



TWO SCOOPS *of* django

BEST PRACTICES
for DJANGO 1.5

BY DANIEL GREENFELD
AND AUDREY ROY

Two Scoops of Django

Best Practices For Django 1.5

Daniel Greenfeld

Audrey Roy

Two Scoops of Django: Best Practices for Django 1.5
First Edition, Final Version, 20130411
by Daniel Greenfeld and Audrey Roy

Copyright © 2013 Daniel Greenfeld, Audrey Roy, and Cartwheel Web.

All rights reserved. This book may not be reproduced in any form, in whole or in part, without written permission from the authors, except in the case of brief quotations embodied in articles or reviews.

Limit of Liability and Disclaimer of Warranty: The authors have used their best efforts in preparing this book, and the information provided herein “as is.” The information provided is sold without warranty, either express or implied. Neither the authors nor Cartwheel Web will be held liable for any damages to be caused either directly or indirectly by the contents of this book.

Trademarks: Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.

First printing, January 2013

For more information, visit <https://django.2scoops.org>.

For Malcolm Tredinnick
1971-2013
We miss you.

Contents

List of Figures	xv
List of Tables	xxii
Authors' Notes	xix
A Few Words From Daniel Greenfeld	xix
A Few Words From Audrey Roy	xx
Introduction	xxi
A Word About Our Recommendations	xxi
Why Two Scoops of Django?	xxii
Before You Begin	xxiii
This book is intended for Django 1.5 and Python 2.7.x	xxiii
Each Chapter Stands On Its Own	xxiii
Conventions Used in This Book	xxiv
Core Concepts	xxv
Keep It Simple, Stupid	xxv
Fat Models, Helper Modules, Thin Views, Stupid Templates	xxvi
Start With Django By Default	xxvi
Stand on the Shoulders of Giants	xxvi
1 Coding Style	1
1.1 The Importance of Making Your Code Readable	1
1.2 PEP 8	2
1.3 The Word on Imports	2
1.4 Use Explicit Relative Imports	3
1.5 Avoid Using Import *	6
1.6 Django Coding Style Guidelines	7

1.7	Never Code to the IDE (or Text Editor)	8
1.8	Summary	8
2	The Optimal Django Environment Setup	9
2.1	Use the Same Database Engine Everywhere	9
2.1.1	Fixtures Are Not a Magic Solution	9
2.1.2	You Can't Examine an Exact Copy of Production Data Locally	10
2.1.3	Different Databases Have Different Field Types/Constraints	10
2.2	Use Pip and Virtualenv	11
2.3	Install Django and Other Dependencies via Pip	13
2.4	Use a Version Control System	14
2.5	Summary	14
3	How to Lay Out Django Projects	15
3.1	Django 1.5's Default Project Layout	15
3.2	Our Preferred Project Layout	16
3.2.1	Top Level: Repository Root	16
3.2.2	Second Level: Django Project Root	16
3.2.3	Third Level: Configuration Root	17
3.3	Sample Project Layout	17
3.4	What About the Virtualenv?	20
3.5	Using a Startproject Template to Generate Our Layout	21
3.6	Other Alternatives	22
3.7	Summary	22
4	Fundamentals of Django App Design	23
4.1	The Golden Rule of Django App Design	23
4.1.1	A Practical Example of Apps in a Project	24
4.2	What to Name Your Django Apps	25
4.3	When in Doubt, Keep Apps Small	26
4.4	Summary	26
5	Settings and Requirements Files	27
5.1	Avoid Non-Versioned Local Settings	28
5.2	Using Multiple Settings Files	29
5.2.1	A Development Settings Example	32
5.2.2	Multiple Development Settings	33

5.3	Keep Secret Keys Out With Environment Variables	34
5.3.1	A Caution Before Using Environment Variables for Secrets	35
5.3.2	How to Set Environment Variables Locally	35
5.4	How to Set Environment Variables in Production	37
5.4.1	Handling Missing Secret Key Exceptions	38
5.5	Using Multiple Requirements Files	40
5.5.1	Installing From Multiple Requirements Files	41
5.5.2	Using Multiple Requirements Files With Platforms as a Service (PaaS) . .	42
5.6	Handling File Paths in Settings	42
5.7	Summary	45
6	Database/Model Best Practices	47
6.1	Basics	48
6.1.1	Break Up Apps With Too Many Models	48
6.1.2	Don't Drop Down to Raw SQL Until It's Necessary	48
6.1.3	Add Indexes as Needed	49
6.1.4	Be Careful With Model Inheritance	49
6.1.5	Model Inheritance in Practice: The TimeStampedModel	51
6.1.6	Use South for Migrations	53
6.2	Django Model Design	53
6.2.1	Start Normalized	54
6.2.2	Cache Before Denormalizing	54
6.2.3	Denormalize Only if Absolutely Needed	54
6.2.4	When to Use Null and Blank	55
6.3	Model Managers	56
6.4	Summary	58
7	Function- and Class-Based Views	61
7.1	When to Use FBVs or CBVs	61
7.2	Keep View Logic Out of URLConfs	63
7.3	Stick to Loose Coupling in URLConfs	64
7.3.1	What if we aren't using CBVs?	66
7.4	Try to Keep Business Logic Out of Views	66
7.5	Summary	66
8	Best Practices for Class-Based Views	69

8.1	Using Mixins With CBVs	70
8.2	Which Django CBV Should Be Used for What Task?	72
8.3	General Tips for Django CBVs	73
8.3.1	Constraining Django CBV Access to Authenticated Users	73
8.3.2	Performing Custom Actions on Views With Valid Forms	74
8.3.3	Performing Custom Actions on Views With Invalid Forms	75
8.4	How CBVs and Forms Fit Together	76
8.4.1	Views + ModelForm Example	77
8.4.2	Views + Form Example	81
8.5	Summary	83
9	Common Patterns for Forms	85
9.1	The Power of Django Forms	85
9.2	Pattern 1: Simple ModelForm With Default Validators	86
9.3	Pattern 2: Custom Form Field Validators in ModelForms	87
9.4	Pattern 3: Overriding the Clean Stage of Validation	91
9.5	Pattern 4: Hacking Form Fields (2 CBVs, 2 Forms, 1 Model)	94
9.6	Pattern 5: Reusable Search Mixin View	98
9.7	Summary	100
10	More Things to Know About Forms	101
10.1	Use the POST Method in HTML Forms	101
10.1.1	Don't Disable Django's CSRF Protection	102
10.2	Know How Form Validation Works	102
10.2.1	Form Data Is Saved to the Form, Then the Model Instance	103
10.3	Summary	104
11	Building REST APIs in Django	105
11.1	Fundamentals of Basic REST API Design	105
11.2	Implementing a Simple JSON API	107
11.3	REST API Architecture	109
11.3.1	Code for an App Should Remain in the App	110
11.3.2	Try to Keep Business Logic Out of API Views	110
11.3.3	Grouping API URLs	110
11.3.4	Test Your API	112
11.4	AJAX and the CSRF Token	112

11.4.1	Posting Data via AJAX	112
11.5	Additional Reading	113
11.6	Summary	114
12	Templates: Best Practices	115
12.1	Follow a Minimalist Approach	115
12.2	Template Architecture Patterns	116
12.2.1	2-Tier Template Architecture Example	116
12.2.2	3-Tier Template Architecture Example	116
12.2.3	Flat Is Better Than Nested	117
12.3	Limit Processing in Templates	118
12.3.1	Gotcha 1: Aggregation in Templates	120
12.3.2	Gotcha 2: Filtering With Conditionals in Templates	122
12.3.3	Gotcha 3: Complex Implied Queries in Templates	124
12.3.4	Gotcha 4: Hidden CPU Load in Templates	125
12.3.5	Gotcha 5: Hidden REST API Calls in Templates	126
12.4	Don't Bother Making Your Generated HTML Pretty	126
12.5	Exploring Template Inheritance	127
12.6	block.super Gives the Power of Control	131
12.7	Useful Things to Consider	132
12.7.1	Avoid Coupling Styles Too Tightly to Python Code	132
12.7.2	Common Conventions	133
12.7.3	Location, Location, Location!	133
12.7.4	Use Named Context Objects	133
12.7.5	Use URL Names Instead of Hardcoded Paths	134
12.7.6	Debugging Complex Templates	135
12.7.7	Don't Replace the Django Template Engine	135
12.8	Summary	135
13	Template Tags and Filters	137
13.1	Filters Are Functions	137
13.1.1	Filters Are Easy to Test	138
13.1.2	Filters, Code Reuse, and Performance	138
13.1.3	When to Write Filters	138
13.2	Custom Template Tags	138
13.2.1	Template Tags Are Harder To Debug	139

13.2.2	Template Tags Make Code Reuse Harder	139
13.2.3	The Performance Cost of Template Tags	139
13.2.4	When to Write Template Tags	139
13.3	Naming Your Template Tag Libraries	140
13.4	Loading Your Template Tag Modules	141
13.4.1	Watch Out for This Crazy Anti-Pattern	141
13.5	Summary	142
14	Tradeoffs of Replacing Core Components	143
14.1	The Temptation to Build FrankenDjango	144
14.2	Case Study: Replacing the Django Template Engine	144
14.2.1	Excuses, Excuses	144
14.2.2	What if I'm Hitting the Limits of Templates?	145
14.2.3	What About My Unusual Use Case?	145
14.3	Summary	146
15	Working With the Django Admin	147
15.1	It's Not for End Users	147
15.2	Admin Customization vs. New Views	147
15.3	Viewing String Representations of Objects	148
15.4	Adding Callables to ModelAdmin Classes	151
15.5	Django's Admin Documentation Generator	152
15.6	Securing the Django Admin and Django Admin Docs	153
15.7	Summary	153
16	Dealing With the User Model	155
16.1	Use Django's Tools for Finding the User Model	155
16.1.1	Use settings.AUTH_USER_MODEL for Foreign Keys to User	156
16.2	Custom User Fields for Projects Starting at Django 1.5	156
16.2.1	Option 1: Linking Back From a Related Model	157
16.2.2	Option 2: Subclass AbstractUser	158
16.2.3	Option 3: Subclass AbstractBaseUser	159
16.3	Summary	165
17	Django's Secret Sauce: Third-Party Packages	167
17.1	Examples of Third-Party Packages	167
17.2	Know About the Python Package Index	168

17.3	Know About DjangoPackages.com	168
17.4	Know Your Resources	168
17.5	Tools for Installing and Managing Packages	169
17.6	Package Requirements	169
17.7	Wiring Up Django Packages: The Basics	169
17.7.1	Step 1: Read the Documentation for the Package	169
17.7.2	Step 2: Add Package and Version Number to Your Requirements	170
17.7.3	Step 3: Install the Requirements Into Your Virtualenv	171
17.7.4	Step 4: Follow the Package's Installation Instructions Exactly	171
17.8	Troubleshooting Third-Party Packages	171
17.9	Releasing Your Own Django Packages	172
17.10	What Makes a Good Django Package?	172
17.10.1	Purpose	173
17.10.2	Scope	173
17.10.3	Documentation	173
17.10.4	Tests	173
17.10.5	Activity	174
17.10.6	Community	174
17.10.7	Modularity	174
17.10.8	Availability on PyPI	174
17.10.9	Proper Version Numbers	175
17.10.10	License	175
17.10.11	Clarity of Code	176
17.11	Summary	176
18	Testing Stinks and Is a Waste of Money!	179
18.1	Testing Saves Money, Jobs, and Lives	179
18.2	How to Structure Tests	180
18.3	How to Write Unit Tests	181
18.3.1	Each Test Method Tests One Thing	181
18.3.2	Don't Write Tests That Have to Be Tested	184
18.3.3	Don't Rely on Fixtures	185
18.3.4	Things That Should Be Tested	185
18.4	Continuous Integration	186
18.4.1	Resources for Continuous Integration	187
18.5	Who Cares? We Don't Have Time for Tests!	187

18.6	The Game of Test Coverage	188
18.7	Setting Up the Test Coverage Game	188
18.7.1	Step 1: Set Up a Test Runner	188
18.7.2	Step 2: Run Tests and Generate Coverage Report	189
18.7.3	Step 3: Generate the report!	190
18.8	Playing the Game of Test Coverage	191
18.9	Summary	191
19	Documentation: Be Obsessed	193
19.1	Use reStructuredText for Python Docs	193
19.1.1	Use Sphinx to Generate Documentation From reStructuredText	195
19.2	What Docs Should Django Projects Contain?	196
19.3	Wikis and Other Documentation Methods	197
19.4	Summary	197
20	Finding and Reducing Bottlenecks	199
20.1	Should You Even Care?	199
20.2	Speed Up Query-Heavy Pages	199
20.2.1	Find Excessive Queries With Django Debug Toolbar	199
20.2.2	Reduce the Number of Queries	200
20.2.3	Speed Up Common Queries	201
20.3	Get the Most Out of Your Database	202
20.3.1	Know What Doesn't Belong in the Database	202
20.3.2	Getting the Most Out of PostgreSQL	203
20.3.3	Getting the Most Out of MySQL	203
20.4	Cache Queries With Memcached or Redis	203
20.5	Identify Specific Places to Cache	204
20.6	Consider Third-Party Caching Packages	204
20.7	Compression and Minification of HTML, CSS, and JavaScript	205
20.8	Use Upstream Caching or a Content Delivery Network	206
20.9	Other Resources	206
20.10	Summary	207
21	Security Best Practices	209
21.1	Harden Your Servers	209
21.2	Know Django's Security Features	209

21.3	Turn Off DEBUG Mode in Production	210
21.4	Keep Your Secret Keys Secret	210
21.5	HTTPS Everywhere	210
21.5.1	Use Secure Cookies	211
21.5.2	Use HTTP Strict Transport Security (HSTS)	212
21.6	Use Django 1.5's Allowed Hosts Validation	213
21.7	Always Use CSRF Protection With HTTP Forms That Modify Data	213
21.7.1	Posting Data via AJAX	214
21.8	Prevent Against Cross-Site Scripting (XSS) Attacks	214
21.9	Defend Against Python Code Injection Attacks	215
21.9.1	Python Built-ins That Execute Code	215
21.9.2	Python Standard Library Modules That Can Execute Code	215
21.9.3	Third-Party Libraries That Can Execute Code	215
21.10	Validate All User Input With Django Forms	216
21.11	Handle User-Uploaded Files Carefully	218
21.12	Don't Use ModelForms.Meta.exclude	219
21.13	Beware of SQL Injection Attacks	222
21.14	Never Store Credit Card Data	222
21.15	Secure the Django Admin	222
21.15.1	Change the Default Admin URL	223
21.15.2	Use django-admin-honeypot	223
21.15.3	Only Allow Admin Access via HTTPS	223
21.15.4	Limit Admin Access Based on IP	224
21.15.5	Use the allow_tags Attribute With Caution	224
21.16	Secure the Admin Docs	224
21.17	Monitor Your Sites	224
21.18	Keep Your Dependencies Up-to-Date	225
21.19	Put Up a Vulnerability Reporting Page	225
21.20	Keep Up-to-Date on General Security Practices	225
21.21	Summary	226
22	Logging: What's It For, Anyway?	227
22.1	Application Logs vs. Other Logs	227
22.2	Why Bother With Logging?	228
22.3	When to Use Each Log Level	228
22.3.1	Log Catastrophes With CRITICAL	229

22.3.2	Log Production Errors With ERROR	229
22.3.3	Log Lower-Priority Problems With WARNING	230
22.3.4	Log Useful State Information With INFO	231
22.3.5	Log Debug-Related Messages to DEBUG	231
22.4	Log Tracebacks When Catching Exceptions	233
22.5	One Logger Per Module That Uses Logging	234
22.6	Log Locally to Rotating Files	234
22.7	Other Logging Tips	235
22.8	Necessary Reading Material	236
22.9	Useful Third-Party Tools	236
22.10	Summary	236
23	Signals: Use Cases and Avoidance Techniques	237
23.1	When to Use and Avoid Signals	237
23.2	Signal Avoidance Techniques	238
23.2.1	Using Custom Model Manager Methods Instead of Signals	238
23.2.2	Validate Your Model Elsewhere	241
23.2.3	Override Your Model's Save or Delete Method Instead	241
24	What About Those Random Utilities?	243
24.1	Create a Core App for Your Utilities	243
24.2	Django's Own Swiss Army Knife	244
24.2.1	django.contrib.humanize	245
24.2.2	django.utils.html.remove_tags(value, tags)	245
24.2.3	django.utils.html.strip_tags(value)	245
24.2.4	django.utils.text.slugify(value)	245
24.2.5	django.utils.timezone	246
24.2.6	django.utils.translation	246
24.3	Summary	247
25	Deploying Django Projects	249
25.1	Using Your Own Web Servers	249
25.2	Using a Platform as a Service	250
25.3	Summary	252
26	Where and How to Ask Django Questions	253
26.1	What to Do When You're Stuck	253

26.2	How to Ask Great Django Questions in IRC	253
26.3	Insider Tip: Be Active in the Community	254
26.3.1	10 Easy Ways to Participate	254
26.4	Summary	255
27	Closing Thoughts	257
	Appendix A: Packages Mentioned In This Book	259
	Appendix B: Troubleshooting	263
	Identifying the Issue	263
	Our Recommended Solutions	263
	Check Your Virtualenv Installation	264
	Check If Your Virtualenv Has Django 1.5 Installed	265
	Check For Other Problems	265
	Appendix C: Additional Resources	267
	Acknowledgments	269
	Index	275

List of Figures

1	Throwing caution to the wind.	xxii
1.1	Using <code>import *</code> in an ice cream shop.	7
2.1	Pip, virtualenv, and virtualenvwrapper in ice cream bar form.	13
3.1	Three-tiered scoop layout.	17
3.2	Project layout differences of opinion can cause ice cream fights.	22
4.1	Our vision for Icecreamlandia.	25
5.1	Over 130 settings are available to you in Django 1.5.	27
6.1	A common source of confusion.	56
7.1	Should you use a FBV or a CBV? flow chart.	62
8.1	Popular and unpopular mixins used in ice cream.	70
8.2	The other CBV: class-based vanilla ice cream.	76
8.3	Views + ModelForm Flow	78
8.4	Views + Form Flow	82
12.1	An excerpt from the Zen of Ice Cream.	117
12.2	Bubble gum ice cream looks easy to eat but requires a lot of processing.	125
15.1	Admin list page for an ice cream bar app.	148
15.2	Improved admin list page with better string representation of our objects.	149
15.3	Improved admin list page with better string representation of our objects.	150
15.4	Displaying URL in the Django Admin.	152
16.1	This looks strange too.	156

18.1	Test as much of your project as you can, as if it were free ice cream.	186
19.1	Even ice cream could benefit from documentation.	197
22.1	CRITICAL/ERROR/WARNING/INFO logging in ice cream	228
22.2	Appropriate usage of DEBUG logging in ice cream.	233
25.1	How ice cream is deployed to cones and bowls.	252

List of Tables

Author's Ice Cream Preferences	xxv
1.1 Imports: Absolute vs. Explicit Relative vs. Implicit Relative	5
3.1 Repository Root Files and Directories	19
3.2 Django Project Files and Directories	20
5.1 Settings files and their purpose	30
5.2 Setting DJANGO_SETTINGS_MODULE per location	31
6.1 Pros and Cons of the Model Inheritance Styles	50
6.2 When To use Null and Blank by Field	56
8.1 Django CBV Usage Table	72
11.1 HTTP Methods	106
11.2 HTTP Status Codes	107
11.3 URLConf for the Flavor REST APIs	109
12.1 Template Tags in base.html	129
12.2 Template Objects in about.html	130
14.1 Fad-based Reasons to Replace Components of Django	144
19.1 Documentation Django Projects Should Contain	196
25.1 Gunicorn vs Apache	250

Authors' Notes

A Few Words From Daniel Greenfeld

In the spring of 2006, I was working for NASA on a project that implemented a Java-based RESTful web service that was taking weeks to deliver. One evening, when management had left for the day, I reimplemented the service in Python in 90 minutes.

I knew then that I wanted to work with Python.

I wanted to use Django for the web front-end of the web service, but management insisted on using a closed-source stack because “Django is only at version 0.9x, hence not ready for real projects.” I disagreed, but stayed happy with the realization that at least the core architecture was in Python. Django used to be edgy during those heady days, and it scared people the same way that Node.js scares people today.

Nearly seven years later, Django is considered a mature, powerful, secure, stable framework used by incredibly successful corporations (Instagram, Pinterest, Mozilla, etc.) and government agencies (NASA, et al) all over the world. Convincing management to use Django isn’t hard anymore, and if it is hard to convince them, finding jobs which let you use Django has become much easier.

In my 6+ years of building Django projects, I’ve learned how to launch new web applications with incredible speed while keeping technical debt to an absolute minimum.

My goal in this book is to share with you what I’ve learned. My knowledge and experience have been gathered from advice given by core developers, mistakes I’ve made, successes shared with others, and an enormous amount of note taking. I’m going to admit that the book is opinionated, but many of the leaders in the Django community use the same or similar techniques.

This book is for you, the developers. I hope you enjoy it!

A Few Words From Audrey Roy

I first discovered Python in a graduate class at MIT in 2005. In less than 4 weeks of homework assignments, each student built a voice-controlled system for navigating between rooms in MIT's Stata Center, running on our HP iPaks running Debian. I was in awe of Python and wondered why it wasn't used for everything. I tried building a web application with Zope but struggled with it.

A couple of years passed, and I got drawn into the Silicon Valley tech startup scene. I wrote graphics libraries in C and desktop applications in C++ for a startup. At some point, I left that job and picked up painting and sculpture. Soon I was drawing and painting frantically for art shows, co-directing a 140-person art show, and managing a series of real estate renovations. I realized that I was doing a lot at once and had to optimize. Naturally, I turned to Python and began writing scripts to generate some of my artwork. That was when I rediscovered the joy of working with Python.

Many friends from the Google App Engine, SuperHappyDevHouse, and hackathon scenes in Silicon Valley inspired me to get into Django. Through them and through various freelance projects and partnerships I discovered how powerful Django was.

Before I knew it, I was attending PyCon 2010, where I met my fiance Daniel Greenfeld. We met at the end of James Bennett's Django In Depth tutorial, and now this chapter in our lives has come full circle with the publication of this book.

Django has brought more joy to my life than I thought was possible with a web framework. My goal with this book is to give you the thoughtful guidance on common Django development practices that are normally left unwritten (or implied), so that you can get past common hurdles and experience the joy of using the Django web framework for your projects.

Introduction

Our aim in writing this book is to write down all of the unwritten tips, tricks, and common practices that we've learned over the years while working with Django.

While writing, we've thought of ourselves as scribes, taking the various things that people assume are common knowledge and recording them with simple examples.

A Word About Our Recommendations

Like the official Django documentation, this book covers how to do things in Django, illustrating various scenarios with code examples.

Unlike the Django documentation, this book recommends particular coding styles, patterns, and library choices. While core Django developers may agree with some or many of these choices, keep in mind that many of our recommendations are just that: personal recommendations formed after years of working with Django.

Throughout this book, we advocate certain practices and techniques that we consider to be the best approaches. We also express our own personal preferences for particular tools and libraries.

Sometimes we reject common practices that we consider to be anti-patterns. For most things we reject, we try to be polite and respectful of the hard work of the authors. There are the rare, few things that we may not be so polite about. This is in the interest of helping you avoid dangerous pitfalls.

We have made every effort to give thoughtful recommendations and to make sure that our practices are sound. We've subjected ourselves to harsh, nerve-racking critiques from Django core developers

whom we greatly respect. We've had this book reviewed by more technical reviewers than the average technical book, and we've poured countless hours into revisions. That being said, there is always the possibility of errors or omissions. There is also the possibility that better practices may emerge than those described here.

We are fully committed to iterating on and improving this book, and we mean it. If you see any practices that you disagree with or anything that can be done better, we humbly ask that you send us your suggestions for improvements.

Please don't hesitate to tell us what can be improved. We will take your feedback constructively. If immediate action is required, we will send out errata or an updated version to readers ASAP at no cost.

Why Two Scoops of Django?

Like most people, we, the authors of this book, love ice cream. Every Saturday night we throw caution to the wind and indulge in ice cream. Don't tell anyone, but sometimes we even have some when it's not Saturday night!

