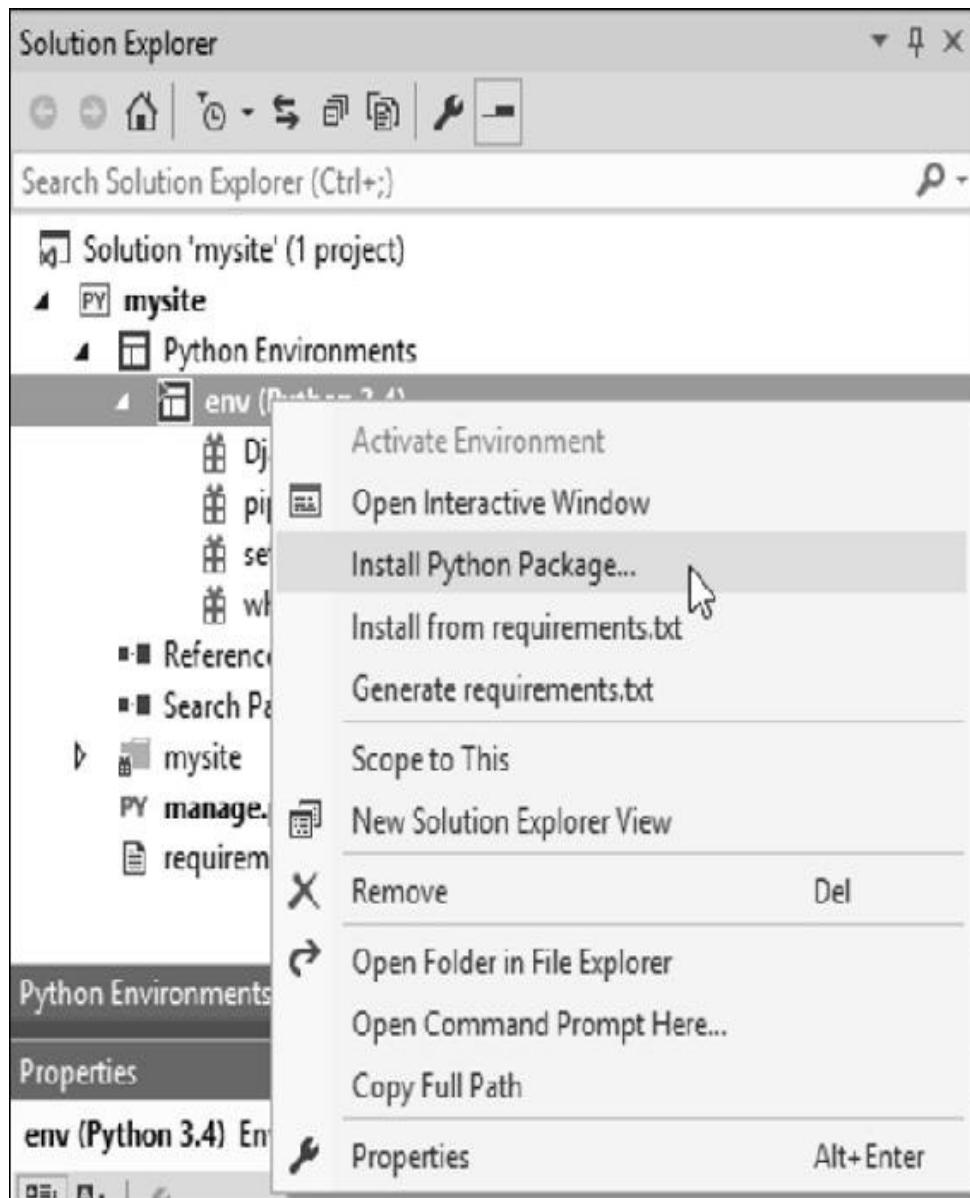


Figure G.11: Install Python package

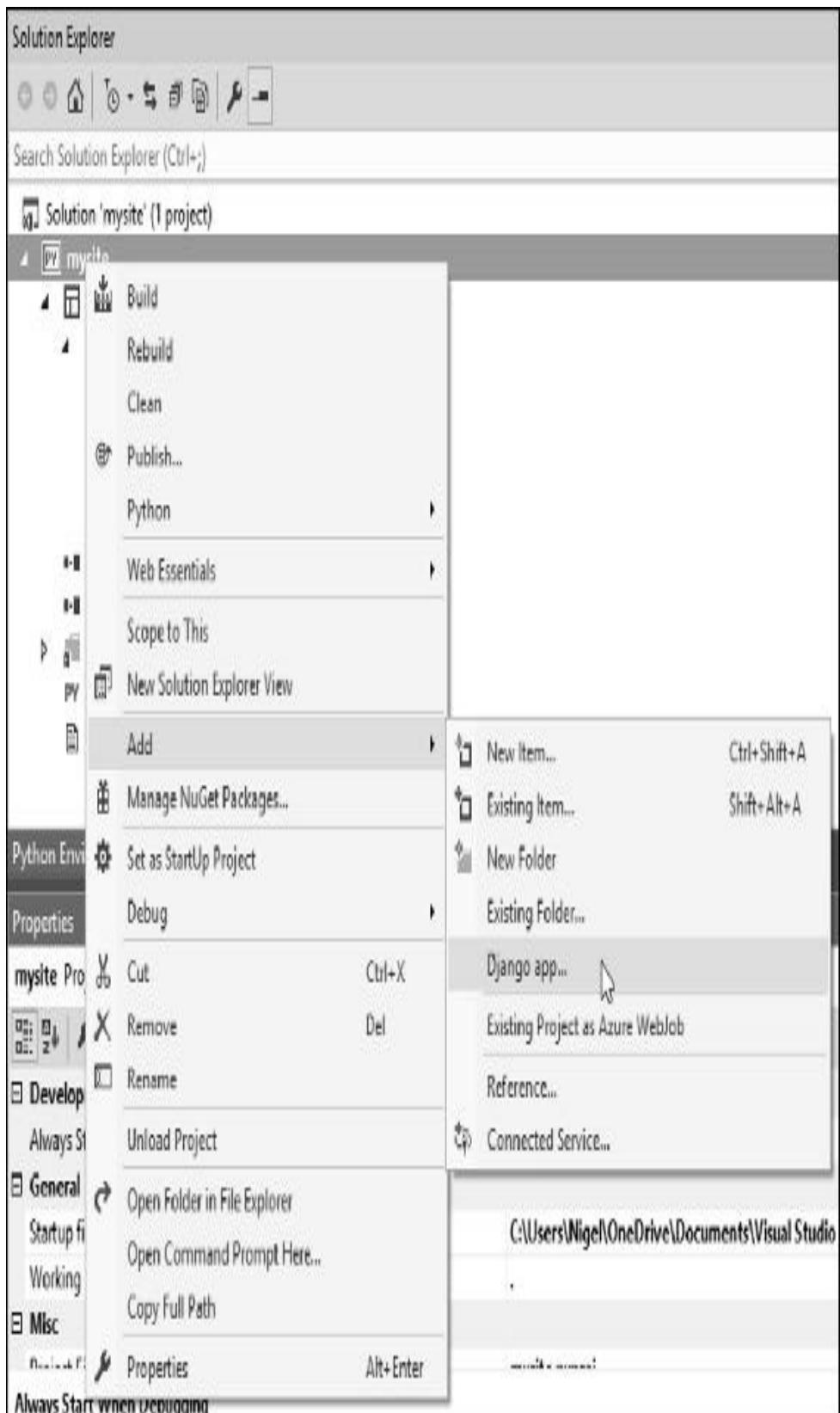


Figure G.12: Add a Django app

This is only a quick overview of the things you can do with Visual Studio; just to get you started. Other things worth exploring are:

- VS's repository management including full integration with local Git repos and GitHub.
- Deployment to Azure with a free MSDN developer account (only supports MySQL and SQLite and the time of writing).
- Inbuilt mixed-mode debugger. For example, debug Django and JavaScript in the same debugger.
- Inbuilt support for testing.
- Did I mention full IntelliSense support?