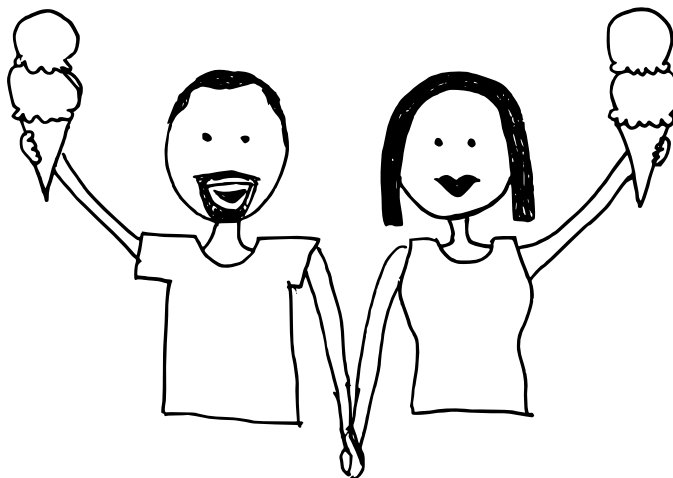


Figure 1: Throwing caution to the wind.

We like to try new flavors and discuss their merits against our old favorites. Tracking our progress through all these flavors, and possibly building a club around it, makes for a great sample Django project.

When we do find a flavor we really like, the new flavor brings a smile to our face, just like when we find great tidbits of code or advice in a technical book. One of our goals for this book is to write the kind of technical book that brings the ice cream smile to readers.

Best of all, using ice cream analogies has allowed us to come up with more vivid code examples. We've had a lot of fun writing this book. You may see us go overboard with ice cream silliness here and there; please forgive us.

Before You Begin

If you are new to Django, this book will be helpful, but large parts will be challenging for you. To use this book to its fullest extent, you should have an understanding of the Python programming language and have at least gone through the 5 page Django tutorial: <https://docs.djangoproject.com/en/1.5/intro/tutorial01/>. Experience with object-oriented programming is also very useful.

This Book Is Intended for Django 1.5 and Python 2.7.x

This book should work well with the Django 1.4 series, less so with Django 1.3, and so on. Even though we make no promises about functional compatibility, at least the general approaches from most of this book stand up over every post-1.0 version of Django.

As for the Python version, this book relies on Python 2.7.x. We hope to release an updated edition once more of the Django community starts moving toward Python 3.3 (or higher).

None of the content in this book, including our practices, the code examples, and the libraries referenced applies to Google App Engine (GAE). If you try to use this book as a reference for GAE development, you may run into problems.

Each Chapter Stands on Its Own

Unlike tutorial and walkthrough books where each chapter builds upon the previous chapter's project, we've written this book in a way that each chapter intentionally stands by itself.

We’ve done this in order to make it easy for you to reference chapters about specific topics when needed while you’re working on a project.

The examples in each chapter are completely independent. They aren’t intended to be combined into one project. Consider them useful, isolated snippets that illustrate and help with various coding scenarios.

Conventions Used in This Book

Code blocks like the following are used throughout the book:

```
EXAMPLE 0.1
class Scoop(object):
    def __init__(self):
        self._is_yummy = True
```

To keep these snippets compact, we sometimes violate the PEP 8 conventions on comments and line spacing.

Special “Don’t Do This!” code blocks like the following indicate examples of bad code that you should avoid:

```
BAD EXAMPLE 0.1
# DON'T DO THIS!
from rotten_ice_cream import something_bad
```

We use the following typographical conventions throughout the book:

- Constant width for code fragments or commands.
- *Italic* for filenames.
- **Bold** when introducing a new term or important word.

Boxes containing notes, warnings, tips, and little anecdotes are also used in this book:

TIP: Something You Should Know

Tip boxes give handy advice.

WARNING: Some Dangerous Pitfall

Warning boxes help you avoid common mistakes and pitfalls.

PACKAGE TIP: Some Useful Django Package Recommendation

Indicates notes about useful third-party packages related to the current chapter, and general notes about using various Django packages.

We also provide a complete list of packages recommended throughout the book in [Appendix A: Packages Mentioned In This Book](#).

We also use tables to summarize information in a handy, concise way:

	Daniel Greenfeld	Audrey Roy
Can be fed coconut ice cream	No	Yes
Favorite ice cream flavors of the moment	Birthday Cupcake, and anything with peanut butter	Extreme Moose Tracks, Chocolate

Authors' Ice Cream Preferences

Core Concepts

When we build Django projects, we keep the following concepts in mind:

Keep It Simple, Stupid

Kelly Johnson, one of the most renowned and prolific aircraft design engineers in the history of aviation, said it this way about 50 years ago. Centuries earlier, Leonardo da Vinci meant the same

thing when he said “Simplicity is the ultimate sophistication.”

When building software projects, each piece of unnecessary complexity makes it harder to add new features and maintain old ones. Attempt the simplest solution, but take care not to implement overly simplistic solutions that make bad assumptions.

Fat Models, Helper Modules, Thin Views, Stupid Templates

When deciding where to put a piece of code, we like to follow the “Fat Models, Helper Modules, Thin Views, Stupid Templates” approach.

We recommend that you err on the side of putting more logic into anything but views and templates. The results are pleasing. The code becomes clearer, more self-documenting, less duplicated, and a lot more reusable.

As for template tags and filters, they should contain the minimum logic possible to function. We cover this further in [chapter 13](#), ‘Template Tags and Filters’.

Start With Django by Default

Before we consider switching out core Django components for things like alternative template engines, different ORMs, or non-relational databases, we first try an implementation using standard Django components. If we run into obstacles, we explore all possibilities before replacing core Django components.

See [chapter 14](#), ‘Tradeoffs of Replacing Core Components’ for more details.

Stand on the Shoulders of Giants

While we take credit and responsibility for our work, we certainly did not come up with the practices described in this book on our own.

Without all of the talented, creative, and generous developers who make up the Django, Python, and general open-source software communities, this book would not exist. We strongly believe in recognizing the people who have served as our teachers and mentors as well as our sources for information, and we've tried our best to give credit whenever credit is due.

1 | Coding Style

A little attention to following standard coding style guidelines will go a long way. We highly recommend that you read this chapter, even though you may be tempted to skip it.

1.1 The Importance of Making Your Code Readable

Code is read more than it is written. An individual block of code takes moments to write, minutes or hours to debug, and can last forever without being touched again. It's when you or someone else visits code written yesterday or ten years ago that having code written in a clear, consistent style becomes extremely useful. Understandable code frees mental bandwidth from having to puzzle out inconsistencies, making it easier to maintain and enhance projects of all sizes.

What this means is that you should go the extra mile to make your code as readable as possible:

- Avoid abbreviating variable names.
- Write out your function argument names.
- Document your classes and methods.
- Refactor repeated lines of code into reusable functions or methods.

When you come back to your code after time away from it, you'll have an easier time picking up where you left off.

Take those pesky abbreviated variable names, for example. When you see a variable called `balance_sheet_decrease`, it's much easier to interpret in your mind than an abbreviated variable like `bsd` or `bal_s_d`. These types of shortcuts may save a few seconds of typing, but that savings comes at the expense of hours or days of technical debt. It's not worth it.

1.2 PEP 8

PEP 8 is the official style guide for Python. We advise reading it in detail and learn to follow the PEP 8 coding conventions: <http://www.python.org/dev/peps/pep-0008/>

PEP 8 describes coding conventions such as:

- “Use 4 spaces per indentation level.”
- “Separate top-level function and class definitions with two blank lines.”
- “Method definitions inside a class are separated by a single blank line.”

All the Python files in your Django projects should follow PEP 8. If you have trouble remembering the PEP 8 guidelines, find a plugin for your code editor that checks your code as you type.

When an experienced Python developer sees gross violations of PEP 8 in a Django project, even if they don't say something mean, they are probably thinking bad things. Trust us on this one.

WARNING: Don't Change an Existing Project's Conventions

The style of PEP 8 applies to new Django projects only. If you are brought into an existing Django project that follows a different convention than PEP 8, then follow the existing conventions.

Please read the “A Foolish Consistency is the Hobgoblin of Little Minds” section of PEP 8 for details about this and other reasons to break the rules:

- <http://2scoops.co/hobgoblin-of-little-minds>

1.3 The Word on Imports

PEP 8 suggests that imports should be grouped in the following order:

- ❶ Standard library imports
- ❷ Related third-party imports
- ❸ Local application or library specific imports

When we're working on a Django project, our imports look something like the following:

EXAMPLE 1.1

```
# Stdlib imports
from math import sqrt
from os.path import abspath

# Core Django imports
from django.db import models
from django.utils.translation import ugettext_lazy as _

# Third-party app imports
from django_extensions.db.models import TimeStampedModel

# Imports from your apps
from splits.models import BananaSplit
```

(Note: you don't actually need to comment your imports like this; the comments are just here to explain the example.)

The import order here is:

- ❶ Standard library imports.
- ❷ Imports from core Django.
- ❸ Imports from third-party apps.
- ❹ Imports from the apps that you created as part of your Django project. (You'll read more about apps in [chapter 4](#), *Fundamentals of App Design*.)

1.4 Use Explicit Relative Imports

When writing code, it's important to do so in such a way that it's easier to move, rename, and version your work. In Python, explicit relative imports remove the need for hardcoding a module's package, separating individual modules from being tightly coupled to the architecture around them. Since Django apps are simply Python packages, the same rules apply.

To illustrate the benefits of explicit relative imports, let's explore an example.

Imagine that the following snippet is from a Django project that you created to track your ice cream consumption, including all of the waffle/sugar/cake cones that you have ever eaten.

Oh no, your cones app contains hardcoded imports, which are bad!

```
BAD EXAMPLE I.1

# cones/views.py
from django.views.generic import CreateView

# DON'T DO THIS!
# Hardcoding of the 'cones' package
# with implicit relative imports
from cones.models import WaffleCone
from cones.forms import WaffleConeForm
from core.views import FoodMixin

class WaffleConeCreateView(FoodMixin, CreateView):
    model = WaffleCone
    form_class = WaffleConeForm
```

Sure, your cones app works fine within your ice cream tracker project, but it has those nasty hardcoded imports that make it less portable and reusable:

- What if you wanted to reuse your cones app in another project that tracks your general dessert consumption, but you had to change the name due to a naming conflict (e.g. a conflict with a Django app for snow cones)?
- What if you simply wanted to change the name of the app at some point?

With hardcoded imports, you can't just change the name of the app; you have to dig through all of the imports and change them as well. It's not hard to change them manually, but before you dismiss the need for explicit relative imports, keep in mind that the above example is extremely simple compared to a real app with various additional helper modules.

Let's now convert the bad code snippet containing hardcoded imports into a good one containing explicit relative imports. Here's the corrected example:

```
EXAMPLE I.2

# cones/views.py
from django.views.generic import CreateView
```

```
# Relative imports of the 'cones' package
from .models import WaffleCone
from .forms import WaffleConeForm
from core.views import FoodMixin

class WaffleConeCreateView(FoodMixin, CreateView):
    model = WaffleCone
    form_class = WaffleConeForm
```

To summarize, here's a table of the different Python import types and when to use them in Django projects:

Code	Import Type	Usage
<code>from core.views import FoodMixin</code>	absolute import	Use when importing from outside the current app
<code>from .models import WaffleCone</code>	explicit relative	Use when importing from another module in the current app
<code>from cones.models import WaffleCone</code>	implicit relative	Often used when importing from another module in the current app, but not a good idea

Table 1.1: Imports: Absolute vs. Explicit Relative vs. Implicit Relative

Get into the habit of using explicit relative imports. It's very easy to do, and using explicit relative imports is a good habit for any Python programmer to develop.

TIP: Doesn't PEP 328 clash with PEP 8?

See what Guido Van Rossum, BDFL of Python says about it:

➤ <http://2scoops.co/guido-on-pep-8-vs-pep-328>

Additional reading: <http://www.python.org/dev/peps/pep-0328/>

1.5 Avoid Using Import *

In 99% of all our work, we explicitly import each module:

EXAMPLE 1.3

```
from django import forms
from django.db import models
```

Never do the following:

BAD EXAMPLE 1.2

```
# ANTI-PATTERN: Don't do this!
from django.forms import *
from django.db.models import *
```

The reason for this is to avoid implicitly loading all of another Python module's locals into and over our current module's namespace, which can produce unpredictable and sometimes catastrophic results.

We do cover a specific exception to this rule in [chapter 5](#), *Settings and Requirements Files*.

For example, both the Django Forms and Django Models libraries have a class called `CharField`. By implicitly loading both libraries, the Models library overwrote the Forms version of the class. This can also happen with Python built-in libraries and other third-party libraries overwriting critical functionality.

WARNING: Python Naming Collisions

You'll run into similar problems if you try to import two things with the same name, such as:

BAD EXAMPLE 1.3

```
# ANTI-PATTERN: Don't do this!
from django.forms import CharField
from django.db.models import CharField
```

Using `import *` is like being that greedy customer at an ice cream shop who asks for a free taster spoon of all thirty-one flavors, but who only purchases one or two scoops. Don't import everything if you're only going to use one or two things.

If the customer then walked out with a giant ice cream bowl containing a scoop of every or almost every flavor, though, it would be a different matter.

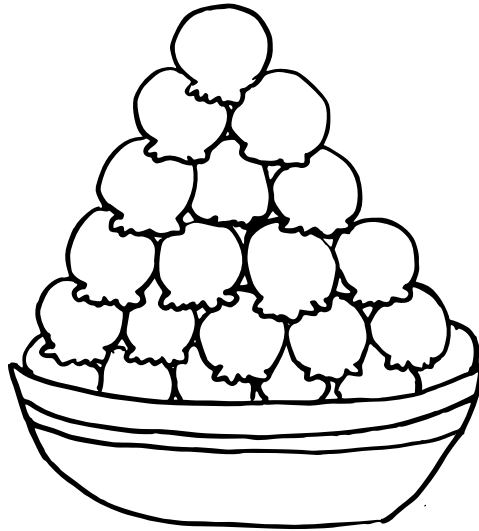


Figure 1.1: Using `import *` in an ice cream shop.

1.6 Django Coding Style Guidelines

It goes without saying that it's a good idea to be aware of common Django style conventions. In fact, internally Django has its own set of style guidelines that extend PEP 8:

- <http://2scoops.co/1.5-coding-style>

While the following are not specified in the official standards, you may want to follow them in your projects:

- Use underscores (the `'_'` character) in URL pattern names rather than dashes as this is friendlier to more IDEs and text editors. Note that we are referring to the name argument of `url()` here, not the actual URL typed into the browser. Dashes in actual URLs are fine.
- For the same reason, use underscores rather than dashes in template block names.

1.7 Never Code to the IDE (or Text Editor)

There are developers who make decisions about the layout and implementation of their project based on the features of IDEs. This can make discovery of project code extremely difficult for anyone whose choice of development tool doesn't match the original author.

Another way of saying “*Never code to the IDE*” could also be “*Coding by Convention*”. Always assume that the developers around you like to use their own tools and that your code and project layout should be transparent enough that someone stuck using NotePad or Nano will be able to navigate your work.

For example, introspecting **template tags** or discovering their source can be difficult and time consuming for developers not using a very, very limited pool of IDEs. Therefore, we follow the commonly used naming pattern of `<app_name>_tags.py`.

1.8 Summary

This chapter covered our preferred coding style and explained why we prefer each technique.

Even if you don't follow the coding style that we use, please follow a consistent coding style. Projects with varying styles are much harder to maintain, slowing development and increasing the chances of developer mistakes.

2 | The Optimal Django Environment Setup

This chapter describes what we consider the best local environment setup for intermediate and advanced developers working with Django.

2.1 Use the Same Database Engine Everywhere

A common developer pitfall is using **SQLite3** for local development and **PostgreSQL** (or another database besides SQLite3) in production. This section applies not only to the SQLite3/PostgreSQL scenario, but to any scenario where you're using two different databases and expecting them to behave identically.

Here are some of the issues we've encountered with using different database engines for development and production:

2.1.1 Fixtures Are Not a Magic Solution

You may be wondering why you can't simply use **fixtures** to abstract away the differences between your local and production databases.

Well, fixtures are great for creating simple hardcoded test data sets. Sometimes you need to pre-populate your databases with fake test data during development, particularly during the early stages of a project.

Fixtures are not a reliable tool for migrating large data sets from one database to another in a database-agnostic way, and they are not meant to be used that way. Don't mistake the ability of fixtures to create basic data (dumpdata/loaddata) with the capability to migrate production data between database tools.

2.1.2 You Can't Examine an Exact Copy of Production Data Locally

When your production database is different from your local development database, you can't grab an exact copy of your production database to examine data locally.

Sure, you can generate a SQL dump from production and import it into your local database, but that doesn't mean that you have an exact copy after the export and import.

2.1.3 Different Databases Have Different Field Types/Constraints

Keep in mind that different databases handle typing of field data differently. Django's ORM attempts to accommodate those differences, but there's only so much that it can do.

For example, some people use SQLite3 for local development and PostgreSQL in production, thinking that the Django ORM gives them the excuse not to think about the differences. Eventually they run into problems, since SQLite3 has dynamic, weak typing instead of strong typing.

Yes, the Django ORM has features that allow your code to interact with SQLite3 in a more strongly typed manner, but form and model validation mistakes in development will go uncaught (even in tests) until the code goes to a production server. You may be saving long strings locally without a hitch, for example, since SQLite3 won't care. But then in production, your PostgreSQL or MySQL database will throw constraint errors that you've never seen locally, and you'll have a hard time replicating the issues until you set up an identical database locally.

Most problems usually can't be discovered until the project is run on a strongly typed database (e.g. PostgreSQL or MySQL). When these types of bugs hit, you end up kicking yourself and scrambling to set up your local development machine with the right database.

TIP: Django+PostgreSQL Rocks

Most Django developers that we know prefer to use PostgreSQL for all environments: development, staging, QA, and production systems.

Depending on your operating system, use these instructions:

- Mac: Download the one-click Mac installer at <http://postgresapp.com>
- Windows: Download the one-click Windows installer at <http://postgresql.org/download/windows/>
- Linux: Install via your package manager, or follow the instructions at <http://postgresql.org/download/linux/>

PostgreSQL may take some work to get running locally on some operating systems, but we find that it's well worth the effort.



2.2 Use Pip and Virtualenv

If you are not doing so already, we strongly urge you to familiarize yourself with both pip and virtualenv. They are the de facto standard for Django projects, and most companies that use Django rely on these tools.

Pip is a tool that fetches Python packages from the Python Package Index and its mirrors. It is used to manage and install Python packages. It's like `easy_install` but has more features, the key feature being support for virtualenv.

Virtualenv is a tool for creating isolated Python environments for maintaining package dependencies. It's great for situations where you're working on more than one project at a time, and where there are clashes between the version numbers of different libraries that your projects use.

For example, imagine that you're working on one project that requires Django 1.4 and another that requires Django 1.5.

- Without virtualenv (or an alternative tool to manage dependencies), you have to reinstall Django every time you switch projects.
- If that sounds tedious, keep in mind that most real Django projects have at least a dozen dependencies to maintain.

Further reading and installation instructions can be found at:

- pip: <http://pip-installer.org>
- virtualenv: <http://virtualenv.org>

TIP: virtualenvwrapper

For developers using Mac OS X or Linux, or those with advanced Windows skills and ample patience, we also highly recommend **virtualenvwrapper** by Doug Hellmann: <http://virtualenvwrapper.readthedocs.org>

Personally, we think virtualenv without virtualenvwrapper can be a pain to use, because every time you want to activate a virtual environment, you have to type something long like:

EXAMPLE 2.1

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

With virtualenvwrapper, you'd only have to type:

EXAMPLE 2.2

```
$ workon twoscoops
```

Virtualenvwrapper is a popular companion tool to pip and virtualenv and makes our lives easier, but it's not an absolute necessity.

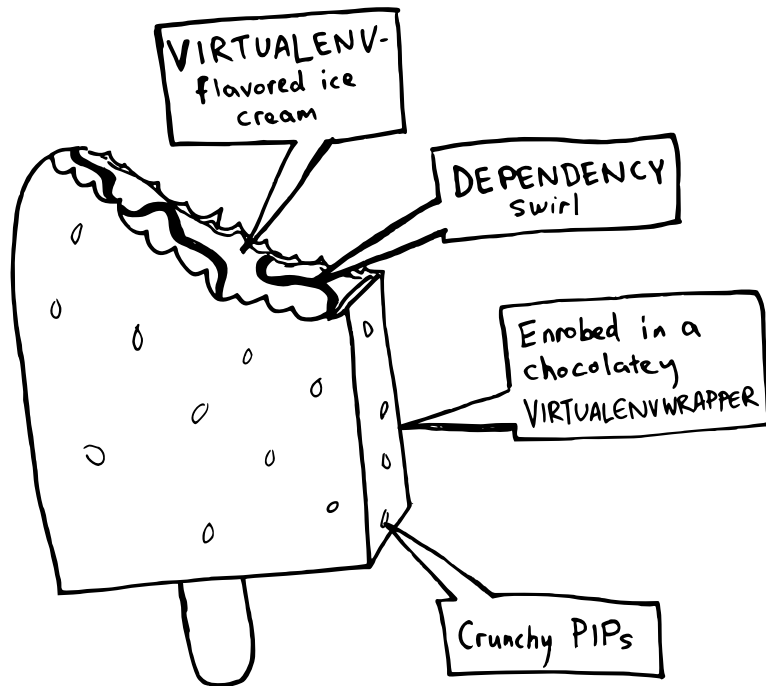


Figure 2.1: Pip, virtualenv, and virtualenvwrapper in ice cream bar form.

2.3 Install Django and Other Dependencies via Pip

The official Django documentation describes several ways of installing Django. Our recommended installation method is with pip and requirements files.

To summarize how this works: a requirements file is like a grocery list of Python packages that you want to install. It contains the name and desired version of each package. You use pip to install packages from this list into your virtual environment.

We cover the setup of and installation from requirements files in [chapter 5](#), 'Settings and Requirements Files'.

TIP: Setting PYTHONPATH

If you have a firm grasp of the command line and environment variables, you can set your `virtualenv` `PYTHONPATH` so that the `django-admin.py` command can be used to serve your site and perform other tasks. If you don't know how to set this, don't worry about it and stick with `manage.py`. Additional reading:

- <http://cs.simons-rock.edu/python/pythonpath.html>
- <https://docs.djangoproject.com/en/1.5/ref/django-admin/>

2.4 Use a Version Control System

Version control systems are also known as revision control or source control. Whenever you work on any Django project, you should use a version control system to keep track of your code changes.

Wikipedia has a detailed comparison of different version control systems:

- http://en.wikipedia.org/wiki/Comparison_of_revision_control_software

Of all the options, **Git** and **Mercurial** seem to be the most popular among Django developers. Both Git and Mercurial make it easy to create branches and merge changes.

When using a version control system, it's important to not only have a local copy of your code repository, but also to use a code hosting service for backups. For this, we recommend that you use GitHub (<https://github.com/>) or Bitbucket (<https://bitbucket.org/>).

2.5 Summary

This chapter covered using the same database in development as in production, `pip`, `virtualenv`, and version control systems. These are good to have in your toolchest, since they are commonly used not just in Django, but in the majority of Python software development.

3 | How to Lay Out Django Projects

Project layout is one of those areas where core Django developers have differing opinions about what they consider best practice. In this chapter, we present our approach, which is one of the most commonly-used ones.

3.1 Django 1.5's Default Project Layout

Let's examine the default project layout that gets created when you run `startproject` and `startapp`:

EXAMPLE 3.1

```
$ django-admin.py startproject mysite
$ cd mysite
$ django-admin.py startapp my_app
```

Here's the resulting project layout:

EXAMPLE 3.2

```
mysite/
  manage.py
  my_app/
    __init__.py
    models.py
    tests.py
    views.py
  mysite/
    __init__.py
```

```
settings.py
urls.py
wsgi.py
```

3.2 Our Preferred Project Layout

We rely on a three-tiered approach that builds on what is generated by the `django-admin.py startproject` management command. We place that inside another directory which serves as the git repository root. Our layouts at the highest level are:

EXAMPLE 3.3

```
<repository_root>/
  <django_project_root>/
    <configuration_root>/
```

Let's go over each level in detail:

3.2.1 Top Level: Repository Root

The top-level `<repository_root>/` directory is the absolute root directory of the project. In addition to the `<django_project_root>` we also place other critical components like the *README.rst*, *docs/* directory, *design/* directory, *.gitignore*, *requirements.txt* files, and other high-level files that are required for deployment.

3.2.2 Second Level: Django Project Root

Generated by the `django-admin.py startproject` command, this is what is traditionally considered the Django project root.

This directory contains the `<configuration_root>`, media and static directories, a site-wide templates directory, as well as Django apps specific to your particular project.

TIP: Common Practice Varies Here

Some developers like to make the `<django_project_root>` the `<repository_root>` of the project.

3.2.3 Third Level: Configuration Root

Also generated by the `django-admin.py startproject` command, the `<configuration_root>` directory is where the settings module and base URLConf (`urls.py`) are placed. This must be a valid Python package (containing an `__init__.py` module).

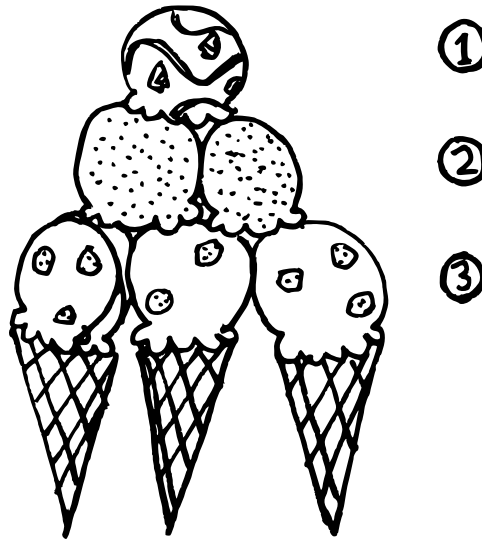


Figure 3.1: Three-tiered scoop layout.

3.3 Sample Project Layout

Let's take a common example: a simple rating site. Imagine that we are creating Ice Cream Ratings, a web application for rating different brands and flavors of ice cream.

This is how we would lay out such a project:

EXAMPLE 3.4

```
icecreamratings_project/
  .gitignore
  Makefile
  docs/
  README.rst
  requirements.txt
  icecreamratings/
    manage.py
    media/
    products/
    profiles/
    ratings/
    static/
    templates/
  icecreamratings/
    __init__.py
    settings/
    urls.py
    wsgi.py
```

Let’s do an in-depth review of this layout. As you can see, in the *icecreamratings_project/* directory, which is the *<repository_root>*, we have the following files and directories. We describe them in the table below:

File or Directory	Purpose
<i>.gitignore</i>	Lists the files and directories that Git should ignore. (This file is different for other version control systems. For example, if you are using Mercurial instead, you’d have an <i>.hgignore</i> file.)
<i>README.rst</i> and <i>docs/</i>	Developer-facing project documentation. You’ll read more about this in chapter 19, Documentation .
<i>Makefile</i>	Contains simple deployment tasks and macros. For more complex deployments you may want to rely on tools like Fabric .

File or Directory	Purpose
<i>requirements.txt</i>	A list of Python packages required by your project, including the Django 1.5 package. You'll read more about this in chapter 17 , <i>Django's Secret Sauce: Third-Party Packages</i> .
<i>icecreamratings/</i>	The <django_project_root> of the project.

Table 3.1: Repository Root Files and Directories

When anyone visits this project, they are provided with a high-level view of the project. We've found that this allows us to work easily with other developers and even non-developers. For example, it's not uncommon for designer-focused directories to be created in the root directory.

Many developers like to make this at the same level as our <repository_root>, and that's perfectly alright with us. We just like to see our projects a little more separated.

Inside the *icecreamratings_project/icecreamratings* directory, at the <django_project_root>, we place the following files/directories:

File or Directory	Purpose
<i>manage.py</i>	If you leave this in, don't modify its contents. Refer to chapter 5 , <i>Settings and Requirements Files</i> for more details.
<i>media/</i>	User-generated static media assets such as photos uploaded by users. For larger projects, this will be hosted on separate static media server(s).
<i>products/</i>	App for managing and displaying ice cream brands.
<i>profiles/</i>	App for managing and displaying user profiles.
<i>ratings/</i>	App for managing user ratings.
<i>static/</i>	Non-user-generated static media assets including CSS, JavaScript, and images. For larger projects, this will be hosted on separate static media server(s).
<i>templates/</i>	Where you put your site-wide Django templates.
<i>icecreamratings/</i>	The <configuration_root> of the project, where project-wide <i>settings</i> , <i>urls.py</i> , and <i>wsgi.py</i> modules are placed (We'll cover settings layout later in chapter 5 , <i>Settings and Requirements Files</i>).

File or Directory	Purpose
-------------------	---------

Table 3.2: Django Project Files and Directories

TIP: Conventions For Static Media Directory Names

In the example above, we follow the official Django documentation's convention of using *static/* for the (non-user-generated) static media directory.

If you find this confusing, there's no harm in calling it *assets/* or *site_assets/* instead. Just remember to update your `STATICFILES_DIRS` setting appropriately.

3.4 What About the Virtualenv?

Notice how there is no virtualenv directory anywhere in the project directory or its subdirectories? That is completely intentional.

A good place to create the virtualenv for this project would be a separate directory where you keep all of your virtualenvs for all of your Python projects. We like to put all our environments in one directory and all our projects in another.

On Mac OS X or Linux:

EXAMPLE 3.5

```
~/projects/icecreamratings_project/  
~/.envs/icecreamratings/
```

On Windows:

EXAMPLE 3.6

```
c:\projects\icecreamratings_project\  
c:\envs\icecreamratings\
```

If you're using `virtualenvwrapper` (only for Mac OS X or Linux), that directory defaults to `~/.virtualenvs/` and the `virtualenv` would be located at:

EXAMPLE 3.7

```
~/.virtualenvs/icecreamratings/
```

TIP: Listing Current Dependencies

If you have trouble determining which versions of dependencies you are using in your `virtualenv`, at the command-line you can list your dependencies by typing:

EXAMPLE 3.8

```
$ pip freeze --local
```

Also, remember, there's no need to keep the contents of your `virtualenv` in version control since it already has all the dependencies captured in *requirements.txt*, and since you won't be editing any of the source code files in your `virtualenv` directly. Just remember that *requirements.txt* does need to remain in version control!

3.5 Using a Startproject Template to Generate Our Layout

Want to use our layout with a minimum of fuss? If you have Django 1.5 (or even Django 1.4), you can use the `startproject` command as follows, all on one line:

EXAMPLE 3.9

```
$ django-admin.py startproject --template=https://github.com/
twoscoops/django-twoscoops-project/zipball/master
--extension=py,rst,html icecreamratings_project
```

This will create an `icecreamratings_project` where you run the command, and this follows the layout example we provided. It also builds settings, requirements, and templates in the same pattern as those items are described later in the book.

3.6 Other Alternatives

As we mentioned, there's no one right way when it comes to project layout. It's okay if a project differs from our layout, just so long as things are either done in a hierarchical fashion or the locations of elements of the project (docs, templates, apps, settings, etc) are documented in the root *README.rst*.

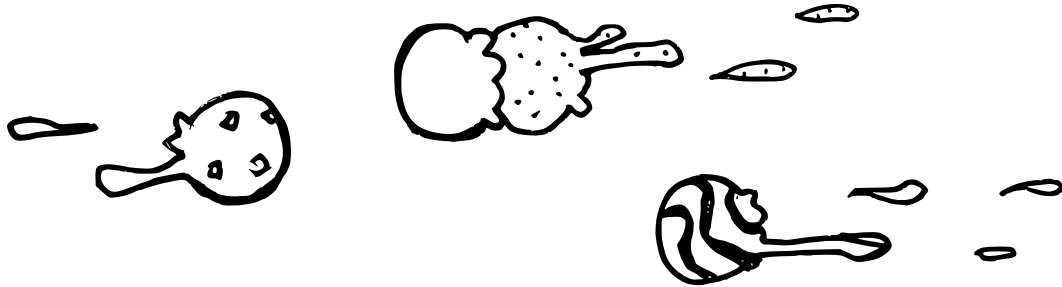


Figure 3.2: Project layout differences of opinion can cause ice cream fights.

3.7 Summary

In this chapter, we covered our approach to basic Django project layout. We provided a detailed example to give you as much insight as possible into our practices.

Project layout is one of those areas of Django where practices differ widely from developer to developer and group to group. What works for a small team may not work for a large team with distributed resources. Whatever layout is chosen should be documented clearly.

4 | Fundamentals of Django App Design

It's not uncommon for new Django developers to become understandably confused by Django's usage of the word "app." So before we get into Django app design, it's very important that we go over some definitions.

A Django project is a web application powered by the Django web framework.

Django apps are small libraries designed to represent a single aspect of a project. A Django project is made up of many Django apps. Some of those apps are internal to the project and will never be reused; others are third-party Django packages.

Third-party Django packages are simply pluggable, reusable Django apps that have been packaged with the Python packaging tools. We'll begin coverage of them in [chapter 17](#), *Django's Secret Sauce: Third-Party Packages*.

4.1 The Golden Rule of Django App Design

James Bennett serves as both a Django core developer and its release manager. He taught us everything we know about good Django app design. We quote him:

"The art of creating and maintaining a good Django app is that it should follow the truncated Unix philosophy according to Douglas McIlroy: 'Write programs that do one thing and do it well.'"

In essence, **each app should be tightly focused on its task**. If an app can't be explained in a single sentence of moderate length, or you need to say 'and' more than once, it probably means the app is too big and should be broken up.

4.1.1 A Practical Example of Apps in a Project

Imagine that we're creating a web application for our fictional ice cream shop called "Two Scoops." Picture us getting ready to open the shop: polishing the countertops, making the first batches of ice cream, and building the website for our shop.

We'd call the Django project for our shop's website *twoscoops_project*. The apps within our Django project might be something like:

- A *flavors* app to track all of our ice cream flavors and list them on our website.
- A *blog* app for the official Two Scoops blog.
- An *events* app to display listings of our shop's events on our website: events such as Strawberry Sundae Sundays and Fudgy First Fridays.

Each one of these apps does one particular thing. Yes, the apps relate to each other, and you could imagine *events* or *blog* posts that are centered around certain ice cream flavors, but it's much better to have three specialized apps than one app that does everything.

In the future, we might extend the site with apps like:

- A *shop* app to allow us to sell pints by mail order.
- A *tickets* app, which would handle ticket sales for premium all-you-can-eat ice cream fests.

Notice how events are kept separate from ticket sales. Rather than expanding the *events* app to sell tickets, we create a separate *tickets* app because most events don't require tickets, and because event calendars and ticket sales have the potential to contain complex logic as the site grows.

Eventually, we hope to use the *tickets* app to sell tickets to Icecreamlandia, the ice cream theme park filled with thrill rides that we've always wanted to open.

Did we say that this was a fictional example? Ahem...well, here's an early concept map of what we envision for Icecreamlandia:

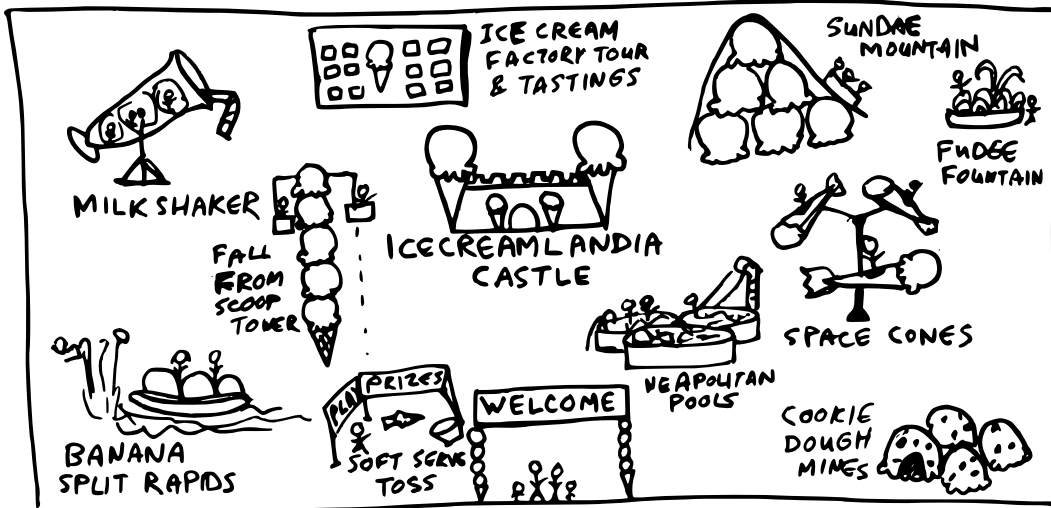


Figure 4.1: Our vision for Icecreamlandia.

4.2 What to Name Your Django Apps

Everyone has their own conventions, and some people like to use really colorful names. We like to use naming systems that are dull, boring, and obvious. In fact, we advocate doing the following:

When possible keep to single word names like *flavors*, *animals*, *blog*, *polls*, *dreams*, *estimates*, and *finances*. A good, obvious app name makes the project easier to maintain.

As a general rule, the app's name should be a plural version of the app's main model, but there are many good exceptions to this rule, *blog* being one of the most common ones.

Don't just consider the app's main model, though. You should also consider how you want your URLs to appear when choosing a name. If you want your site's blog to appear at <http://www.example.com/weblog/>, then consider naming your app *weblog* rather than *blog*, *posts*, or *blogposts*, even if the main model is `Post`, to make it easier for you to see which app corresponds with which part of the site.

Use valid, PEP 8-compliant, importable Python package names: short, all-lowercase names without numbers, dashes, periods, spaces, or special characters. If needed for readability, you can use underscores to separate words, although the use of underscores is discouraged.

4.3 When in Doubt, Keep Apps Small

Don't worry too hard about getting app design perfect. It's an art, not a science. Sometimes you have to rewrite them or break them up. That's okay.

Try and keep your apps small. Remember, it's better to have many small apps than to have a few giant apps.

4.4 Summary

This chapter covered the art of Django app design. Specifically, each Django app should be tightly-focused on its own task, possess a simple, easy-to-remember name. If an app seems too complex, it should be broken up into smaller apps. Getting app design takes practice and effort, but it's well worth the effort.

5 | Settings and Requirements Files

Django 1.5 has over 130 settings that can be controlled in the settings module, most of which come with default values. Settings are loaded when your server starts up, and experienced Django developers stay away from trying to change settings without requiring a restart.

Some best practices we like to follow:

- **All settings files need to be version-controlled.** This is especially true in production environments, where dates, times, and explanations for settings changes absolutely must be tracked.
- **Don't Repeat Yourself.** You should inherit from a base settings file rather than cutting-and-pasting from one file to another.
- **Keep secret keys safe.** They should be kept out of version control.



Figure 5.1: Over 130 settings are available to you in Django 1.5.

5.1 Avoid Non-Versioned Local Settings

We used to advocate the non-versioned **local_settings anti-pattern**. Now we know better.

As developers, we have our own necessary settings for development, such as settings for debug tools which should be disabled (and often not installed to) staging or production servers.

Furthermore, there are often good reasons to keep specific settings out of public or private code repositories. The `SECRET_KEY` setting is the first thing that comes to mind, but API key settings to services like Amazon, Stripe, and other password-type variables need to be protected.

WARNING: Protect Your Secrets!

The `SECRET_KEY` setting is used in Django’s cryptographic signing functionality, and needs to be set to a unique, unpredictable setting best kept out of version control. Running Django with a known `SECRET_KEY` defeats many of Django’s security protections, which can lead to serious security vulnerabilities. For more details, read <https://docs.djangoproject.com/en/1.5/topics/signing/>.

The same warning for `SECRET_KEY` also applies to production database passwords, AWS keys, OAuth tokens, or any other sensitive data that your project needs in order to operate.

We’ll show how to handle the `SECRET_KEY` issue in the “Keep Secret Keys Out With Environment Settings” section.

A common solution is to create *local_settings.py* modules that are created locally per server or development machine, and are purposefully kept out of version control. Developers now make development-specific settings changes, including the incorporation of business logic without the code being tracked in version control. Staging and deployment servers can have location specific settings and logic without them being tracked in version control.

What could possibly go wrong?!?

Ahem...

- Every machine has untracked code.
- How much hair will you pull out, when after hours of failing to duplicate a production bug locally, you discover that the problem was custom logic in a production-only setting?

- How fast will you run from everyone when the ‘bug’ you discovered locally, fixed and pushed to production was actually caused by customizations you made in your own *local.settings.py* module and is now crashing the site?
- Everyone copy/pastes the same *local.settings.py* module everywhere. Isn’t this a violation of Don’t Repeat Yourself but on a larger scale?

Let’s take a different approach. Let’s break up development, staging, test, and production settings into separate components that inherit from a common base file all tracked by version control. We’ll make sure we do it in such a way that server secrets will remain secret.

Read on and see how it’s done!

5.2 Using Multiple Settings Files

TIP: This is Adapted From ‘The One True Way’

The setup described here is based on “The One True Way”, from Jacob Kaplan-Moss’ The Best (and Worst) of Django talk at OSCON 2011 (www.slideshare.net/jacobian/the-best-and-worst-of-django).

Instead of having one *settings.py* file, with this setup you have a *settings/* directory containing your settings files. This directory will typically contain something like the following:

EXAMPLE 5.1

```
settings/  
  __init__.py  
  base.py  
  local.py  
  staging.py  
  test.py  
  production.py
```

WARNING: Requirements + Settings

Each settings module should have its own corresponding requirements file. We'll cover this at the end of this chapter in [section 5.5](#), 'Using Multiple Requirements Files.'

Settings file	Purpose
<i>base.py</i>	Settings common to all instances of the project.
<i>local.py</i>	This is the settings file that you use when you're working on the project locally. Local development-specific settings include <code>DEBUG</code> mode, log level, and activation of developer tools like <code>django-debug-toolbar</code> . Developers sometimes name this file <i>dev.py</i> .
<i>staging.py</i>	Staging version for running a semi-private version of the site on a production server. This is where managers and clients should be looking before your work is moved to production.
<i>test.py</i>	Settings for running tests including test runners, in-memory database definitions, and log settings.
<i>production.py</i>	This is the settings file used by your live production server(s). That is, the server(s) that host the real live website. This file contains production-level settings only. It is sometimes called <i>prod.py</i> .

Table 5.1: Settings files and their purpose

TIP: Multiple Files with Continuous Integration Servers

You'll also want to have a *ci.py* module containing that server's settings. Similarly, if it's a large project and you have other special-purpose servers, you might have custom settings files for each of them.

Let's take a look at how to use the shell and runserver management commands with this setup. You'll have to use the `--settings` command line option, so you'll be entering the following at the command-line.

To start the Python interactive interpreter with Django, using your *settings/local.py* settings file:

EXAMPLE 5.2

```
python manage.py shell --settings=twoscoops.settings.local
```

To run the local development server with your *settings/local.py* settings file:

EXAMPLE 5.3

```
python manage.py runserver --settings=twoscoops.settings.local
```

TIP: DJANGO_SETTINGS_MODULE

A great alternative to using the `--settings` command line option everywhere is to set the `DJANGO_SETTINGS_MODULE` environment variable to your desired settings module path. You'd have to set `DJANGO_SETTINGS_MODULE` to the corresponding settings module for each environment, of course.

For the settings setup that we just described, here are the values to use with the `--settings` command line option or the `DJANGO_SETTINGS_MODULE` environment variable:

Environment	Option To Use With <code>--settings</code> (or <code>DJANGO_SETTINGS_MODULE</code> value)
Your local development server	<code>twoscoops.settings.local</code>
Your staging server	<code>twoscoops.settings.staging</code>
Your test server	<code>twoscoops.settings.test</code>
Your production server	<code>twoscoops.settings.production</code>

Table 5.2: Setting `DJANGO_SETTINGS_MODULE` per location

TIP: Using `django-admin.py` instead of `manage.py`

The official Django documentation says that you should use *django-admin.py* rather than *manage.py* when working with multiple settings files: <https://docs.djangoproject.com/en/1.5/ref/django-admin/>

That being said, if you're struggling with *django-admin.py*, it's perfectly okay to develop and launch your site running it with *manage.py*.

5.2.1 A Development Settings Example

As mentioned earlier, we need settings configured for development, such as setting the email host to localhost, setting the project to run in DEBUG mode, and setting other configuration options that are used solely for development purposes. We place development settings like the following into *settings/local.py*:

EXAMPLE 5.4

```
# settings/local.py
from .base import *

DEBUG = True
TEMPLATE_DEBUG = DEBUG

EMAIL_HOST = "localhost"
EMAIL_PORT = 1025

DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql_psycopg2",
        "NAME": "twoscoops",
        "USER": "",
        "PASSWORD": "",
        "HOST": "localhost",
        "PORT": "",
    }
}
```

```
}

INSTALLED_APPS += ("debug_toolbar", )
INTERNAL_IPS = ("127.0.0.1",)
MIDDLEWARE_CLASSES += \
    ("debug_toolbar.middleware.DebugToolbarMiddleware", )
```

Now try it out at the command line with:

```
EXAMPLE 5.5
python manage.py runserver --settings=twoscoops.settings.local
```

Open <http://127.0.0.1:8000> and enjoy your development settings, ready to go into version control! You and other developers will be sharing the same development settings files, which for shared projects, is awesome.

Yet there's another advantage: No more 'if DEBUG' or 'if not DEBUG' logic to copy/paste around between projects. Settings just got a whole lot simpler!

At this point we want to take a moment to note that Django settings files are the single, solitary place we advocate using `import *`. The reason is that *for the singular case of Django setting modules we want to override all the namespaces*.

5.2.2 Multiple Development Settings

Sometimes we're working on a large project where different developers need different settings, and sharing the same dev.py settings file with teammates won't do.

Well, it's still better tracking these settings in version control than relying on everyone customizing the same dev.py module to their own tastes. A nice way to do this is with multiple dev settings files, e.g. *dev_audreyr.py* and *dev_pydanny.py*:

EXAMPLE 5.6

```
# settings/dev_pydanny.py
from .local import *

# Set short cache timeout
CACHE_TIMEOUT = 30
```

Why? It's not only good to keep all your own settings files in version control, but it's also good to be able to see your teammates' dev settings files. That way, you can tell if someone's missing a vital or helpful setting in their local development setup, and you can make sure that everyone's local settings files are synchronized. Here is what our projects frequently use for settings layout:

EXAMPLE 5.7

```
settings/
  __init__.py
  base.py
  dev_audreyr.py
  dev_pydanny.py
  local.py
  staging.py
  test.py
  production.py
```

5.3 Keep Secret Keys Out With Environment Variables

One of the causes of the `local_settings` anti-pattern is that putting `SECRET_KEY`, AWS keys, API keys, or server-specific values into settings files has problems:

- Secrets often should be just that: secret! Keeping them in version control means that everyone with repository access has access to them.
- Secret keys are configuration values, not code.
- Platforms-as-a-service usually don't give you the ability to edit code on individual servers. Even if they allow it, it's a terribly dangerous practice.

To resolve this, our answer is to use **environment variables**.

Every operating system supported by Django (and Python) provides the easy capability to create environment variables.

Here are the benefits of using environment variables for secret keys:

- Keeping secrets out of settings allows you to store every settings file in version control without hesitation. All of your Python code really should be stored in version control, including your settings.
- Instead of each developer maintaining an easily-outdated, copy-and-pasted version of the *local_settings.py.example* file for their own development purposes, everyone shares the same version-controlled *settings/local.py*.
- System administrators can rapidly deploy the project without having to modify files containing Python code.
- Most platforms-as-a-service recommend the use of environment variables for configuration and have built-in features for setting and managing them.

5.3.1 A Caution Before Using Environment Variables for Secrets

Before you begin setting environment variables, you should have the following:

- A way to manage the secret information you are going to store.
- A good understanding of how bash settings work on servers, or a willingness to have your project hosted by a platform-as-a-service.

For more information, see <http://2scoops.co/wikipedia-env-variable>.

5.3.2 How to Set Environment Variables Locally

On Mac and many Linux distributions that use **bash** for the shell, one can add lines like the following to the end of a *.bashrc*, *.bash_profile*, or *.profile*. When dealing with multiple projects using the same API but with different keys, you can also place these at the end of your virtualenv's *bin/activate* script:

EXAMPLE 5.8

```
$ export SOME_SECRET_KEY=1c3-cr3am-15-yummy
$ export AUDREY_FREEZER_KEY=y34h-r1ght-d0nt-t0uch-my-1c3-cr34m
```

On Windows systems, it's a bit trickier. You can set them one-by-one at the command line (**cmd.exe**) in a persistent way with the `setx` command, but you'll have to close and reopen your command prompt for them to go into effect. A better way is to place these commands at the end of the `virtualenv's bin/activate.bat` script so they are available upon activation:

EXAMPLE 5.9

```
> setx SOME_SECRET_KEY 1c3-cr3am-15-yummy
```

PowerShell is much more powerful than the default Windows shell and comes with Windows Vista and above. Setting environment variables while using PowerShell:

For the current User only:

EXAMPLE 5.10

```
[Environment]::SetEnvironmentVariable("SOME_SECRET_KEY",  
                                     "1c3-cr3am-15-yummy", "User")  
[Environment]::SetEnvironmentVariable("AUDREY_FREEZER_KEY",  
                                     "y34h-r1ght-d0nt-t0uch-my-1c3-cr34m", "User")
```

Machine wide:

EXAMPLE 5.11

```
[Environment]::SetEnvironmentVariable(SOME_SECRET_KEY",  
                                     "1c3-cr3am-15-yummy", "Machine")  
[Environment]::SetEnvironmentVariable(AUDREY_FREEZER_KEY",  
                                     "y34h-r1ght-d0nt-t0uch-my-1c3-cr34m", "Machine")
```

For more information on Powershell, see <http://2scoops.co/powershell>

TIP: virtualenvwrapper Makes This Easier

Mentioned earlier in this book, **virtualenvwrapper**, simplifies per-virtualenv environment variables. It's a great tool. Of course, it requires an understanding of the shell and either Mac OS X or Linux.

5.4 How to Set Environment Variables in Production

If you're using your own servers, your exact practices will differ depending on the tools you're using and the complexity of your setup. For the simplest 1-server setup, it's just a matter of appending to your `.bashrc` file as described above. But if you're using scripts or tools for automated server provisioning and deployment, your approach may be more complex. Check the documentation for your deployment tools for more information.

If your Django project is deployed via a platform-as-a-service, check the documentation for specific instructions. We've included Gondor.io, Heroku, and dotCloud instructions here so that you can see that it's similar for different platform-as-a-service options.

On [Gondor.io](#), you set environment variables with the following command, executed from your development machine:

EXAMPLE 5.12

```
$ gondor env:set SOME_SECRET_KEY=1c3-cr3am-15-yummy
```

On [Heroku](#), you set environment variables with the following command, executed from your development machine:

EXAMPLE 5.13

```
$ heroku config:add SOME_SECRET_KEY=1c3-cr3am-15-yummy
```

On [dotCloud](#), you set environment variables with the following command, executed from your development machine:

EXAMPLE 5.14

```
$ dotcloud env set SOME_SECRET_KEY=1c3-cr3am-15-yummy
```

To see how you access environment variables from the Python side, open up a new Python prompt and type:

EXAMPLE 5.15

```
>>> import os
>>> os.environ["SOME_SECRET_KEY"]
"1c3-cr3am-15-yummy"
```

To access environment variables from one of your settings files, you can do something like this:

EXAMPLE 5.16

```
# Top of settings/production.py
import os
SOME_SECRET_KEY = os.environ["SOME_SECRET_KEY"]
```

This snippet simply gets the value of the `SOME_SECRET_KEY` environment variable from the operating system and saves it to a Python variable called `SOME_SECRET_KEY`.

Following this pattern means all code can remain in version control, and all secrets remain safe.

5.4.1 Handling Missing Secret Key Exceptions

In the above implementation, if the `SECRET_KEY` isn't available, it will throw a `KeyError`, making it impossible to start the project. That's great, but a `KeyError` doesn't tell you that much about what's actually wrong. Without a more helpful error message, this can be hard to debug, especially under the pressure of deploying to servers while users are waiting and your ice cream is melting.

Here's a useful code snippet that makes it easier to troubleshoot those missing environment variables. If you're using our recommended environment variable secrets approach, you'll want to add this to your *settings/base.py* file:

EXAMPLE 5.17

```
# settings/base.py
import os

# Normally you should not import ANYTHING from Django directly
```

```
# into your settings, but ImproperlyConfigured is an exception.
from django.core.exceptions import ImproperlyConfigured

def get_env_variable(var_name):
    """ Get the environment variable or return exception """
    try:
        return os.environ[var_name]
    except KeyError:
        error_msg = "Set the %s environment variable" % var_name
        raise ImproperlyConfigured(error_msg)
```

Then, in any of your settings files, you can load secret keys from environment variables as follows:

EXAMPLE 5.18

```
SOME_SECRET_KEY = get_env_variable("SOME_SECRET_KEY")
```

Now, if you don't have `SOME_SECRET_KEY` set as an environment variable, you get a traceback that ends with a useful error message like this:

EXAMPLE 5.19

```
django.core.exceptions.ImproperlyConfigured: Set the SOME_SECRET_KEY
environment variable.
```

WARNING: Don't Import Django Components Into Settings Modules

This can have many unpredictable side effects, so avoid any sort of import of Django components into your settings. `ImproperlyConfigured` is the exception because it's the official Django exception for...well...improperly configured projects. And just to be helpful we add the name of the problem setting to the error message.

5.5 Using Multiple Requirements Files

Finally, there's one more thing you need to know about the multiple settings files setup. It's good practice for each settings file to have its own corresponding requirements file. This means we're only installing what is required on each server.

To follow this pattern, recommended to us by Jeff Triplett, first create a *requirements/* directory in the **<repository.root>**. Then create *.txt* files that match the contents of your settings directory. The results should look something like:

EXAMPLE 5.20

```
requirements/  
  base.txt  
  local.txt  
  staging.txt  
  production.txt
```

In the *base.txt* file, place the dependencies used in all environments. For example, you might have something like the following in there:

EXAMPLE 5.21

```
Django==1.5.1  
psycopg2==2.4.5  
South==0.7.6
```

Your *local.txt* file should have dependencies used for local development, such as:

EXAMPLE 5.22

```
-r base.txt # includes the base.txt requirements file  
  
coverage==3.6  
django-discover-runner==0.2.2  
django-debug-toolbar==0.9.4
```

The needs of a continuous integration server might prompt the following for a *ci.txt* file:

EXAMPLE 5.23

```
-r base.txt # includes the base.txt requirements file

coverage==3.6
django-discover-runner==0.2.2
django-jenkins==0.13.0
```

Production installations should be close to what is used in other locations, so *production.txt* commonly just calls *base.txt*:

EXAMPLE 5.24

```
-r base.txt # includes the base.txt requirements file
```

5.5.1 Installing From Multiple Requirements Files

For local development:

EXAMPLE 5.25

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

For production:

EXAMPLE 5.26

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

TIP: Don't Know What Dependencies You Installed?

You can use `pip` to output a list of packages that are currently installed in your Python environment. From the command-line, type:

EXAMPLE 5.27

```
$ pip freeze --local
```

5.5.2 Using Multiple Requirements Files With Platforms as a Service (PaaS)

See [section 25.2](#), ‘Using a Platform as a Service’, in [chapter 25](#), *Deploying Django Projects*.

5.6 Handling File Paths in Settings

If you switch to the multiple settings setup and get new filepath errors to things like templates and media, don't be alarmed. This section will help you resolve these errors.

We humbly beseech the reader to never hardcode file paths in Django settings files. This is *really* bad:

BAD EXAMPLE 5.1

```
# settings/base.py

# Configuring MEDIA_ROOT
# 'DONT DO THIS! Hardcoded to just one user's preferences
MEDIA_ROOT = "~/pydanny/twoscoops_project/media"

# Configuring STATIC_ROOT
# 'DONT DO THIS! Hardcoded to just one user's preferences
STATIC_ROOT = "~/pydanny/twoscoops_project/collected_static"

# Configuring TEMPLATE_DIRS
# 'DONT DO THIS! Hardcoded to just one user's preferences
```



```
TEMPLATE_DIRS = (  
    "~/pydanny/twoscoops_project/templates",  
)
```

The above code represents a common pitfall called **hardcoding**. The above code, called a **fixed path**, is bad because as far as you know, **pydanny** (Daniel Greenfeld) is the only person who has set up their computer to match this path structure. Anyone else trying to use this example will see their project break, forcing them to either change their directory structure (unlikely) or change the settings module to match their preference (causing problems for everyone else including pydanny).

Don't hardcode your paths!

To fix the path issue, we dynamically set a project root variable intuitively named `PROJECT_ROOT` at the top of the base settings module. Since `PROJECT_ROOT` is determined in relation to the location of `base.py`, your project can be run from any location on any development computer or server.

We find the cleanest way to set a `PROJECT_ROOT`-like setting is with **Unipath** (<http://pypi.python.org/pypi/Unipath/>), a Python package that does elegant, clean path calculations:

```
EXAMPLE 5.28  
  
# At the top of settings/base.py  
from unipath import Path  
  
PROJECT_DIR = Path(__file__).ancestor(3)  
MEDIA_ROOT = PROJECT_DIR.child("media")  
STATIC_ROOT = PROJECT_DIR.child("static")  
STATICFILES_DIRS = (  
    PROJECT_DIR.child("assets"),  
)  
TEMPLATE_DIRS = (  
    PROJECT_DIR.child("templates"),  
)
```

If you really want to set your `PROJECT_ROOT` with the Python standard library's `os.path` library,

though, this is one way to do it in a way that will account for paths:

EXAMPLE 5.29

```
# At the top of settings/base.py
from os.path import join, abspath, dirname

here = lambda *x: join(abspath(dirname(__file__)), *x)
PROJECT_ROOT = here("../", "..")
root = lambda *x: join(abspath(PROJECT_ROOT), *x)

# Configuring MEDIA_ROOT
MEDIA_ROOT = root("media")

# Configuring STATIC_ROOT
STATIC_ROOT = root("collected_static")

# Additional locations of static files
STATICFILES_DIRS = (
    root("assets"),
)

# Configuring TEMPLATE_DIRS
TEMPLATE_DIRS = (
    root("templates"),
)
```

With your various path settings dependent on `PROJECT_ROOT`, your filepath settings should work, which means your templates and media should be loading without error.

TIP: How different are your settings from the Django defaults?

If you want to know how things in your project differ from Django's defaults, use the `diffsettings` management command.

5.7 Summary

Remember, everything except for critical security related values ought to be tracked in version control.

Any project that's destined for a real live production server is bound to need multiple settings and requirements files. Even beginners to Django need this kind of settings/requirements file setup once their projects are ready to leave the original development machine. We provide our solution since it works well for both beginning and advanced developers.

The same thing applies to requirements files. Working with untracked dependency differences increases risk as much as untracked settings.

6 | Database/Model Best Practices

Models are the foundation of most Django projects. Racing to write Django models without thinking things through can lead to problems down the road.

All too frequently we developers rush into adding or modifying models without considering the ramifications of what we are doing. The quick fix or sloppy “temporary” design decision that we toss into our code base now can hurt us in the months or years to come, forcing crazy workarounds or corrupting existing data.

So keep this in mind when adding new models in Django or modifying existing ones. Take your time to think things through, and design your foundation to be as strong and sound as possible.

PACKAGE TIP: Our Picks For Working With Models

Here’s a quick list of the model-related Django packages that we use in practically every project.

- South for database migrations. South is so commonplace these days that using it has become a de facto best practice. We’ll cover tips for working with South later in this chapter.
- django-model-utils to handle common patterns like **TimeStampedModel**.
- django-extensions has a powerful management command called ‘shell_plus’ which autoloads the model classes for all installed apps. The downside of this library is that it includes a lot of other functionality which breaks from our preference for small, focused apps.

6.1 Basics

6.1.1 Break Up Apps With Too Many Models

If there are 20+ models in a single app, think about ways to break it down into smaller apps, as it probably means your app is doing too much. In practice, we like to lower this number to no more than five models per app.

6.1.2 Don't Drop Down to Raw SQL Until It's Necessary

Most of the queries we write are simple. The ORM provides a great productivity shortcut: writing decent SQL that comes complete with validation and security. If you can write your query easily with the ORM, then take advantage of it!

It's also good to keep in mind that if you ever release one of your Django apps as a third-party package, using raw SQL will decrease the portability of the work.

Finally, in the rare event that the data has to be migrated from one database to another, any database-specific features that you use in your SQL queries will complicate the migration.

So when should you actually write raw SQL? If expressing your query as raw SQL would drastically simplify your Python code or the SQL generated by the ORM, then go ahead and do it. For example, if you're chaining a number of `QuerySet` operations that each operate on a large data set, there may be a more efficient way to write it as raw SQL.

TIP: Malcolm Tredinnick's Advice On Writing SQL in Django

Django core developer Malcolm Tredinnick said (paraphrased):

“The ORM can do many wonderful things, but sometimes SQL is the right answer. The rough policy for the Django ORM is that it's a storage layer that happens to use SQL to implement functionality. If you need to write advanced SQL you should write it. I would balance that by cautioning against overuse of the `raw()` and `extra()` methods.”

TIP: Jacob Kaplan-Moss' Advice On Writing SQL in Django

Django project co-leader Jacob Kaplan-Moss says (paraphrased):

“If it’s easier to write a query using SQL than Django, then do it. `extra()` is nasty and should be avoided; `raw()` is great and should be used where appropriate.”

6.1.3 Add Indexes as Needed

While adding `db_index=True` to any model field is easy, understanding when it should be done takes a bit of judgment. Our preference is to start without indexes and add them as needed.

When to consider adding indexes:

- The index is used frequently, as in 10–25% of all queries.
- There is real data, or something that approximates real data, so we can analyze the results of indexing.
- We can run tests to determine if indexing generates an improvement in results.

When using PostgreSQL, `pg_stat_activity` tells us what indexes are actually being used.

Once a project goes live, [chapter 20](#), *Finding and Reducing Bottlenecks*, has information on index analysis.

6.1.4 Be Careful With Model Inheritance

Model inheritance in Django is a tricky subject. Django provides three ways to do model inheritance: **abstract base classes**, **multi-table inheritance**, and **proxy models**.

WARNING: Django Abstract Base Classes <> Python Abstract Base Classes

Don’t confuse Django abstract base classes with the abstract base classes in the Python standard library’s `abc` module, as they have very different purposes and behaviors.

Here are the pros and cons of the three model inheritance styles. To give a complete comparison, we also include the option of using no model inheritance to begin with:

Model Inheritance Style	Pros	Cons
No model inheritance: if models have a common field, give both models that field.	Makes it easiest to understand at a glance how Django models map to database tables.	If there are a lot of fields duplicated across models, this can be hard to maintain.
Abstract base classes: tables are only created for derived models.	Having the common fields in an abstract parent class saves us from typing them more than once. We don't get the overhead of extra tables and joins that are incurred from multi-table inheritance.	We cannot use the parent class in isolation.
Multi-table inheritance: tables are created for both parent and child. An implied <code>OneToOneField</code> links parent and child.	Gives each model its own table, so that we can query either parent or child model. Also gives us the ability to get to a child object from a parent object: <code>parent.child</code>	Adds substantial overhead since each query on a child table requires joins with all parent tables. We strongly recommend against using multi-table inheritance. See the warning below.
Proxy models: a table is only created for the original model.	Allows us to have an alias of a model with different Python behavior.	We cannot change the model's fields.

Table 6.1: Pros and Cons of the Model Inheritance Styles

WARNING: Avoid Multi-Table Inheritance

Multi-table inheritance, sometimes called “concrete inheritance,” is considered by the authors and many other developers to be a bad thing. We strongly recommend against using it. We’ll go into more detail about this shortly.

Here are some simple rules of thumb for knowing which type of inheritance to use and when:

- If the overlap between models is minimal (e.g. you only have a couple of models that share one

or two obvious fields), there might not be a need for model inheritance. Just add the fields to both models.

- If there is enough overlap between models that maintenance of models' repeated fields causes confusion and inadvertent mistakes, then in most cases the code should be refactored so that the common fields are in an abstract base class.
- Proxy models are an occasionally-useful convenience feature, but they're very different from the other two model inheritance styles.
- At all costs, everyone should avoid multi-table inheritance (see warning above) since it adds both confusion and substantial overhead. Instead of multi-table inheritance, use explicit `OneToOneFields` and `ForeignKeys` between models so you can control when joins are traversed.

6.1.5 Model Inheritance in Practice: The `TimeStampedModel`

It's very common in Django projects to include a `created` and `modified` timestamp field on all your models. We could manually add those fields to each and every model, but that's a lot of work and adds the risk of human error. A better solution is to write a `TimeStampedModel` to do the work for us:

EXAMPLE 6.1

```
# core/models.py
from django.db import models

class TimeStampedModel(models.Model):
    """
    An abstract base class model that provides self-
    updating ``created`` and ``modified`` fields.
    """
    created = models.DateTimeField(auto_now_add=True)
    modified = models.DateTimeField(auto_now=True)

    class Meta:
        abstract = True
```

Take careful note of the very last two lines in the example, which turn our example into an abstract base class:

EXAMPLE 6.2

```
class Meta:
    abstract = True
```

By defining `TimeStampedModel` as an abstract base class when we define a new class that inherits from it, Django doesn't create a `model_utils.time_stamped_model` table when `syncdb` is run.

Let's put it to the test:

EXAMPLE 6.3

```
# flavors/models.py
from django.db import models

from core.models import TimeStampedModel

class Flavor(TimeStampedModel):
    title = models.CharField(max_length=200)
```

This only creates one table: the `flavors_flavor` database table. That's exactly the behavior we wanted.

On the other hand, if `TimeStampedModel` was not an abstract base class (i.e. a concrete base class via multi-table inheritance), it would also create a `model_utils.time_stamped_model` table. Not only that, but all of its subclasses including `Flavor` would lack the fields and have implicit foreign keys back to `TimeStampedModel` just to handle `created/modified` timestamps. Any reference to `Flavor` that reads or writes to the `TimeStampedModel` would impact two tables. (Thank goodness it's abstract!)

Remember, concrete inheritance has the potential to become a nasty performance bottleneck. This is even more true when you subclass a concrete model class multiple times.

Further reading:

➤ <http://2scoops.co/1.5-model-inheritance>

6.1.6 Use South for Migrations

South is one of those rare third-party packages that almost everyone in the Django community uses these days. South is a tool for managing data and schema migrations. Get to know South's features well.

A few South tips:

- As soon as a new app or model is created, take that extra minute to create the initial South migrations for that new app or model.
- Write reverse migrations and test them! You can't always write perfect round-trips, but not being able to back up to an earlier state really hurts bug tracking and sometimes deployment in larger projects.
- While working on a Django app, flatten migration(s) to just one before pushing the new code to production. In other words, commit **"just enough migrations"** to get the job done.
- Never remove migration code that's already in production.
- If a project has tables with millions of rows in them, do extensive tests against data of that size on staging servers before running a South migration on a production server. Migrations on real data can take much, much, much more time than anticipated.

WARNING: Don't Remove Migrations From Existing Projects In Production

We're reiterating the bullet on removing migrations from existing projects in production. Regardless of any justifications given for removing migrations, doing so removes the history of the project at a number of levels. Removing migrations is analogous to deleting an audit trail, and any problems that may be caused by deletion of migrations might not be detectable for some time.

6.2 Django Model Design

One of the most difficult topics that receives the least amount of attention is how to design good Django models.

How do you design for performance without optimizing prematurely? Let's explore some strategies here.

6.2.1 Start Normalized

We suggest that readers of this book need to be familiar with **database normalization**. If you are unfamiliar with database normalization, make it your responsibility to gain an understanding, as working with models in Django effectively requires a working knowledge of this. Since a detailed explanation of the subject is outside the scope of this book, we recommend the following resources:

- http://en.wikipedia.org/wiki/Database_normalization
- http://en.wikibooks.org/wiki/Relational_Database_Design/Normalization

When you're designing your Django models, always start off normalized. Take the time to make sure that no model should contain data already stored in another model.

At this stage, use relationship fields liberally. Don't denormalize prematurely. You want to have a good sense of the shape of your data.

6.2.2 Cache Before Denormalizing

Often, setting up caching in the right places can save you the trouble of denormalizing your models. We'll cover caching in much more detail in [chapter 20](#), *Finding and Reducing Bottlenecks*, so don't worry too much about this right now.

6.2.3 Denormalize Only if Absolutely Needed

It can be tempting, especially for those new to the concepts of data normalization, to denormalize prematurely. Don't do it! Denormalization may seem like a panacea for what causes problems in a project. However it's a tricky process that risks adding complexity to your project and dramatically raises the risk of losing data.

Please, please, please explore caching before denormalization.

When a project has reached the limits of what the techniques described in [chapter 20](#), *Finding and Reducing Bottlenecks* can address, that's when research into the concepts and patterns of database denormalization should begin.

6.2.4 When to Use Null and Blank

When defining a model field, you have the ability to set the `null=True` and the `blank=True` options. By default, they are `False`.

Knowing when to use these options is a common source of confusion for developers.

We've put this guide together to serve as a guide for standard usage of these model field arguments.

Field Type	Setting <code>null=True</code>	Setting <code>blank=True</code>
<code>CharField</code> , <code>TextField</code> , <code>SlugField</code> , <code>EmailField</code> , <code>CommaSeparatedIntegerField</code> , etc.	<i>Don't do this.</i> Django's convention is to store empty values as the empty string, and to always retrieve <code>NULL</code> or empty values as the empty string for consistency.	<i>Okay.</i> Do this if you want the corresponding form widget to accept empty values. If you set this, empty values get stored as empty strings in the database.
<code>BooleanField</code>	<i>Don't do this.</i> Use <code>NullBooleanField</code> instead.	<i>Don't do this.</i>
<code>IntegerField</code> , <code>FloatField</code> , <code>DecimalField</code> , etc	<i>Okay</i> if you want to be able to set the value to <code>NULL</code> in the database.	<i>Okay</i> if you want the corresponding form widget to accept empty values. If so, you will also want to set <code>null=True</code> .
<code>DateTimeField</code> , <code>.DateField</code> , <code>TimeField</code> , etc.	<i>Okay</i> if you want to be able to set the value to <code>NULL</code> in the database.	<i>Okay</i> if you want the corresponding form widget to accept empty values, or if you are using <code>auto_now</code> or <code>auto_now_add</code> . If so, you will also want to set <code>null=True</code> .
<code>ForeignKey</code> , <code>ManyToManyField</code> , <code>OneToOneField</code>	<i>Okay</i> if you want to be able to set the value to <code>NULL</code> in the database.	<i>Okay</i> if you want the corresponding form widget (e.g. the select box) to accept empty values.

Field Type	Setting null=True	Setting blank=True
IPAddressField, GenericIPAddress- Field	Okay if you want to be able to set the value to NULL in the database.	<i>Not recommended.</i> In PostgreSQL, the native inet type is used here and cannot be set to the empty string. (Other database backends use char or varchar for this, though.)

Table 6.2: When To use Null and Blank by Field

WARNING: IPAddressField in PostgreSQL

At the time of this writing, there is an open ticket ([#5622](#)) related to IPAddressFields: ‘Empty ipaddress raises an error (invalid input syntax for type inet: "") [sic].’

Until this ticket is resolved, we recommend using `null=True` and `blank=False` with IPAddressFields.

See <http://code.djangoproject.com/ticket/5622> for more details.

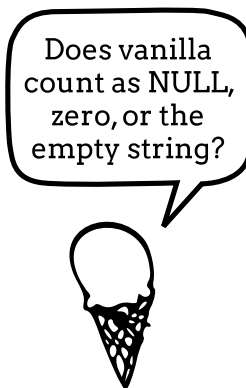


Figure 6.1: A common source of confusion.

6.3 Model Managers

Every time we use the Django ORM to query a model, we are using an interface called a **model manager** to interact with the database. Model managers are said to act on the full set of all possible

instances of this model class (all the data in the table) to restrict the ones you want to work with. Django provides a default model manager for each model class, but we can define our own.

Here's a simple example of a custom model manager:

EXAMPLE 6.4

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

class PublishedManager(models.Manager):

    use_for_related_fields = True

    def published(self, **kwargs):
        return self.filter(pub_date__lte=timezone.now(), **kwargs)

class FlavorReview(models.Model):
    review = models.CharField(max_length=255)
    pub_date = models.DateTimeField()

    # add our custom model manager
    objects = PublishedManager()
```

Now, if we first want to display a count of all of the ice cream flavor reviews, and then a count of just the published ones, we can do the following:

EXAMPLE 6.5

```
>>> from reviews.models import FlavorReview
>>> FlavorReview.objects.count()
35
>>> FlavorReview.objects.published().count()
31
```

Easy, right? Yet wouldn't it make more sense if you just added a second model manager? That way you could have something like:

BAD EXAMPLE 6.1

```
>>> from reviews.models import FlavorReview
>>> FlavorReview.objects.filter().count()
35
>>> FlavorReview.published.filter().count()
31
```

On the surface, replacing the default model manager seems like the obvious thing to do. Unfortunately, our experiences in real project development makes us very careful when we use this method. Why?

First, when using model inheritance, children of abstract base classes receive their parent's model manager, and children of concrete base classes do not.

Second, the first manager applied to a model class is the one that Django treats as the default. This breaks significantly with the normal Python pattern, causing what can appear to be unpredictable results from QuerySets.

With this knowledge in mind, in your model class, `objects = models.Manager()` should be defined manually above any custom model manager.

WARNING: Know the Model Manager Order of Operations

Always set `objects = models.Manager()` above any custom model manager that has a new name.

Additional reading:

➤ <https://docs.djangoproject.com/en/1.5/topics/db/managers/>

6.4 Summary

Models are the foundation for most Django projects, so take the time to design them thoughtfully.

Start normalized, and only denormalize if you've already explored other options thoroughly. You may be able to simplify slow, complex queries by dropping down to raw SQL, or you may be able to address your performance issues with caching in the right places.

Don't forget to use indexes. Add indexes when you have a better feel for how you're using data throughout your project.

If you decide to use model inheritance, inherit from abstract base classes rather than concrete models. You'll save yourself from the confusion of dealing with implicit, unneeded joins.

Watch out for the “gotchas” when using the `null=True` and `blank=True` model field options. Refer to our handy table for guidance.

Finally, use South to manage your data and schema migrations. It's a fantastic tool. You may also find `django-model-utils` and `django-extensions` pretty handy.

Our next chapter is all about views.

7 | Function- and Class-Based Views

Both function-based views (FBVs) and class-based views (CBVs) are in Django 1.5. We recommend that you understand how to use both types of views.

TIP: Function-Based Views Are Not Deprecated

There was a bit of confusion about this due to the wording of the release notes and incorrect information on some blog posts. To clarify:

- ❶ **Function-based views are still in Django 1.5.** No plans exist for removing function-based views from Django. They are in active use, and they are great to have when you need them.
- ❷ Function-based generic views such as `direct_to_template` and `object_list` were deprecated in Django 1.3 and removed in 1.5.

7.1 When to Use FBVs or CBVs

Whenever you implement a view, think about whether it would make more sense to implement as a FBV or as a CBV. Some views are best implemented as CBVs, and others are best implemented as FBVs.

If you aren't sure which method to choose, on the next page we've included a flow chart that might be of assistance.

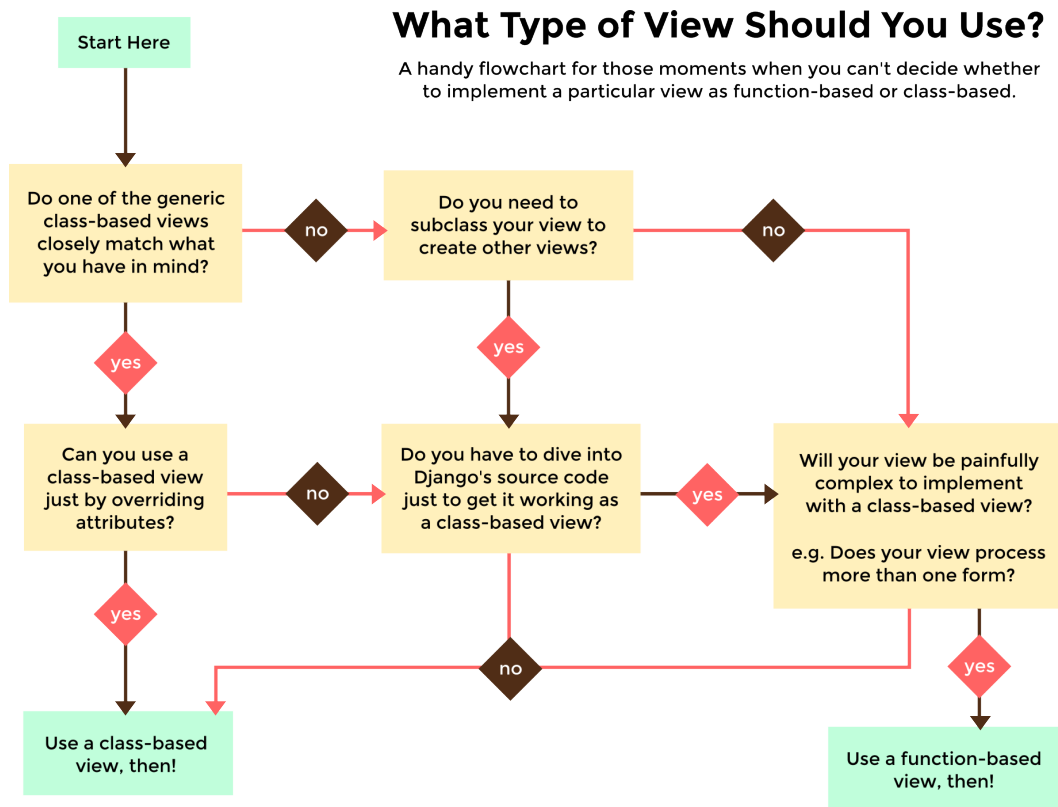


Figure 7.1: Should you use a FBV or a CBV? flow chart.

This flowchart follows our preference for using CBVs over FBVs. We prefer to use CBVs for most views, using FBVs to implement only the complicated views that would be a pain to implement with CBVs.

TIP: Alternative Approach - Staying With FBVs

Some developers prefer to err on the side of using FBVs for most views and CBVs only for views that need to be subclassed. That strategy is fine as well.

7.2 Keep View Logic Out of URLConfs

Requests are routed to views via **URLConfs**, in a module that is normally named *urls.py*. Per Django's URL design philosophy (<http://2scoops.co/1.5-url-design>), the coupling of views with urls is loose, allows for infinite flexibility, and encourages best practices.

And yet, this is what Daniel feels like yelling every time he sees complex *urls.py* files:

"I didn't write J2EE XML and Zope ZCML configuration files back in the day just so you darn kids could stick logic into Django url files!"

Remember that Django has a wonderfully simple way of defining URL routes. Like everything else we bring up in this book, that simplicity is to be honored and respected. The rules of thumb are obvious:

- ❶ The views modules should contain view logic.
- ❷ The URL modules should contain URL logic.

Ever see code like this? Perhaps in the official Django tutorial?

BAD EXAMPLE 7.1

```
from django.conf.urls import patterns, url
from django.views.generic import DetailView

from tastings.models import Tasting

urlpatterns = patterns("",
    url(r"^(?P<pk>\d+)/$",
        DetailView.as_view(
            model=Tasting,
            template_name="tastings/detail.html"),
        name="detail"),
    url(r"^(?P<pk>\d+)/results/$",
        DetailView.as_view(
            model=Tasting,
            template_name="tastings/results.html"),
        name="results"),
)
```

At a glance this code might seem okay, but we argue that it violates the Django design philosophies:

- **Loose coupling** between views, urls, and models has been replaced with tight coupling, meaning you can never reuse the view definitions.
- **Don't Repeat Yourself** is violated by using the same/similar arguments repeatedly between CBVs.
- Infinite Flexibility (for URLs) is destroyed. Class inheritance, the primary advantage of Class Based Views, is impossible using this anti-pattern.
- Lots of other issues: What happens when you have to add in authentication? And what about authorization? Are you going to wrap each URLConf view with two or more decorators? Putting your view code into your URLConfs quickly turns your URLConfs into an unmaintainable mess.

In fact, we've heard from developers that seeing CBVs defined in URLConfs this way was part of why they steered clear of using them.

Alright, enough griping. We'll show our preferences in the next section.

7.3 Stick to Loose Coupling in URLConfs

Here is how to create URLconfs that avoid the problems we mentioned on the previous page. First, we write the views:

EXAMPLE 7.1

```
# tastings/views.py
from django.views.generic import DetailView

from .models import Tasting

class TasteDetailView(DetailView):
    model = Tasting

class TasteResultsView(TasteDetailView):
    template_name = "tastings/results.html"
```

Then we define the urls:

EXAMPLE 7.2

```
# tastings/urls.py
from django.conf.urls import patterns
from django.conf.urls import url

from .views import TasteDetailView
from .views import TasteResultsView

urlpatterns = patterns("",
    url(
        regex=r"^(?P<pk>\d+)/$",
        view=TasteDetailView.as_view(),
        name="detail"
    ),
    url(
        regex=r"^(?P<pk>\d+)/results/$",
        view=TasteResultsView.as_view(),
        name="results"
    ),
)
```

Your first response to our version of this should go something like, “*Are you sure this is a good idea? You changed things to use two files AND more lines of code! How is this better?*”

Well, this is the way we do it. Here are some of the reasons we find it so useful:

- **Don’t Repeat Yourself:** No argument or attribute is repeated between views.
- **Loose coupling:** We’ve removed the model and template names from the URLConf because views should be views and URLConfs should be URLConfs. We should be able to call our views from one or more URLConfs, and our approach lets us do just that.
- **URLConfs should do one thing and do it well:** Related to our previous bullet, our URLConf is now focused primarily on just one thing: URL routing. We aren’t tracking down view logic across both views and URLConfs, we just look in our views.
- **Our views benefit from being class-based:** Our views, by having a formal definition in the views module, can inherit from other classes. This means adding authentication, authorization, new content formats, or anything other business requirement tossed our way is much easier to

handle.

- **Infinite flexibility:** Our views, by having a formal definition in the views module, can implement their own custom logic.

7.3.1 What if we aren't using CBVs?

The same rules apply.

We've encountered debugging nightmares of projects using FBVs with extensive URLConf hackery, such as elaborate tricks with the `__file__` attribute of Python modules combined with directory walking and regular expressions to automagically create URLConfs. If that sounds painful, it was.

Keep logic out of URLConfs!

7.4 Try to Keep Business Logic Out of Views

In the past, we've placed an amazing amount of sophisticated business logic into our views. Unfortunately, when it became time to generate PDFs, add a REST API, or serve out other formats, placing so much logic in our views made it much harder to deliver new formats.

This is where our preferred approach of model methods, manager methods, or general utility helper function come into play. When business logic is placed into easily reusable components, and called from within views, it makes extending components of the project to do more things much easier.

Since it's not always possible to do this at the beginning of a project, our rule of thumb has become whenever we find ourselves duplicating business logic instead of Django boilerplate between views, it's time to move code out of the view.

7.5 Summary

This chapter started with discussing when to use either FBVs or CBVs, and matched our own preference for the latter. In fact, in the next chapter we'll start to dig deep into the functionality that can be exploited when using CBVs.

We also discussed keeping view logic out of the URLConfs. We feel view code belongs in the apps' *views.py* modules, and URLConf code belongs in the apps' *urls.py* modules. Adhering to this practice allows for object inheritance when used with class-based views, easier code reuse, and greater flexibility of design.

8 | Best Practices for Class-Based Views

Since the release of version 1.3, Django has supported class-based views (CBVs). Early problems with CBVs have been addressed almost entirely, thanks to improvements in the core CBV documentation, resources such as Marc Tamlyn and Charles Denton's ccbv.co.uk code inspector, and the advent of `django-braces`.

With a little practice, CBVs allow developers to create views at an astonishing pace. CBVs encourage the reuse of view code, allowing you to create base views and subclass them. They were brought into Django core because of their power and flexibility.

Here is a list of must-read Django CBV documentation:

- <http://2scoops.co/1.5-topics-class-based-views>
- <http://2scoops.co/1.5-cbv-generic-display>
- <http://2scoops.co/1.5-cbv-generic-editing>
- <http://2scoops.co/1.5-cbv-mixins>
- <http://2scoops.co/1.5-ref-class-based-views/>
- The CBV inspector at <http://ccbv.co.uk>

PACKAGE TIP: CBVs + `django-braces` Are Great Together

We feel that `django-braces` is the missing component for Django CBVs. It provides a set of clearly coded mixins that make Django CBVs much easier and faster to implement. The next few chapters will demonstrate its mixins in various code examples.

The power of CBVs comes at the expense of simplicity: CBVs come with a complex inheritance chain that can have up to eight superclasses on import. As a result, trying to work out exactly which view to use or which method to customize can be very challenging at times.

We follow these guidelines when writing CBVs:

- Less view code is better.
- Never repeat code in views.
- Views should handle presentation logic. Try to keep business logic in models when possible, or in forms if you must.
- Keep your views simple.
- Keep your **mixins** simpler.

8.1 Using Mixins With CBVs

Think of mixins in programming along the lines of mixins in ice cream: you can enhance any ice cream flavor by mixing in crunchy candy bits, sliced fruit, or even bacon.

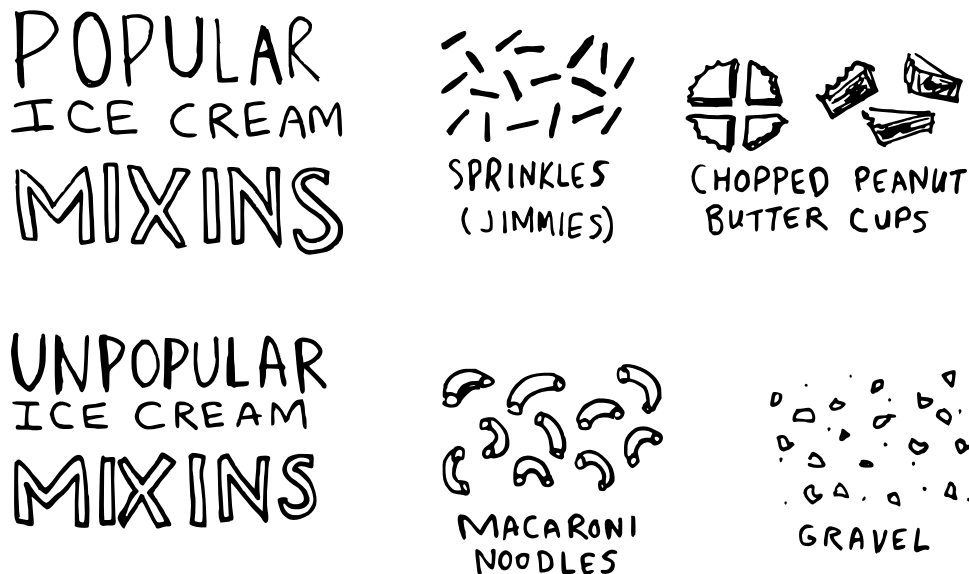


Figure 8.1: Popular and unpopular mixins used in ice cream.

Soft serve ice cream greatly benefits from mixins: ordinary vanilla soft serve turns into birthday cake ice cream when sprinkles, blue buttercream icing, and chunks of yellow cake are mixed in.

In programming, a mixin is a class that provides functionality to be inherited, but isn't meant for instantiation on its own. In programming languages with multiple inheritance, mixins can be used to add enhanced functionality and behavior to classes.

You can use the power of mixins to composite useful and interesting new view classes for your Django apps.

When using mixins to composite your own view classes, we recommend these rules of inheritance provided by Kenneth Love. The rules follow Python's **method resolution order**, which in the most simplistic definition possible, proceeds from left to right:

- ❶ The base view classes provided by Django *always* go to the right.
- ❷ Mixins go to the left of the base view.
- ❸ Mixins should inherit from Python's built-in object type.

Example of the rules in action:

EXAMPLE 8.1

```
from django.views.generic import TemplateView

class FreshFruitMixin(object):

    def get_context_data(self, **kwargs):
        context = super(FreshFruitMixin,
                        self).get_context_data(**kwargs)
        context["has_fresh_fruit"] = True
        return context

class FruityFlavorView(FreshFruitMixin, TemplateView):
    template_name = "fruity_flavor.html"
```

In our rather silly example, the `FruityFlavorView` class inherits from both `FreshFruitMixin` and `TemplateView`.

Since `TemplateView` is the base view class provided by Django, it goes on the far right (rule 1), and to its left we place the `FreshFruitMixin` (rule 2). This way we know that our methods and properties will execute correctly.

Finally, `FreshFruitMixin` inherits from `object` (rule 3).

8.2 Which Django CBV Should Be Used for What Task?

It can be challenging to determine which view you should use where. Some views are very obvious, such as those that perform operations that create, read, update, or delete data, but others are harder to determine.

Here's a handy chart listing the name and purpose of each Django CBV. All views listed here are assumed to be prefixed with `django.views.generic` (prefix omitted in order to save space in the table).

Name	Purpose	Two Scoops Example
<code>View</code>	Base view or handy view that can be used for anything.	See section 11.2 , 'Implementing a Simple JSON API'.
<code>RedirectView</code>	Redirect user to another URL	Send users who visit <code>'/log-in/'</code> to <code>'/login/'</code> .
<code>TemplateView</code>	Display a Django HTML template.	The <code>'/about/'</code> page of our site.
<code>ListView</code>	List objects	List of ice cream flavors.
<code>DetailView</code>	Display an object	Details on an ice cream flavor.
<code>FormView</code>	Submit a form	The site's contact or email form.
<code>CreateView</code>	Create an object	Create a new ice cream flavor.
<code>UpdateView</code>	Update an object	Update an existing ice cream flavor.
<code>DeleteView</code>	Delete an object	Delete an unpleasant ice cream flavor like Vanilla Steak.
Generic date views	For display of objects that occur over a range of time.	Blogs are a common reason to use them. For Two Scoops, we could create a public history of when flavors have been added to the database.

Table 8.1: Django CBV Usage Table

TIP: The Three Schools of Django CBV Usage

We've found that there are three major schools of thought around CBV usage. They are:

The School of "Use all the views"!

This school of thought is based on the idea that since Django provides functionality to reduce your workload, why not use that functionality? We tend to belong to this school of thought to great success, rapidly building and then maintaining a number of projects.

The School of "Just use `django.views.generic.View`"

This school of thought is based on the idea that the base Django CBV does just enough. While we don't follow this approach ourselves, some very good Django developers do.

The School of "Avoid them unless you're actually subclassing views"

Jacob Kaplan-Moss says, "My general advice is to start with function views since they're easier to read and understand, and only use CBVs where you need them. Where do you need them? Any place where you need a fair chunk of code to be reused among multiple views."

We belong to the first school, but it's good for you to know that there's no real consensus on best practices here.

8.3 General Tips for Django CBVs

This section covers useful tips for all or many Django CBV implementations.

8.3.1 Constraining Django CBV Access to Authenticated Users

While the Django CBV documentation gives a helpful working example of using the `django.contrib.auth.decorators.login_required` decorator with a CBV, the example contains a lot of boilerplate cruft: <http://2scoops.co/1.5-login-required-cbv>

Fortunately, `django-braces` provides a ready implementation of a `LoginRequiredMixin` that you

can attach in moments. For example, we could do the following in all of the Django CBVs that we've written so far:

EXAMPLE 8.2

```
# flavors/views.py
from django.views.generic import DetailView

from braces.views import LoginRequiredMixin

from .models import Flavor

class FlavorDetailView(LoginRequiredMixin, DetailView):
    model = Flavor
```

TIP: Don't Forget the CBV Mixin Order!

Remember that:

- LoginRequiredMixin must always go on the far left side.
- The base view class must always go on the far right side.

If you forget and switch the order, you will get broken or unpredictable results.

8.3.2 Performing Custom Actions on Views With Valid Forms

When you need to perform a custom action on a view with a **valid** form, the `form.valid()` method is where the CBV workflow sends the request.

EXAMPLE 8.3

```
from django.views.generic import CreateView

from braces.views import LoginRequiredMixin

from .models import Flavor
```



```
class FlavorCreateView(LoginRequiredMixin, CreateView):
    model = Flavor

    def form_valid(self, form):
        # Do custom logic here
        return super(FlavorCreateView, self).form_valid(form)
```

To perform custom logic on form data that has already been validated, simply add the logic to `form_valid()`. The return value of `form_valid()` should be a `django.http.HttpResponseRedirect`.

8.3.3 Performing Custom Actions on Views With Invalid Forms

When you need to perform a custom action on a view with an **invalid** form, the `form_invalid()` method is where the Django CBV workflow sends the request. This method should return a `django.http.HttpResponse`.

```
EXAMPLE 8.4

from django.views.generic import CreateView

from braces.views import LoginRequiredMixin

from .models import Flavor

class FlavorCreateView(LoginRequiredMixin, CreateView):
    model = Flavor

    def form_invalid(self, form):
        # Do custom logic here
        return super(FlavorCreateView, self).form_invalid(form)
```

Just as you can add logic to `form_valid()`, you can also add logic to `form_invalid()`.

You'll see an example of overriding both of these methods in [chapter 10](#), *More Things To Know About Forms*, [subsection 10.2.1](#), 'Form Data Is Saved to the Form, Then the Model Instance.'

Additional References:

- <http://pydanny.com/tag/class-based-views.html>
- www.python.org/download/releases/2.3/mro/

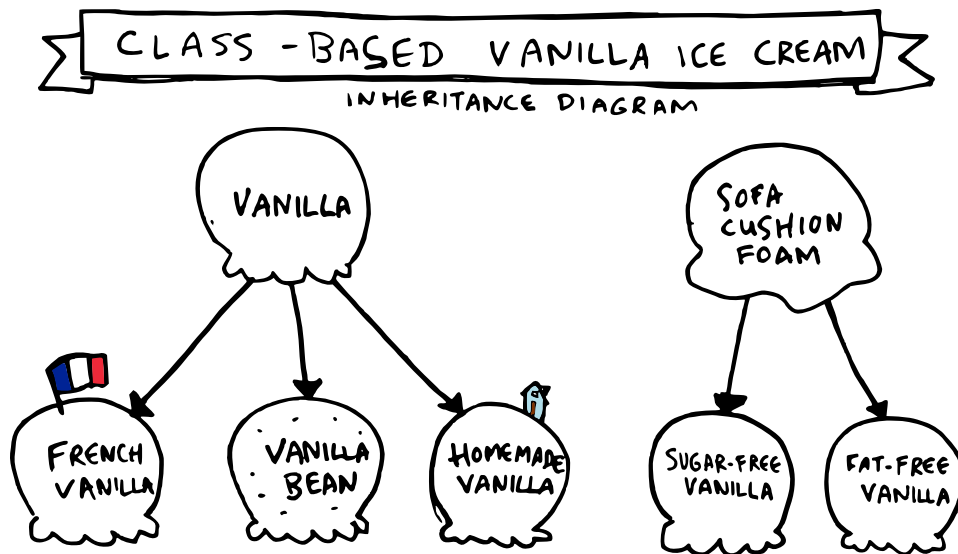


Figure 8.2: The other CBV: class-based vanilla ice cream.

8.4 How CBVs and Forms Fit Together

A common source of confusion with CBVs is their usage with Django forms.

Using our favorite example of the ice cream flavor tracking app, let's chart out a couple of examples of how form-related views might fit together.

First, let's define a flavor model to use in this section's view examples:

EXAMPLE 8.5

```
# flavors/models.py
from django.core.urlresolvers import reverse
from django.db import models

class Flavor(models.Model):
    title = models.CharField(max_length=255)
    slug = models.SlugField()
    scoops_remaining = models.IntegerField(default=0)

    def get_absolute_url(self):
        return reverse("flavor_detail", kwargs={"slug": self.slug})
```

Now, let's explore some common Django form scenarios that most Django users run into at one point or another.

8.4.1 Views + ModelForm Example

This is the simplest and most common Django form scenario. Typically when you create a model, you want to be able to add new records and update existing records that correspond to the model.

In this example, we'll show you how to construct a set of views that will create, update and display Flavor records. We'll also demonstrate how to provide confirmation of changes.

Here we have the following views:

- ❶ **FlavorCreateView** corresponds to a form for adding new flavors.
- ❷ **FlavorUpdateView** corresponds to a form for editing existing flavors.
- ❸ **FlavorDetailView** corresponds to the confirmation page for both flavor creation and flavor updates.

To visualize our views:

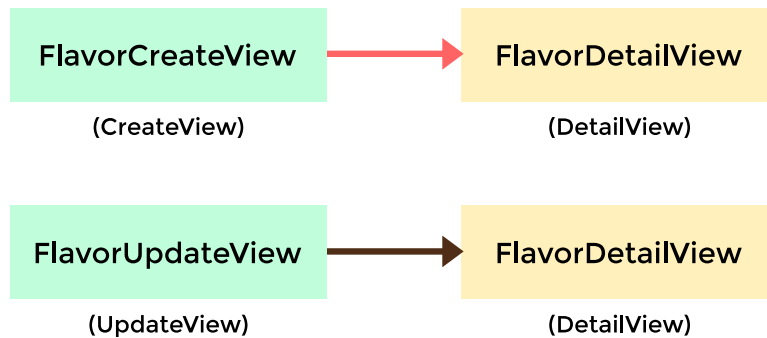


Figure 8.3: Views + ModelForm Flow

Note that we stick as closely as possible to Django naming conventions. `FlavorCreateView` subclasses Django's `CreateView`, `FlavorUpdateView` subclasses Django's `UpdateView`, and `FlavorDetailView` subclasses Django's `DetailView`.

Writing these views is easy, since it's mostly a matter of using what Django gives us:

EXAMPLE 8.6

```
# flavors/views.py
from django.views.generic import CreateView, UpdateView, DetailView

from braces.views import LoginRequiredMixin

from .models import Flavor

class FlavorCreateView(LoginRequiredMixin, CreateView):
    model = Flavor

class FlavorUpdateView(LoginRequiredMixin, UpdateView):
    model = Flavor

class FlavorDetailView(DetailView):
    model = Flavor
```

Simple at first glance, right? We accomplish so much with just a little bit of code!

But wait, there's a catch. If we wire these views into a *urls.py* module and create the necessary templates, we'll uncover a problem:

The FlavorDetailView is not a confirmation page.

For now, that statement is correct. Fortunately, we can fix it quickly with a few modifications to existing views and templates.

The first step in the fix is to use `django.contrib.messages` to inform the user visiting the `FlavorDetailView` that they just added or updated the flavor.

We'll need to override the `FlavorCreateView.form_valid()` and `FlavorUpdateView.form_valid()` methods. We can do this conveniently for both views with a `FlavorActionMixin`.

For the confirmation page fix, we change *flavors/views.py* to contain the following:

EXAMPLE 8.7

```
# flavors/views.py
from django.contrib import messages
from django.views.generic import CreateView, UpdateView, DetailView

from braces.views import LoginRequiredMixin

from .models import Flavor

class FlavorActionMixin(object):

    @property
    def action(self):
        msg = "{0} is missing action.".format(self.__class__)
        raise NotImplementedError(msg)

    def form_valid(self, form):
        msg = "Flavor {0}!".format(self.action)
        messages.info(self.request, msg)
        return super(FlavorActionMixin, self).form_valid(form)
```

```
class FlavorCreateView(LoginRequiredMixin, FlavorActionMixin,
                       CreateView):
    model = Flavor
    action = "created"

class FlavorUpdateView(LoginRequiredMixin, FlavorActionMixin,
                       UpdateView):
    model = Flavor
    action = "updated"

class FlavorDetailView(DetailView):
    model = Flavor
```

Earlier in this chapter, we covered a simpler example of how to override `form.valid()` within a CBV. Here, we reuse a similar `form.valid()` override method by creating a mixin to inherit from in multiple views.

Now we're using Django's **messages** framework to display confirmation messages to the user upon every successful add or edit. We define a `FlavorActionMixin` whose job is to queue up a confirmation message corresponding to the action performed in a view.

TIP: Mixins Should Inherit From Object

Please take notice that the `FlavorActionMixin` inherits from Python's `object` type rather than a pre-existing mixin or view. It's important that mixins have as shallow inheritance chain as possible. Simplicity is a virtue!

After a flavor is created or updated, a list of messages is passed to the context of the `FlavorDetailView`. We can see these messages if we add the following code to the views' template and then create or update a flavor:

EXAMPLE 8.8

```
{# templates/flavors/flavor_detail.html #}
{% if messages %}
```

```
<ul class="messages">
    {% for message in messages %}
    <li id="message_{{ forloop.counter }}"
        {% if message.tags %} class="{{ message.tags }}"
        {% endif %}>
        {{ message }}
    </li>
    {% endfor %}
</ul>
{% endif %}
```

TIP: Reuse the Messages Template Code!

It is common practice to put the above code into your project's base HTML template. Doing this allows message support for templates in your project.

To recap, this example demonstrated yet again how to override the `form_valid()` method, incorporate this into a mixin, how to incorporate multiple mixins into a view, and gave a quick introduction to the very useful `django.contrib.messages` framework.

8.4.2 Views + Form Example

Sometimes you want to use a Django `Form` rather than a `ModelForm`. Search forms are a particularly good use case for this, but you'll run into other scenarios where this is true as well.

In this example, we'll create a simple flavor search form. This involves creating a HTML form that doesn't modify any flavor data. The form's action will query the ORM, and the records found will be listed on a search results page.

Our intention is that when using our flavor search page, if users do a flavor search for "Dough", they should be sent to a page listing ice cream flavors like "Chocolate Chip Cookie Dough," "Fudge Brownie Dough," "Peanut Butter Cookie Dough," and other flavors containing the string "Dough" in their title. Mmm, we definitely want this feature in our web application.

There are more complex ways to implement this, but for our simple use case, all we need is a single view. We'll use a `FlavorListView` for both the search page and the search results page.

Here's an overview of our implementation:

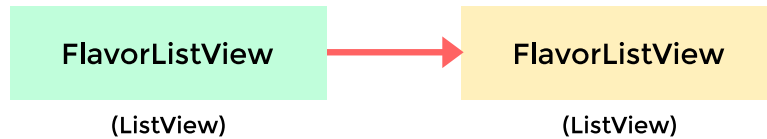


Figure 8.4: Views + Form Flow

In this scenario, we want to follow the standard internet convention for search pages, where 'q' is used for the search query parameter. We also want to accept a GET request rather than a POST request, which is unusual for forms but perfectly fine for this use case. Remember, this form doesn't add, edit, or delete objects, so we don't need a POST request here.

To return matching search results based on the search query, we need to modify the standard queryset supplied by the `ListView`. To do this, we override the `ListView`'s `get_queryset()` method. We add the following code to *flavors/views.py*:

EXAMPLE 8.9

```
from django.views.generic import ListView

from .models import Flavor

class FlavorListView(ListView):
    model = Flavor

    def get_queryset(self):
        # Fetch the queryset from the parent get_queryset
        queryset = super(FlavorListView, self).get_queryset()

        # Get the q GET parameter
        q = self.request.GET.get("q")
        if q:
            # Return a filtered queryset
            return queryset.filter(title__icontains=q)
```



```
# Return the base queryset
return queryset
```

Now, instead of listing all of the flavors, we list only the flavors whose titles contain the search string.

As we mentioned, search forms are unusual in that unlike nearly every other HTML form they specify a GET request in the HTML form. This is because search forms are not changing data, but simply retrieving information from the server. The search form should look something like this:

EXAMPLE 8.10

```
{# templates/flavors/_flavor_search.html #}
{% comment %}
    Usage: {% include "flavors/_flavor_search.html" %}
{% endcomment %}
<form action="{% url 'flavor_list' %}" method="GET">
    <input type="text" name="q" />
    <button type="submit">search</button>
</form>
```

TIP: Specify the Form Target in Search Forms

We also take care to specify the URL in the form action, because we've found that search forms are often included in several pages. This is why we prefix them with `'_'` and create them in such a way as to be included in other templates.

Once we get past overriding the `ListView's get_queryset()` method, the rest of this example is just a simple HTML form. We like this kind of simplicity.

8.5 Summary

This chapter covered:

- Using mixins with CBVs

- Which Django CBV should be used for which task
- General tips for CBV usage
- Connecting CBVs to forms

The next chapter explores common CBV/form patterns. Knowledge of these is helpful to have in your developer toolbox.

9 | Common Patterns for Forms

Django forms are powerful, flexible, extensible, and robust. For this reason, the Django admin and CBVs use them extensively. In fact, all the major Django API frameworks use `ModelForms` as part of their validation because of their powerful validation features.

Combining forms, models, and views allows us to get a lot of work done for little effort. The learning curve is worth it: once you learn to work fluently with these components, you'll find that Django provides the ability to create an amazing amount of useful, stable functionality at an amazing pace.

PACKAGE TIP: Useful Form-Related Packages

- **django-floppyforms** for rendering Django inputs in HTML5.
- **django-crispy-forms** for advanced form layout controls. By default, forms are rendered with Twitter Bootstrap form elements and styles. This package plays well with `django-floppyforms`, so they are often used together.
- **django-forms-bootstrap** is a simple tool for rendering Django forms using Twitter Bootstrap styles. This package plays well with `django-floppyforms` but conflicts with `django-crispy-forms`.

9.1 The Power of Django Forms

You might not be aware of the fact that even if your Django project uses an API framework and doesn't serve HTML, you are probably still using Django forms. Django forms are not just for web pages; their powerful validation features are useful on their own.

Interestingly enough, the design that Django’s API frameworks use is some form of class-based view. They might have their own implementation of CBVs (i.e. `django-tastypie`) or run off of Django’s own CBVs (`django-rest-framework`), but the use of inheritance and composition is a constant. We would like to think this is proof of the soundness of both Django forms and the concept of CBVs.

With that in mind, this chapter goes explicitly into one of the best parts of Django: forms, models, and CBVs working in concert. This chapter covers five common form patterns that should be in every Django developer’s toolbox.

9.2 Pattern 1: Simple ModelForm With Default Validators

The simplest data-changing form that we can make is a `ModelForm` using several default validators as-is, without modification. In fact, we already relied on default validators in [chapter 8](#), *Best Practices for Class-Based Views*, [subsection 8.4.1](#), “Views + `ModelForm` Example.”

If you recall, using `ModelForms` with CBVs to implement add/edit forms can be done in just a few lines of code:

EXAMPLE 9.1

```
# flavors/views.py
from django.views.generic import CreateView, UpdateView

from braces.views import LoginRequiredMixin

from .models import Flavor

class FlavorCreateView(LoginRequiredMixin, CreateView):
    model = Flavor

class FlavorUpdateView(LoginRequiredMixin, UpdateView):
    model = Flavor
```

To summarize how we use default validation as-is here:

- `FlavorCreateView` and `FlavorUpdateView` are assigned `Flavor` as their model.

- Both views auto-generate a `ModelForm` based on the `Flavor` model.
- Those `ModelForms` rely on the default field validation rules of the `Flavor` model.

Yes, Django gives us a lot of great defaults for data validation, but in practice, the defaults are never enough. We recognize this, so as a first step, the next pattern will demonstrate how to create a custom field validator.

9.3 Pattern 2: Custom Form Field Validators in ModelForms

What if we wanted to be certain that every use of the `title` field across our project's dessert apps started with the word 'Tasty'?

This is a string validation problem that can be solved with a simple **custom field validator**.

In this pattern, we cover how to create custom single-field validators and demonstrate how to add them to both abstract models and forms.

Imagine for the purpose of this example that we have a project with two different dessert-related models: a `Flavor` model for ice cream flavors, and a `Milkshake` model for different types of milkshakes. Assume that both of our example models have `title` fields.

To validate all editable model titles, we start by creating a *validators.py* module:

EXAMPLE 9.2

```
# core/validators.py
from django.core.exceptions import ValidationError

def validate_tasty(value):
    """ Raise a ValidationError if the
        value doesn't start with the
        word 'Tasty'
    """
    if not value.startswith(u"Tasty"):
        msg = u"Must start with Tasty"
        raise ValidationError(msg)
```

In Django, a custom field validator is simply a function that raises an error if the submitted argument doesn't pass its test.

Of course, while our `validate_tasty()` validator function just does a simple string check for the sake of example, it's good to keep in mind that form field validators can become quite complex in practice.

TIP: Test Your Validators Carefully

Since validators are critical in keeping corruption out of Django project databases, it's especially important to write detailed tests for them.

These tests should include thoughtful edge case tests for every condition related to your validators' custom logic.

In order to use our `validate_tasty()` validator function across different dessert models, we're going to first add it to an abstract model called `TastyTitleAbstractModel`, which we plan to use across our project.

Assuming that our `Flavor` and `Milkshake` models are in separate apps, it doesn't make sense to put our validator in one app or the other. Instead, we create a *core/models.py* module and place the `TastyTitleAbstractModel` there.

EXAMPLE 9.3

```
# core/models.py
from django.db import models

from .validators import validate_tasty

class TastyTitleAbstractModel(models.Model):

    title = models.CharField(max_length=255, validators=[validate_tasty])

    class Meta:
        abstract = True
```

The last two lines of the above example code for *core/models.py* make `TastyTitleAbstractModel` an abstract model, which is what we want.

Let's alter the original *flavors/models.py* Flavor code to use `TastyTitleAbstractModel` as the parent class:

EXAMPLE 9.4

```
# flavors/models.py
from django.core.urlresolvers import reverse
from django.db import models

from core.models import TastyTitleAbstractModel

class Flavor(TastyTitleAbstractModel):
    slug = models.SlugField()
    scoops_remaining = models.IntegerField(default=0)

    def get_absolute_url(self):
        return reverse("flavor_detail", kwargs={"slug": self.slug})
```

This works with the `Flavor` model, and it will work with any other tasty food-based model such as a `WaffleCone` or `Cake` model. Any model that inherits from the `TastyTitleAbstractModel` class will throw a validation error if anyone attempts to save a model with a title that doesn't start with 'Tasty'.

Now, let's explore a couple of questions that might be forming in your head:

- What if we wanted to use `validate_tasty()` in just forms?
- What if we wanted to assign it to other fields besides the title?

To support these behaviors, we need to create a custom `FlavorForm` that utilizes our custom field validator:

EXAMPLE 9.5

```
# flavors/forms.py
from django import forms
```

```
from core.validators import validate_delicious
from .models import Flavor

class FlavorForm(forms.ModelForm):
    def __init__(self, *args, **kwargs):
        super(FlavorForm, self).__init__(*args, **kwargs)
        self.fields["title"].validators.append(validate_delicious)
        self.fields["slug"].validators.append(validate_delicious)

    class Meta:
        model = Flavor
```

A nice thing about both examples of validator usage in this pattern is that we haven't had to change the `validate_tasty()` code at all. Instead, we just import and use it in new places.

Attaching the custom form to the views is our next step. The default behavior of Django model-based edit views is to auto-generate the `ModelForm` based on the view's `model` attribute. We are going to override that default and pass in our custom `FlavorForm`. This occurs in the *flavors/views.py* module, where we alter the create and update forms as demonstrated below:

```
EXAMPLE 9.6
# flavors/views.py
from django.contrib import messages
from django.views.generic import CreateView, UpdateView, DetailView

from braces.views import LoginRequiredMixin

from .models import Flavor
from .forms import FlavorForm

class FlavorActionMixin(object):

    @property
    def action(self):
        msg = "{0} is missing action.".format(self.__class__)
        raise NotImplementedError(msg)
```



```
def form_valid(self, form):
    msg = "Flavor {0}!".format(self.action)
    messages.info(self.request, msg)
    return super(FlavorActionMixin, self).form_valid(form)

class FlavorCreateView(LoginRequiredMixin, FlavorActionMixin,
                       CreateView):
    model = Flavor
    action = "created"
    # Explicitly attach the FlavorForm class
    form_class = FlavorForm

class FlavorUpdateView(LoginRequiredMixin, FlavorActionMixin,
                       UpdateView):
    model = Flavor
    action = "updated"
    # Explicitly attach the FlavorForm class
    form_class = FlavorForm

class FlavorDetailView(DetailView):
    model = Flavor
```

The `FlavorCreateView` and `FlavorUpdateView` views now use the new `FlavorForm` to validate incoming data.

Note that with these modifications, the `Flavor` model can either be identical to the one at the start of this chapter, or it can be an altered one that inherits from `TastyTitleAbstractModel`.

9.4 Pattern 3: Overriding the Clean Stage of Validation

Let's discuss some interesting validation use cases:

- Multi-field validation
- Validation involving existing data from the database that has already been validated

Both of these are great scenarios for overriding the `clean()` and `clean_<field.name>()` methods with custom validation logic.

After the default and custom field validators are run, Django provides a second stage and process for validating incoming data, this time via the `clean()` method and `clean_<field.name>()` methods. You might wonder why Django provides more hooks for validation, so here are our two favorite arguments:

- ❶ The `clean()` method is the place to validate two or more fields against each other, since it's not specific to any one particular field.
- ❷ The clean validation stage is a better place to attach validation against persistent data. Since the data already has some validation, you won't waste as many database cycles on needless queries.

Let's explore this with another validation example. Perhaps we want to implement an ice cream ordering form, where users could specify the flavor desired, add toppings, and then come to our store and pick them up.

Since we want to prevent users from ordering flavors that are out of stock, we'll put in a `clean_slug()` method. With our flavor validation, our form might look like:

EXAMPLE 9.7

```
# flavors/forms.py
from django import forms
from flavors.models import Flavor

class IceCreamOrderForm(forms.Form):
    """ Normally done with forms.ModelForm. But we use forms.Form here
        to demonstrate that these sorts of techniques work on every
        type of form.
    """

    slug = forms.ChoiceField("Flavor")
    toppings = forms.CharField()

    def __init__(self, *args, **kwargs):
        super(IceCreamOrderForm, self).__init__(*args,
                                                **kwargs)
        # We dynamically set the choices here rather than
```

```
# in the flavor field definition. Setting them in
# the field definition means status updates won't
# be reflected in the form without server restarts.
self.fields["slug"].choices = [
    (x.slug, x.title) for x in Flavor.objects.all()
]
# NOTE: We could filter by whether or not a flavor
#       has any scoops, but this is an example of
#       how to use clean_slug, not filter().

def clean_slug(self):
    slug = self.cleaned_data["slug"]
    if Flavor.objects.get(slug=slug).scoops_remaining <= 0:
        msg = u"Sorry, we are out of that flavor."
        raise forms.ValidationError(msg)
    return slug
```

For HTML-powered views, the `clean_slug()` method in our example, upon throwing an error, will attach a “Sorry, we are out of that flavor” message to the flavor HTML input field. This is a great shortcut for writing HTML forms!

Now imagine if we get common customer complaints about orders with too much chocolate. Yes, it’s silly and quite impossible, but we’re just using ‘too much chocolate’ as a completely mythical example for the sake of making a point.

In any case, let’s use the `clean()` method to validate the flavor and toppings fields against each other.

EXAMPLE 9.8

```
# attach this code to the previous example (9.13)
def clean(self):
    cleaned_data = super(IceCreamOrderForm, self).clean()
    slug = cleaned_data.get("slug", "")
    toppings = cleaned_data.get("toppings", "")

    # Silly "too much chocolate" validation example
```

```
if u"chocolate" in slug.lower() and \
    u"chocolate" in toppings.lower():
    msg = u"Your order has too much chocolate."
    raise forms.ValidationError(msg)
return cleaned_data
```

There we go, an implementation against the impossible condition of too much chocolate!

9.5 Pattern 4: Hacking Form Fields (2 CBVs, 2 Forms, 1 Model)

This is where we start to get fancy. We're going to cover a situation where two views/forms correspond to one model. We'll hack Django forms to produce a form with custom behavior.

It's not uncommon to have users create a record that contains a few empty fields which need additional data later. An example might be a list of stores, where we want each store entered into the system as fast as possible, but want to add more data such as phone number and description later. Here's our `IceCreamStore` model:

EXAMPLE 9.9

```
# stores/models.py
from django.core.urlresolvers import reverse
from django.db import models

class IceCreamStore(models.Model):
    title = models.CharField(max_length=100)
    block_address = models.TextField()
    phone = models.CharField(max_length=20, blank=True)
    description = models.TextField(blank=True)

    def get_absolute_url(self):
        return reverse("store_detail", kwargs={"pk": self.pk})
```

The default `ModelForm` for this model forces the user to enter the `title` and `block_address` field but allows the user to skip the `phone` and `description` fields. That's great for initial data entry, but

as mentioned earlier, we want to have future updates of the data to require the phone and description fields.

The way we implemented this in the past before we began to delve into their construction was to override the phone and description fields in the edit form. This resulted in heavily-duplicated code that looked like this:

```
BAD EXAMPLE 9.1
# stores/forms.py
from django import forms

from .models import IceCreamStore

class IceCreamStoreUpdateForm(forms.ModelForm):
    # Don't do this! Duplication of the model field!
    phone = forms.CharField(required=True)
    # Don't do this! Duplication of the model field!
    description = forms.TextField(required=True)

    class Meta:
        model = IceCreamStore
```

This form should look very familiar. Why is that?

Well, we're nearly copying the `IceCreamStore` model!

This is just a simple example, but when dealing with a lot of fields on a model, the duplication becomes extremely challenging to manage. In fact, what tends to happen is copy-pasting of code from models right into forms, which is a gross violation of **Don't Repeat Yourself**.

Want to know how gross? Using the above approach, if we add a simple `help_text` attribute to the `description` field in the model, it will not show up in the template until we also modify the `description` field definition in the form. If that sounds confusing, that's because it is.

A better way is to rely on a useful little detail that's good to remember about Django forms: instantiated form objects store fields in a dict-like attribute called `fields`.

Instead of copy-pasting field definitions from models to forms, we can simply apply new attributes to each field in the `__init__()` method of the `ModelForm`:

```
EXAMPLE 9.10

# stores/forms.py
# Call phone and description from the self.fields dict-like object
from django import forms

from .models import IceCreamStore

class IceCreamStoreUpdateForm(forms.ModelForm):

    class Meta:
        model = IceCreamStore

    def __init__(self, *args, **kwargs):
        # Call the original __init__ method before assigning
        # field overloads
        super(IceCreamStoreUpdateForm, self).__init__(*args,
                                                       **kwargs)
        self.fields["phone"].required = True
        self.fields["description"].required = True
```

This improved approach allows us to stop copy-pasting code and instead focus on just the field-specific settings.

An important point to remember is that when it comes down to it, Django forms are just Python classes. They get instantiated as objects, they can inherit from other classes, and they can act as superclasses.

Therefore, we can rely on inheritance to trim the line count in our ice cream store forms:

```
EXAMPLE 9.11

# stores/forms.py
from django import forms

from .models import IceCreamStore
```

```

class IceCreamStoreCreateForm(forms.ModelForm):

    class Meta:
        model = IceCreamStore
        fields = ("title", "block_address", )

class IceCreamStoreUpdateForm(IceCreamStoreCreateForm):

    def __init__(self, *args, **kwargs):
        super(IceCreamStoreUpdateForm,
              self).__init__(*args, **kwargs)
        self.fields["phone"].required = True
        self.fields["description"].required = True

    class Meta(IceCreamStoreCreateForm.Meta):
        # show all the fields!
        fields = ("title", "block_address", "phone",
                  "description", )

```

WARNING: Use Meta.fields and Never Use Meta.exclude

We use `Meta.fields` instead of `Meta.exclude` so that we know exactly what fields we are exposing. See [chapter 21](#), *Security Best Practices*, [section 21.12](#), ‘Don’t use `ModelForms.Meta.exclude`’.

Finally, now we have what we need to define the corresponding CBVs. We’ve got our form classes, so let’s use them in the `IceCreamStore` create and update views:

```

EXAMPLE 9.12

# stores/views
from django.views.generic import CreateView, UpdateView

from .forms import IceCreamStoreCreateForm
from .forms import IceCreamStoreUpdateForm
from .models import IceCreamStore

```

```
class IceCreamCreateView(CreateView):
    model = IceCreamStore
    form_class = IceCreamStoreCreateForm

class IceCreamUpdateView(UpdateView):
    model = IceCreamStore
    form_class = IceCreamStoreUpdateForm
```

We now have two views and two forms that work with one model.

9.6 Pattern 5: Reusable Search Mixin View

In this example, we're going to cover how to reuse a search form in two views that correspond to two different models.

Assume that both models have a field called `title` (this pattern also demonstrates why naming standards in projects is a good thing). This example will demonstrate how a single CBV can be used to provide simple search functionality on both the `Flavor` and `IceCreamStore` models.

We'll start by creating a simple search mixin for our view:

```
EXAMPLE 9.13
# core/views.py
class TitleSearchMixin(object):

    def get_queryset(self):
        # Fetch the queryset from the parent's get_queryset
        queryset = super(TitleSearchMixin, self).get_queryset()

        # Get the q GET parameter
        q = self.request.GET.get("q")
        if q:
            # return a filtered queryset
            return queryset.filter(title__icontains=q)
```



```
# No q is specified so we return queryset
return queryset
```

The above code should look very familiar as we used it almost verbatim in the Forms + View example. Here's how you make it work with both the Flavor and IceCreamStore views. First the flavor views:

```
EXAMPLE 9.14
# add to flavors/views.py
from django.views.generic import ListView

from core.views import TitleSearchMixin
from .models import Flavor

class FlavorListView(TitleSearchMixin, ListView):
    model = Flavor
```

And we'll add it to the ice cream store views:

```
EXAMPLE 9.15
# add to stores/views.py
from django.views.generic import ListView

from core.views import TitleSearchMixin
from .models import Store

class IceCreamStoreListView(TitleSearchMixin, ListView):
    model = Store
```

As for the form? We just define it in HTML for each ListView:

```
EXAMPLE 9.16
{# form to go into stores/store_list.html template #}
<form action="" method="GET">
```

```
<input type="text" name="q" />
<button type="submit">search</button>
</form>
```

and

EXAMPLE 9.17

```
{# form to go into flavors/flavor_list.html template #}
<form action="" method="GET">
  <input type="text" name="q" />
  <button type="submit">search</button>
</form>
```

Now we have the same mixin in both views. Mixins are a good way to reuse code, but using too many mixins in a single class makes for very hard-to-maintain code. As always, try to keep your code as simple as possible.

9.7 Summary

We began this chapter with the simplest form pattern, using a `ModelForm`, `CBV`, and default validators. We iterated on that with an example of a custom validator.

Next, we explored more complex validation. We covered an example overriding the `clean` methods. We also closely examined a scenario involving two views and their corresponding forms that were tied to a single model.

Finally, we covered an example of creating a reusable search mixin to add the same form to two different apps.

10 | More Things to Know About Forms

95% of Django projects should use ModelForms.

91% of all Django projects use ModelForms.

80% of ModelForms require trivial logic.

20% of ModelForms require complicated logic.

– pydanny made-up statistics™

Django's forms are really powerful, but there are edge cases that can cause a bit of anguish.

If you understand the structure of how forms are composed and how to call them, most edge cases can be readily overcome.

10.1 Use the POST Method in HTML Forms

Every HTML form that alters data must submit its data via the POST method:

EXAMPLE 10.1

```
<form action="{% url 'flavor_add' %}" method="post">
```

The only exception you'll ever see to using POST in forms is with search forms, which typically submit queries that don't result in any alteration of data. Search forms that are idempotent should use the GET method.

10.1.1 Don't Disable Django's CSRF Protection

This is covered in [chapter 21](#), *Security Best Practices*, [section 21.7](#), ‘Always Use CSRF Protection With Forms That Modify Data.’ Also, please familiarize yourself with Django’s documentation on the subject: <https://docs.djangoproject.com/en/1.5/ref/contrib/csrf/>

10.2 Know How Form Validation Works

Form validation is one of those areas of Django where knowing the inner workings will drastically improve your code. Let’s take a moment to dig into form validation and cover some of the key points.

When you call `form.is_valid()`, a lot of things happen behind the scenes. The following things occur according to this workflow:

- ❶ If the form has bound data, `form.is_valid()` calls the `form.full_clean()` method.
- ❷ `form.full_clean()` iterates through the form fields and each field validates itself:
 - ❶ Data coming into the field is coerced into Python via the `to_python()` method or raises a `ValidationError`.
 - ❷ Data is validated against field-specific rules, including custom validators. Failure raises a `ValidationError`.
 - ❸ If there are any custom `clean_<field>()` methods in the form, they are called at this time.
- ❸ `form.full_clean()` executes the `form.clean()` method.
- ❹ If it’s a `ModelForm` instance, `form.post_clean()` does the following:
 - ❶ Sets `ModelForm` data to the `Model` instance, regardless of whether `form.is_valid()` is `True` or `False`.
 - ❷ Calls the model’s `clean()` method. For reference, saving a model instance through the ORM does not call the model’s `clean()` method.

If this seems complicated, just remember that it gets simpler in practice, and that all of this functionality lets us really understand what’s going on with incoming data. The example in the next section should help to explain this further.

10.2.1 Form Data Is Saved to the Form, Then the Model Instance

We like to call this the *WHAT?!?* of form validation. At first glance, form data being set to the form instance might seem like a bug. But it's not a bug. It's intended behavior.

In a `ModelForm`, form data is saved in two distinct steps:

- ❶ First, form data is saved to the form instance.
- ❷ Later, form data is saved to the model instance.

Since `ModelForms` don't save to the model instance until they are activated by the `form.save()` method, we can take advantage of this separation as a useful feature.

For example, perhaps you need to catch the details of failed submission attempts for a form, saving both the user-supplied form data as well as the intended model instance changes.

A simple, perhaps simplistic, way of capturing that data is as follows. First, we create a form failure history model in *core/models.py*:

EXAMPLE 10.2

```
# core/models.py
from django.db import models

class ModelFormFailureHistory(models.Model):
    form_data = models.TextField()
    model_data = models.TextField()
```

Second, we add the following to the `FlavorActionMixin` in *flavors/views.py*:

EXAMPLE 10.3

```
# flavors/models.py
import json

from django.contrib import messages
from django.core import serializers

from core.models import ModelFormFailureHistory
```

```
class FlavorActionMixin(object):

    @property
    def action(self):
        msg = "{0} is missing action.".format(self.__class__)
        raise NotImplementedError(msg)

    def form_valid(self, form):
        msg = "Flavor {0}!".format(self.action)
        messages.info(self.request, msg)
        return super(FlavorActionMixin, self).form_valid(form)

    def form_invalid(self, form):
        """ Save invalid form and model data for later reference """
        form_data = json.dumps(form.cleaned_data)
        model_data = serializers.serialize("json",
                                           [form.instance])[1:-1]
        ModelFormFailureHistory.objects.create(
            form_data=form_data,
            model_data=model_data
        )
        return super(FlavorActionMixin,
                     self).form_invalid(form)
```

If you recall, `form_invalid()` is called after failed validation of a form with bad data. When it is called here in this example, both the cleaned form data and the final data saved to the database are saved as a `ModelFormFailureHistory` record.

10.3 Summary

Once you dig into forms, keep yourself focused on clarity of code and testability. Forms are one of the primary validation tools in your Django project, an important defense against attacks and accidental data corruption.

11 | Building REST APIs in Django

Today's internet is much more than HTML-powered websites. Developers need to support AJAX and the mobile web. Having tools that support easy creation of **JSON**, **YAML**, **XML**, and other formats is important. By design, a **Representational State Transfer (REST) Application Programming Interface (API)** exposes application data to other concerns.

PACKAGE TIP: Packages For Crafting APIs

- **django-rest-framework** builds off of Django CBVs, adding a wonderful browsable API feature. It has a lot of features, follows elegant patterns, and is great to work with.
- **django-tastypie** is a more mature API framework that implements its own class-based view system. It's a feature-rich, mature, powerful, stable tool for creating APIs from Django models. It was created by Daniel Lindsley, the developer also behind **django-haystack**, the most commonly used Django search library.
- **django-braces** *can* be used in direct conjunction with Django CBVs to create super-quick, super-simple one-off REST API views. The downside is that when you get into the full range of HTTP methods such as **PUT**, it rapidly becomes a hindrance.

11.1 Fundamentals of Basic REST API Design

The Hypertext Transfer Protocol (HTTP) is a protocol for distributing content that provides a set of methods to declare actions. By convention, REST APIs rely on these methods, so use the appropriate HTTP method for each type of action:

Purpose of Request	HTTP Method	Rough SQL equivalent
--------------------	-------------	----------------------

Create a new resource	POST	INSERT
Read an existing resource	GET	SELECT
Request the header of an existing resource	HEAD	
Update an existing resource	PUT	UPDATE
Update part of an existing resource	PATCH	UPDATE
Delete an existing resource	DELETE	DELETE
Return the supported HTTP methods for the given URL	OPTIONS	
Echo back the request	TRACE	
Tunneling over TCP/IP (usually not implemented)	CONNECT	

Table 11.1: HTTP Methods

A couple of notes on the above:

- If you're implementing a read-only API, you might only need to implement GET methods.
- If you're implementing a read-write API you must at least also use POST, but should also consider using PUT and DELETE.
- By definition, GET, PUT, and DELETE are idempotent. POST and PATCH are not.
- PATCH is often not implemented, but it's a good idea to implement it if your API supports PUT requests.

Here are some common HTTP status codes that you should consider supporting when implementing your REST API. Note that this is a partial list; a much longer list of status codes can be found at http://en.wikipedia.org/wiki/List_of_HTTP_status_codes.

HTTP Status Code	Success/Failure	Meaning
200 OK	Success	GET - Return resource PUT - Provide status message or return resource
201 Created	Success	POST - Provide status message or return newly created resource
204 No Content	Success	DELETE

HTTP Status Code	Success/Failure	Meaning
304 Unchanged	Redirect	ANY - Indicates no changes since the last request. Used for checking Last-Modified and Etag headers to improve performance.
400 Bad Request	Failure	PUT, POST - Return error messages, including form validation errors.
401 Unauthorized	Failure	ALL - Authentication required but user did not provide credentials.
403 Forbidden	Failure	ALL - User attempted to access restricted content
404 Not Found	Failure	ALL - Resource is not found
405 Method Not Allowed	Failure	ALL - An invalid HTTP method was attempted.

Table 11.2: HTTP Status Codes

11.2 Implementing a Simple JSON API

Let's use the *flavors* app example from previous chapters as our base, providing the capability to create, read, update, and delete flavors via HTTP requests using AJAX, python-requests, or some other library. We'll also use **django-rest-framework**, as it provides us with the capability to build a REST API quickly using patterns similar to the class-based views we describe in previous chapters. We'll begin by listing the Flavor model again:

EXAMPLE 11.1

```
# flavors/models.py
from django.core.urlresolvers import reverse
from django.db import models

class Flavor(models.Model):
    title = models.CharField(max_length=255)
    slug = models.SlugField()
    scoops_remaining = models.IntegerField(default=0)

    def get_absolute_url(self):
        return reverse("flavor_detail", kwargs={"slug": self.slug})
```

Now let's add in some views:

EXAMPLE 11.2

```
from rest_framework.generics import (
    ListCreateAPIView,
    RetrieveUpdateDestroyAPIView
)

from .models import Flavor

class FlavorCreateReadView(ListCreateAPIView):
    model = Flavor

class FlavorReadUpdateDeleteView(RetrieveUpdateDestroyAPIView):
    model = Flavor
```

We're done! Wow, that was fast!

WARNING: Our Simple API Does Not Use Permissions

If you implement an API using our example, don't forget to authenticate users and assign them permissions appropriately!

Now we'll wire this into our *flavors/urls.py* module:

EXAMPLE 11.3

```
# flavors/urls.py
from django.conf.urls.defaults import patterns, url

from flavors import views

urlpatterns = patterns("",
    url(
        regex=r"^api/$",
        view=views.FlavorCreateReadView.as_view(),
        name="flavor_rest_api"
    ),
```

```

url(
    regex=r"^api/(?P<slug>[-\w]+)/$",
    view=views.FlavorReadUpdateDeleteView.as_view(),
    name="flavor_rest_api"
)

```

What we are doing is reusing the same view and URLConf name, making it easier to manage when you have a need for a JavaScript-heavy front-end. All you need to do is access the Flavor resource via the `{% url %}` template tag. In case it's not clear exactly what our URLConf is doing, let's review it with a table:

Url	View	Url Name (same)
/flavors/api/	FlavorCreateReadView	flavor_rest_api
/flavors/api/:slug/	FlavorReadUpdateDeleteView	flavor_rest_api

Table 11.3: URLConf for the Flavor REST APIs

The end result is the traditional REST-style API definition:

EXAMPLE 11.4

```

flavors/api/
flavors/api/:slug/

```

We've shown you (if you didn't know already) how it's very easy to build REST APIs in Django, now let's go over some advice on maintaining and extending them.

11.3 REST API Architecture

Building quick APIs is easy with tools like `django-rest-framework` and `django-tastypie`, but extending and maintaining them to match your project's needs takes a bit more thought.

11.3.1 Code for an App Should Remain in the App

When it comes down to it, REST APIs are just views. In our opinion, REST API views should go into `views.py` modules and follow the same guidelines we endorse when it comes to any other view. The same goes for app or model specific serializers and renderers. If you do have app specific serializers or renderers the same applies.

11.3.2 Try to Keep Business Logic Out of API Views

It's a good idea to try to keep as much logic as possible out of API views. If this sounds familiar, it should. We covered this in 'Try to Keep Business Logic out of Views', [chapter 7](#) *Function- and Class-Based Views*, and API views are just another type of view, after all.

11.3.3 Grouping API URLs

If you have REST API views in multiple Django apps, how do you build a project-wide API that looks like this?

EXAMPLE 11.5

```
api/flavors/ # GET, POST
api/flavors/:slug/ # GET, PUT, DELETE
api/users/ # GET, POST
api/users/:slug/ # GET, PUT, DELETE
```

In the past, we placed all API view code into a dedicated Django app called *api* or *apiv1*, with custom logic in some of the REST views, serializers, and more. In theory it's a pretty good approach, but in practice it means we have logic for a particular app in more than just one location.

Our current approach is to lean on URL configuration. When building a project-wide API we write the REST views in the *views.py* modules, wire them into a URLConf called something like *core/api.py* or *core/apiv1.py* and include that from the project root's *urls.py* module. This means that we might have something like the following code:

EXAMPLE 11.6

```
# core/api.py
""" Called from the project root's urls.py URLConf thus:
    url(r"^api/", include("core.api"), namespace="api"),
"""
from django.conf.urls.defaults import patterns, url

from flavors import views as flavor_views
from users import views as user_views

urlpatterns = patterns("",
    # {% url "api:flavors" %}
    url(
        regex=r"^flavors/$",
        view=flavor_views.FlavorCreateReadView.as_view(),
        name="flavors"
    ),
    # {% url "api:flavors" flavor.slug %}
    url(
        regex=r"^flavors/(?P<slug>[-\w]+)/$",
        view=flavor_views.FlavorReadUpdateDeleteView.as_view(),
        name="flavors"
    ),
    # {% url "api:users" %}
    url(
        regex=r"^users/$",
        view=user_views.UserCreateReadView.as_view(),
        name="users"
    ),
    # {% url "api:users" user.slug %}
    url(
        regex=r"^users/(?P<slug>[-\w]+)/$",
        view=user_views.UserReadUpdateDeleteView.as_view(),
        name="users"
    ),
)
```

11.3.4 Test Your API

We find that Django's test suite makes it really easy to test API implementations. It's certainly much easier than staring at `curl` results! Testing is covered at length in [chapter 18](#), *Testing Stinks and Is a Waste of Money!*, and we even include in that chapter the tests we wrote for our simple JSON API (see [subsection 18.3.1](#)).

11.4 AJAX and the CSRF Token

If you use AJAX with Django, you may discover that triggering the CSRF token validation blocks your ability to use your API.

11.4.1 Posting Data via AJAX

Django's CSRF protection seems like an inconvenience when writing AJAX. However, if you're using **jQuery** then you can just create a *csrf.js* and use the following on any page with AJAX that is updating data.

EXAMPLE 11.7

```
// Place at /static/js/csrf.js
// using jQuery
function csrfSafeMethod(method) {
    // These HTTP methods do not require CSRF protection
    return (/^(GET|HEAD|OPTIONS|TRACE)$/.test(method));
}
$.ajaxSetup({
    crossDomain: false, // Obviates need for sameOrigin test
    beforeSend: function(xhr, settings) {
        if (!csrfSafeMethod(settings.type)) {
            xhr.setRequestHeader("X-CSRFToken", csrftoken);
        }
    }
});
```

WARNING: This Works Only for jQuery 1.5+

This JavaScript will not work with versions of jQuery before 1.5. Please read the CSRF documentation specific for other versions of Django.

Now let's include the JavaScript on a page which has a ice cream shopping cart form for ordering ice cream:

EXAMPLE 11.8

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

{% block title %}Ice Cream Shopping Cart{% endblock %}

{% block content %}
    <h1>Ice Cream Shopping Cart</h1>
    <div class="shopping-cart"></div>
{% endblock %}

{% block javascript %}
    {{ block.super }}
    <script type="text/javascript"
        src="{% static "js/csrf.js" %}"></script>
    <script type="text/javascript"
        src="{% static "js/shopping_cart.js" %}"></script>
{% endblock %}
```

Recommended reading:

- <https://docs.djangoproject.com/en/1.5/ref/contrib/csrf/>

11.5 Additional Reading

We highly recommend reading the following:

- <http://en.wikipedia.org/wiki/REST>

- http://en.wikipedia.org/wiki/List_of_HTTP_status_codes
- <http://jacobian.org/writing/rest-worst-practices/>
- <http://paltman.com/2012/04/30/integration-backbonejs-tastypie/>

11.6 Summary

In this chapter we covered:

- API Creation Libraries.
- Grouping Strategies
- Fundamentals of Basic REST API Design.
- Implementing a Simple JSON API.
- AJAX and CSRF tokens.

In the next chapter we'll switch back to HTML rendering via templates.

12 | Templates: Best Practices

One of Django's early design decisions was to limit the functionality of the template language. This heavily constrains what can be done with Django templates, which is actually a very good thing since it forces us to keep business logic in the Python side of things.

Think about it: the limitations of Django templates force us to put the most critical, complex and detailed parts of our project into `.py` files rather than into template files. Python happens to be one of the most clear, concise, elegant programming languages of the planet, so why would we want things any other way?

12.1 Follow a Minimalist Approach

We recommend taking a minimalist approach to your template code. Treat the so-called limitations of Django templates as a blessing in disguise. Use those constraints as inspiration to find simple, elegant ways to put more of your business logic into Python code rather than into templates.

Taking a minimalist approach to templates also makes it much easier to adapt your Django apps to changing format types. When your templates are bulky and full of nested looping, complex conditionals, and data processing, it becomes harder to reuse business logic code in templates, not to mention impossible to use the same business logic in template-less views such as API views. Structuring your Django apps for code reuse is especially important as we move forward into the era of increased API development, since APIs and web pages often need to expose identical data with different formatting.

To this day, HTML remains a standard expression of content, and therein lies the practices and patterns for this chapter.

12.2 Template Architecture Patterns

We've found that for our purposes, simple 2-tier or 3-tier template architectures are ideal. The difference in tiers is how many levels of template extending needs to occur before content in apps is displayed. See the examples below:

12.2.1 2-Tier Template Architecture Example

With a 2-tier template architecture, all templates inherit from a single root *base.html* file.

```
EXAMPLE 12.1
templates/
  base.html
  dashboard.html # extends base.html
profiles/
  profile_detail.html # extends base.html
  profile_form.html # extends base.html
```

This is best for sites with a consistent overall layout from app to app.

12.2.2 3-Tier Template Architecture Example

With a 3-tier template architecture:

- Each app has a *base-<app_name>.html* template. App-level base templates share a common parent *base.html* template.
- Templates within apps share a common parent *base-<app_name>.html* template.
- Any template at the same level as *base.html* inherits *base.html*.

```
EXAMPLE 12.2
templates/
  base.html
  dashboard.html # extends base.html
profiles/
```

```
base_profiles.html # extends base.html
profile_detail.html # extends base_profile.html
profile_form.html # extends base_profile.html
```

The 3-tier architecture is best for websites where each section requires a distinctive layout. For example, a news site might have a local news section, a classified ads section, and an events section. Each of these sections requires its own custom layout.

This is extremely useful when we want HTML to look or behave differently for a particular section of the site that groups functionality.

12.2.3 Flat Is Better Than Nested



Figure 12.1: An excerpt from the Zen of Ice Cream.

Complex template hierarchies make it exceedingly difficult to debug, modify, and extend HTML pages and tie in CSS styles. When template block layouts become unnecessarily nested, you end up digging through file after file just to change, say, the width of a box.

Giving your template blocks as shallow an inheritance structure as possible will make your templates easier to work with and more maintainable. If you're working with a designer, your designer will thank you.

That being said, there's a difference between excessively-complex template block hierarchies and templates that use blocks wisely for code reuse. When you have large, multi-line chunks of the same or very similar code in separate templates, refactoring that code into reusable blocks will make your code more maintainable.

The *Zen of Python* includes the aphorism “*Flat is better than nested*” for good reason. Each level of nesting adds mental overhead. Keep that in mind when architecting your Django templates.

TIP: The Zen of Python

At the command line, do the following:

```
python -c `import this`
```

What you'll see is the *Zen of Python*, an eloquently-expressed set of guiding principles for the design of the Python programming language.

12.3 Limit Processing in Templates

The less processing you try to do in your templates, the better. This is particularly a problem when it comes to queries and iteration performed in the template layer.

Whenever you iterate over a queryset in a template, ask yourself the following questions:

- ❶ How large is the queryset? Looping over gigantic querysets in your templates is almost always a bad idea.
- ❷ How large are the objects being retrieved? Are all the fields needed in this template?
- ❸ During each iteration of the loop, how much processing occurs?

If any warning bells go off in your head, then there's probably a better way to rewrite your template code.

WARNING: Why Not Just Cache?

Sometimes you can just cache away your template inefficiencies. That's fine, but before you cache, you should first try to attack the root of the problem.

You can save yourself a lot of work by mentally tracing through your template code, doing some quick run time analysis, and refactoring.

Let's now explore some examples of template code that can be rewritten more efficiently.

Suspend your disbelief for a moment and pretend that the nutty duo behind Two Scoops ran a 30-second commercial during the Superbowl. "Free pints of ice cream for the first million developers who request them! All you have to do is fill out a form to get a voucher redeemable in stores!"

Naturally, we have a "vouchers" app to track the names and email addresses of everyone who requested a free pint voucher. Here's what the model for this app looks like:

EXAMPLE 12.3

```
from django.core.urlresolvers import reverse
from django.db import models
from .managers import VoucherManager

class Voucher(models.Model):
    """ Vouchers for free pints of ice cream """
    name = models.CharField(max_length=100)
    email = models.EmailField()
    address = models.TextField()
    birth_date = models.DateField(blank=True)
    sent = models.BooleanField(default=False)
    redeemed = models.BooleanField(default=False)

    objects = VoucherManager()
```

This model will be used in the following examples to illustrate a few "gotchas" that you should avoid.

12.3.1 Gotcha 1: Aggregation in Templates

Since we have birth date information, it would be interesting to display a rough breakdown by age range of voucher requests and redemptions.

A very bad way to implement this would be to do all the processing at the template level. To be more specific in the context of this example:

- Don't iterate over the entire voucher list in your template's JavaScript section, using JavaScript variables to hold age range counts.
- Don't use the `add` template filter to sum up the voucher counts.

Those implementations are ways of getting around Django's limitations of logic in templates, but they'll slow down your pages drastically.

The better way is to move this processing out of your template and into your Python code. Sticking to our minimal approach of using templates only to *display* data that has already been processed, our template looks like this:

EXAMPLE 12.4

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

{% block content %}
<table>
  <thead>
    <tr>
      <th>Age Bracket</th>
      <th>Number of Vouchers Issued</th>
    </tr>
  </thead>
  <tbody>
    {% for age_bracket in age_brackets %}
    <tr>
      <td>{{ age_bracket.title }}</td>
      <td>{{ age_bracket.count }}</td>
    </tr>
    {% endfor %}
  </tbody>
</table>
```

```
</table>
{% endblock content %}
```

In this example, we can do the processing with a model manager, using the Django ORM's aggregation methods and the handy *dateutil* library described in [Appendix A: Packages Mentioned In This Book](#):

```
EXAMPLE 12.5
from datetime import datetime

from dateutil.relativedelta import relativedelta

from django.db import models

class VoucherManager(models.Manager):
    def age_breakdown(self):
        """ Returns a dict of age brackets/counts """
        age_brackets = []
        now = datetime.now()

        delta = now - relativedelta(years=18)
        count = self.model.objects.filter(birth_date__gt=delta).count()
        age_brackets.append(
            {"title": "0-17", "count": count}
        )
        delta = now - relativedelta(years=18)
        count = self.model.objects.filter(
            birth_date__lte=delta
        ).count()
        age_brackets.append(
            {"title": "18+", "count": count}
        )
        return age_brackets
```

This method would be called from a view, and the results would be passed to the template as a context

variable.

12.3.2 Gotcha 2: Filtering With Conditionals in Templates

Suppose we want to display a list of all the Greenfelds and the Roys who requested free pint vouchers, so that we could invite them to our family reunion. We want to filter our records on the name field.

A very bad way to implement this would be with giant loops and if statements at the template level.

BAD EXAMPLE 12.1

```
<h2>Greenfelds Who Want Ice Cream</h2>
<ul>
{% for voucher in voucher_list %}
    {% # Don't do this: conditional filtering in templates %}
    {% if "greenfeld" in voucher.name.lower %}
        <li>{{ voucher.name }}</li>
    {% endif %}
{% endfor %}
</ul>

<h2>Roys Who Want Ice Cream</h2>
<ul>
{% for voucher in voucher_list %}
    {% # Don't do this: conditional filtering in templates %}
    {% if "roy" in voucher.name.lower %}
        <li>{{ voucher.name }}</li>
    {% endif %}
{% endfor %}
</ul>
```

In this bad snippet, we're looping and checking for various "if" conditions. That's filtering a potentially gigantic list of records in templates, which is not designed for this kind of work, and will cause performance bottlenecks. On the other hand, databases like PostgreSQL and MySQL are great at filtering records, so this should be done at the database layer. The Django ORM can help us with this as demonstrated in the next example.

EXAMPLE 12.6

```
from django.views.generic import TemplateView

from .models import Voucher

class GreenfeldRoyView(TemplateView):
    template_name = "vouchers/views_conditional.html"

    def get_context_data(self, **kwargs):
        context = super(GreenfeldRoyView, self).get_context_data(**kwargs)
        context["greenfelds"] = \
            Voucher.objects.filter(name__icontains="greenfeld")
        context["roys"] = Voucher.objects.filter(name__icontains="roys")
        return context
```

Then to call the results, we use the following, simpler template:

EXAMPLE 12.7

```
<h2>Greenfelds Who Want Ice Cream</h2>
<ul>
{% for voucher in greenfelds %}
    <li>{{ voucher.name }}</li>
{% endfor %}
</ul>

<h2>Roys Who Want Ice Cream</h2>
<ul>
{% for voucher in roys %}
    <li>{{ voucher.name }}</li>
{% endfor %}
</ul>
```

It's easy to speed up this template by moving the filtering to a model manager. With this change, we now simply use the template to display the already-filtered data.

The above template now follows our preferred minimalist approach.

12.3.3 Gotcha 3: Complex Implied Queries in Templates

Despite the limitations on logic allowed in Django templates, it's all too easy to find ourselves calling unnecessary queries repeatedly in a view. For example, if we list users of our site and all their flavors this way:

BAD EXAMPLE 12.2

```
{# list generated via User.object.all() #}
<h1>Ice Cream Fans and their favorite flavors.</h1>
<ul>
{% for user in user_list %}
    <li>
        {{ user.name }}:
        {# DON'T DO THIS: Generated implicit query per user #}
        {{ user.flavor.title }}
        {# DON'T DO THIS: Second implicit query per user!!! #}
        {{ user.flavor.scoops_remaining }}
    </li>
{% endfor %}
</ul>
```

Then calling each user generates a second query. While that might not seem like much, we are certain that if we had enough users and made this mistake frequently enough, our site would have a lot of trouble.

One quick correction is to use the Django ORM's `select_related()` method:

EXAMPLE 12.8

```
{# list generated via User.object.all().select_related() #}
<h1>Ice Cream Fans and their favorite flavors.</h1>
<ul>
{% for user in user_list %}
    <li>
        {{ user.name }}:
        {{ user.flavor.title }}
    </li>
{% endfor %}
```

```
</ul>
```

One more thing: If you've embraced using model methods, the same applies. Be cautious putting too much query logic in the model methods called from templates.

12.3.4 Gotcha 4: Hidden CPU Load in Templates

Watch out for innocent-looking calls in templates that result in intensive CPU processing. Although a template might look simple and contain very little code, a single line could be invoking an object method that does a lot of processing.

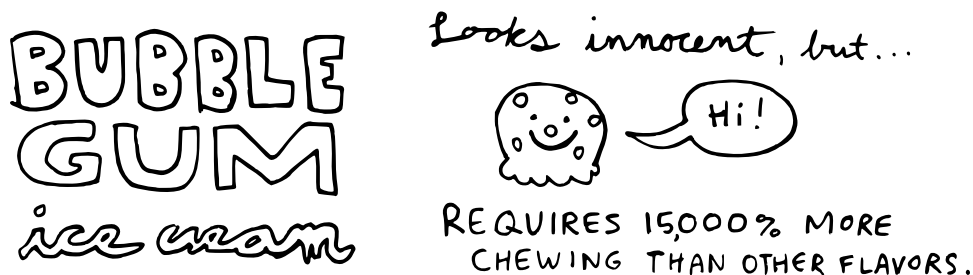


Figure 12.2: Bubble gum ice cream looks easy to eat but requires a lot of processing.

Common examples are template tags that manipulate images, such as the template tags provided by libraries like **sort-thumbnail**. In many cases tools like this work great, but we've had some issues. Specifically, the manipulation and the saving of image data to file systems (often across networks) inside a template means there is a chokepoint within templates.

This is why projects that handle a lot of image or data processing increase the performance of their site by taking the image processing out of templates and into views, models, helper methods, or asynchronous messages queues like Celery.

12.3.5 Gotcha 5: Hidden REST API Calls in Templates

You saw in the previous gotcha how easy it is to introduce template loading delays by accessing object method calls. This is true not just with high-load methods, but also with methods that contain REST API calls. A good example is querying an unfortunately slow maps API hosted by a third-party service that your project absolutely requires. Don't do this in the template code by calling a method attached to an object passed into the view's context.

Where should actual REST API consumption occur? We recommend doing this in:

- JavaScript code so after your project serves out its content, the client's browser handles the work. This way you can entertain or distract the client while they wait for data to load.
- The view's Python code where slow processes might be handled in a variety of ways including message queues, additional threads, multiprocesses, or more.

12.4 Don't Bother Making Your Generated HTML Pretty

Bluntly put, no one cares if the HTML generated by your Django project is attractive. In fact, if someone were to look at your rendered HTML, they'd do so through the lens of a browser inspector, which would realign the HTML spacing anyway. Therefore, if you shuffle up the code in your Django templates to render pretty HTML, you are wasting time obfuscating your code for an audience of yourself.

And yet, we've seen code like the following. This evil code snippet generates nicely formatted HTML but itself is an illegible, unmaintainable template mess:

BAD EXAMPLE 12.3

```
{% comment %}Don't do this! This code bunches everything
together to generate pretty HTML.
{% endcomment %}
{% if list_type=="unordered" %}<ul>{% else %}<ol>{% endif %}{% for
syrup in syrup_list %}<li class="{% syrup.temperature_type|roomtemp
%}"><a href="{% url 'syrup_detail' syrup.slug %}">{% syrup.title %}
</a></li>{% endfor %}{% if list_type=="unordered" %}</ul>{% else %}
</ol>{% endif %}
```

A better way of writing the above snippet is to use indentation and one operation per line to create a readable, maintainable template:

```
EXAMPLE 12.9

{# Use indentation/comments to ensure code quality #}
{# start of list elements #}
{% if list_type=="unordered" %}
    <ul>
{% else %}
    <ol>
{% endif %}

{% for syrup in syrup_list %}
    <li class="{ { syrup.temperature_type|roomtemp } }">
        <a href="{% url 'syrup_detail' syrup.slug %}">
            {% syrup.title %}
        </a>
    </li>
{% endfor %}

{# end of list elements #}
{% if list_type=="unordered" %}
    </ul>
{% else %}
    </ol>
{% endif %}
```

Are you worried about the volume of whitespace generated? Don't be. First of all, experienced developers favor readability of code over obfuscation for the sake of optimization. Second, there are compression and minification tools that can help more than anything you can do manually here. See [chapter 20](#), *Finding and Reducing Bottlenecks*, for more details.

12.5 Exploring Template Inheritance

Let's begin with a simple *base.html* file that we'll inherit from another template:

EXAMPLE 12.10

```
{# simple base.html #}
{% load staticfiles %}
<html>
<head>
    <title>
        {% block title %}Two Scoops of Django{% endblock title %}
    </title>
    {% block stylesheets %}
        <link rel="stylesheet" type="text/css"
            href="{% static "css/project.css" %}">
    {% endblock stylesheets %}
</head>
<body>
    <div class="content">
        {% block content %}
            <h1>Two Scoops</h1>
        {% endblock content %}
    </div>
</body>
</html>
```

The *base.html* file contains the following features:

- A title block containing: “Two Scoops of Django”.
- A stylesheets block containing a link to a *project.css* file used across our site.
- A content block containing “<h1>Two Scoops</h1>”.

Our example relies on just three template tags, which are summarized below:

Template Tag	Purpose
{% load %}	Loads the staticfiles built-in template tag library
{% block %}	Since <i>base.html</i> is a parent template, these define which child blocks can be filled in by child templates. We place links and scripts inside them so we can override if necessary.
{% static %}	Resolves the named static media argument to the static media server.

Table 12.1: Template Tags in `base.html`

To demonstrate *base.html* in use, we'll have a simple *about.html* inherit the following from it:

- A custom title.
- The original stylesheet and an additional stylesheet.
- The original header, a sub header, and paragraph content.
- The use of child blocks.
- The use of the `{{ block.super }}` template variable.

EXAMPLE 12.11

```
{% extends "base.html" %}
{% load staticfiles %}
{% block title %}About Audrey and Daniel{% endblock %}
{% block stylesheets %}
    {{ block.super }}
    <link rel="stylesheet" type="text/css"
        href="{% static "css/about.css" %}">
{% endblock stylesheets %}
{% block content %}
    {{ block.super }}
    <h2>About Audrey and Daniel</h2>
    <p>They enjoy eating ice cream</p>
{% endblock %}
```

When we render this template in a view, it generates the following HTML:

EXAMPLE 12.12

```
<html>
<head>
    <title>
        About Audrey and Daniel
    </title>
    <link rel="stylesheet" type="text/css"
        href="/static/css/project.css">
```

```
<link rel="stylesheet" type="text/css"
      href="/static/css/about.css">
</head>
<body>
  <div class="content">
    <h1>Two Scoops</h1>
    <h2>About Audrey and Daniel</h2>
    <p>They enjoy eating ice cream</p>
  </div>
</body>
</html>
```

Notice how the rendered HTML has our custom title, the additional stylesheet link, and more material in the body?

We'll use the table below to review the template tags and variables in the *about.html* template.

Template Object	Purpose
<code>{% extends %}</code>	Informs Django that <i>about.html</i> is inheriting or extending from <i>base.html</i>
<code>{% block %}</code>	Since <i>about.html</i> is a child template, <code>block</code> overrides the content provided by <i>base.html</i> . For example, this means our title will render as <code><title>Audrey and Daniel</title></code> .
<code>{{ block.super }}</code>	When placed in a child template's block, it ensures that the parent's content is also included in the block. For example, in the content block of the <i>about.html</i> template, this will render <code><h1>Two Scoops</h1></code> .

Table 12.2: Template Objects in *about.html*

Note that the `{% block %}` tag is used differently in *about.html* than in *base.html*, serving to override content. In blocks where we want to preserve the *base.html* content, we use `{{ block.super }}` variable to display the content from the parent block. This brings us to the next topic, `{{ block.super }}`.

12.6 block.super Gives the Power of Control

Let's imagine that we have a template which inherits everything from the *base.html* but replaces the projects' link to the *project.css* file with a link to *dashboard.css*. This use case might occur when you have a project with one design for normal users, and a dashboard with a different design for staff.

If we aren't using `{{ block.super }}`, this often involves writing a whole new base file, often named something like *base_dashboard.html*. For better or for worse, we now have two template architectures to maintain.

If we are using `{{ block.super }}`, we don't need a second (or third or fourth) base template. Assuming all templates extend from *base.html* we use `{{ block.super }}` to assume control of our templates. Here are three examples:

Template using both *project.css* and a custom link:

```
EXAMPLE 12.13
{% extends "base.html" %}
{% block stylesheets %}
    {{ block.super }} {# this brings in project.css #}
    <link rel="stylesheet" type="text/css"
        href="{% static "css/custom" %}" />
{% endblock %}
```

Dashboard template that excludes the *project.css* link:

```
EXAMPLE 12.14
{% extends "base.html" %}
{% block stylesheets %}
    <link rel="stylesheet" type="text/css"
        href="{% static "css/dashboard.css" %}" />
    {% comment %}
        By not using {{ block.super }}, this block overrides the
        stylesheet block of base.html
    {% endcomment %}
{% endblock %}
```

Template just linking the *project.css* file:

EXAMPLE 12.15

```
{% extends "base.html" %}
{% comment %}
    By not using {% block stylesheets %}, this template inherits the
    stylesheets block from the base.html parent, in this case the
    default project.css link.
{% endcomment %}
```

These three examples demonstrate the amount of control `block.super` provides. The variable serves a good way to reduce template complexity, but can take a little bit of effort to fully comprehend.

TIP: `block.super` is similar but not the same as `super()`

For those coming from an object oriented programming background, it might help to think of the behavior of the `{{ block.super }}` variable to be like a very limited version of the Python built-in function, `super()`. In essence, the `{{ block.super }}` variable and the `super()` function both provide access to the parent.

Just remember that they aren't the same. For example, the `{{ block.super }}` variable doesn't accept arguments. It's just a nice mnemonic that some developers might find useful.

12.7 Useful Things to Consider

The following are a series of smaller things we keep in mind during template development.

12.7.1 Avoid Coupling Styles Too Tightly to Python Code

Aim to control the styling of all rendered templates entirely via CSS and JS.

Use CSS for styling whenever possible. Never hardcode things like menu bar widths and color choices into your Python code. Avoid even putting that type of styling into your Django templates.

Here are some tips:

- If you have magic constants in your Python code that are entirely related to visual design layout, you should probably move them to a CSS file.
- The same applies to JavaScript.

12.7.2 Common Conventions

Here are some naming and style conventions that we recommend:

- We prefer underscores over dashes in template names, block names, and other names in templates. Most Django users seem to follow this convention. Why? Well, because underscores are allowed in names of Python objects but dashes are forbidden.
- We rely on clear, intuitive names for blocks. `{% block javascript %}` is good.
- We include the name of the block tag in the endblock. Never write just `{% endblock %}`, include the whole `{% endblock javascript %}`.
- Templates called by other templates are prefixed with `'_'`. This applies to templates called via `{% includes %}` or custom template tags. It does not apply to templates inheritance controls such as `{% extends %}` or `{% block %}`.

12.7.3 Location, Location, Location!

Templates should usually go into the root of the Django project, at the same level as the apps. This is the most common convention, and it's an intuitive, easy pattern to follow.

The only exception is when you bundle up an app into a third-party package. That package's template directory should go into app directly. We'll explore this in [section 17.9](#), How to Release Your Own Django Packages.

12.7.4 Use Named Context Objects

When you use generic display CBVs, you have the option of using the generic `{{ object_list }}` and `{{ object }}` in your template. Another option is to use the ones that are named after your model.

For example, if you have a Topping model, you can use `{{ topping_list }}` and `{{ topping }}` in your templates, `{{ object_list }}` and `{{ object }}`. This means both of the the following template examples will work:

EXAMPLE 12.16

```
{# toppings/topping_list.html #}
{# Using implicit names #}
<ol>
{% for object in object_list %}
    <li>{{ object }} </li>
{% endfor %}
</ol>

{# Using explicit names #}
<ol>
{% for topping in topping_list %}
    <li>{{ topping }} </li>
{% endfor %}
</ol>
```

12.7.5 Use URL Names Instead of Hardcoded Paths

A common developer mistake is to hardcode URLs in templates like this:

BAD EXAMPLE 12.4

```
<a href="/flavors/">
```

The problem with this is that if the URL patterns of the site need to change, all the URLs across the site need to be addressed. This impacts HTML, JavaScript, and even RESTful APIs.

Instead, we use the `{% url %}` tag and references the names in our **URLConf** files:

EXAMPLE 12.17

```
<a href="{% url 'flavors_list' %}">
```

12.7.6 Debugging Complex Templates

A trick recommended by Lennart Regebro is that when templates are complex and it becomes difficult to determine where a variable is failing, you can force more verbose errors through the use of the `TEMPLATE_STRING_IF_INVALID` setting:

EXAMPLE 12.18

```
# settings/local.py
TEMPLATE_STRING_IF_INVALID = "INVALID EXPRESSION: %s"
```

12.7.7 Don't Replace the Django Template Engine

If you need **Jinja2** or any other templating engine for certain views, then it's easy enough to use it for just those views without having to replace Django templates entirely.

For more details, see [chapter 14](#), *Tradeoffs of Replacing Core Components*, for a case study about replacing the Django template engine with Jinja2.

12.8 Summary

In this chapter we covered the following:

- Template inheritance including the use of `{{ block.super }}`.
- Writing legible, maintainable templates.
- Easy methods to optimize template performance.
- Covered issues with limitations of template processing.

In the next chapter we'll examine template tags and filters.

13 | Template Tags and Filters

Django provides dozens of default template tags and filters, all of which share the following common traits:

- All of the defaults have clear, obvious names.
- All of the defaults do just one thing.
- None of the defaults alter any sort of persistent data.

These traits serve as very good best practices when you have to write your own template tags. Let's now dive a bit deeper into practices and recommendations when writing custom filters and template tags.

13.1 Filters Are Functions

Filters are functions that accept just one or two arguments, and that don't give developers the ability to add behavior controls in Django templates.

We feel that this simplicity makes filters are less prone to abuse, since they are essentially just functions with decorators that make Python usable inside of Django templates. This means that they can be called as normal functions (although we prefer to have our filters call functions imported from helper modules).

In fact, a quick scan of the source code of Django's default filters at <http://2scoops.co/slugify-source> shows that the `slugify()` function simply calls the `from django.utils.text.slugify` function.

13.1.1 Filters Are Easy to Test

Testing a filter is just a matter of testing a function, which we cover in [chapter 18](#), *Testing Stinks and Is a Waste of Money!*.

13.1.2 Filters, Code Reuse, and Performance

It's no longer necessary to import `django.template.defaultfilters.slugify`. Instead use `django.utils.text.slugify`. While it might seem to be perfectly acceptable, since `django.template.defaultfilters.slugify` performs an import each time it's used, it can turn into a performance bottleneck.

Similarly, `remove_tags` is available at `django.utils.html.remove_tags()`.

Since filters are just functions, we advocate that anything but the simplest logic for them be moved to more reusable utility functions, perhaps stored in a *utils.py* module. Doing this makes it easier to introspect code bases and test, and can mean dramatically fewer imports.

13.1.3 When to Write Filters

Filters are good for modifying the presentation of data, and they can be readily reused in REST APIs and other output formats. Being constrained to two arguments limits the functionality so it's harder (but not impossible) to make them unbearably complex.

13.2 Custom Template Tags

“Please stop writing so many template tags. They are a pain to debug.”

– Audrey Roy, while debugging Daniel Greenfeld's code.

While template tags are great tools when developers have the discipline to keep them in check, in practice they tend to get abused. This section covers the problems that you run into when you put too much of your logic into template tags and filters.

13.2.1 Template Tags Are Harder To Debug

Template tags of any complexity can be challenging to debug. When they include opening and closing elements, they become even harder to handle. We've found liberal use of log statements and tests are very helpful when they become hard to inspect and correct.

13.2.2 Template Tags Make Code Reuse Harder

It can be difficult to consistently apply the same effect as a template tag on alternative output formats used by REST APIs, RSS feeds, or in PDF/CSV generation. If you do need to generate alternate formats, it's worth considering putting all logic for template tags into *utils.py*, for easy access from other views.

13.2.3 The Performance Cost of Template Tags

Template tags can have a significant performance cost, especially when they load other templates. While templates run much faster than they did in previous versions of Django, it's easy to lose those performance benefits if you don't have a deep understanding of how templates are loaded in Django.

If your custom template tags are loading a lot of templates, you might want to consider caching the loaded templates. See <http://2scoops.co/1.5-template-cached-loader> for more details.

13.2.4 When to Write Template Tags

These days, we're very cautious about adding new template tags. We consider two things before writing them:

- Anything that causes a read/write of data might be better placed in a model or object method.
- Since we implement a consistent naming standard across our projects, we can add an abstract base class model to our core.models module. Can a method or property in our project's abstract base class model do the same work as a custom template tag?

When should you write new template tags? We recommend writing them in situations where they are only responsible for rendering of HTML. For example, Projects with very complex HTML layouts with many different models or data types might use them to create a more flexible, understandable template architecture.

PACKAGE TIP: We Do Use Custom Template Tags

It sounds like we stay away from custom template tags, but that's not the case. We're just cautious. Interestingly enough, Daniel has been involved with at least three prominent libraries that make extensive use of template tags.

- `django-crispy-forms`
- `django-wysiwyg`
- `django-uni-form` (deprecated, use `django-crispy-forms` instead)

13.3 Naming Your Template Tag Libraries

The convention we follow is `<app_name>_tags.py`. Using the `twoscoops` example, we would have files named thus:

- `flavors_tags.py`
- `blog_tags.py`
- `events_tags.py`
- `tickets_tags.py`

This makes determining the source of a template tag library trivial to discover.

WARNING: Don't name your template tag libraries with the same name as your app

For example, naming the `events` app's templatetag library `events.py` is problematic. This will cause all sorts of problems because of the way that Django loads template tags. If you do this, expect things to break.

WARNING: Don't Use Your IDE's Features as an Excuse to Obfuscate Your Code

Do not rely on your text editor or IDE's powers of introspection to determine the name of your templatetag library.

13.4 Loading Your Template Tag Modules

In your template, right after `{% extends "base.html" %}` (or any other parent template besides *base.html*) is where you load your template tags:

EXAMPLE 13.1

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

{% load flavors_tags %}
```

Simplicity itself! Explicit loading of functionality! Hooray!

13.4.1 Watch Out for This Crazy Anti-Pattern

Unfortunately, there is an obscure anti-pattern that will drive you mad with fury each and every time you encounter it:

BAD EXAMPLE 13.1

```
# Don't use this code!
# It's an evil anti-pattern!
from django import template
template.add_to_builtins(
    "flavors.templatetags.flavors_tags"
)
```

The anti-pattern replaces the explicit load method described above with an implicit behavior which supposedly fixes a “Don't Repeat Yourself” (DRY) issue. However, any DRY “improvements” it creates are destroyed by the following:

- It will add some overhead due to the fact this literally loads the template tag library into each and every template loaded by `django.template.Template`. This means every inherited template, `template {% include %}`, `inclusion_tag`, and more will be impacted. While we have cautioned against premature optimization, we are also not in favor of adding this much unneeded extra computational work into our code when better alternatives exist.
- Because the template tag library is implicitly loaded, it immensely adds to the difficulty in introspection and debugging. Per the **Zen of Python**, “Explicit is better than Implicit.”
- The `add_to_builtins` method has no convention for placement. To our chagrin, we often find it placed in an `__init__` module or the template tag library itself, either of which can cause unexpected problems.

Fortunately, this is obscure because beginning Django developers don’t know enough to make this mistake and experienced Django developers get really angry when they have to deal with it.

13.5 Summary

It is our contention that template tags and filters should concern themselves only with the manipulation of presentable data. So long as we remember this when we write or use them, our projects run faster and are easier to maintain.

14 | Tradeoffs of Replacing Core Components

There's a lot of hype around swapping out core parts of Django's stack for other pieces. Should you do it?

Short Answer: Don't do it. These days, even Instagram says on Forbes.com that it's completely unnecessary: <http://2scoops.co/instagram-insights>

Long Answer: It's certainly possible, since Django modules are simply just Python modules. Is it worth it? Well, it's worth it only if:

- You are okay with sacrificing your ability to use third-party Django packages.
- You have no problem giving up the powerful Django admin.
- You have already made a determined effort to build your project with core Django components, but you are running into walls that are major blockers.
- You have already analyzed your own code to find and fix the root causes of your problems. For example, you've done all the work you can to reduce the numbers of queries made in your templates.
- You've explored all other options including caching, denormalization, etc.
- Your project is a real, live production site with tons of users. In other words, you're certain that you're not just optimizing prematurely.
- You're willing to accept the fact that upgrading Django will be extremely painful or impossible going forward.

That doesn't sound so great anymore, does it?

14.1 The Temptation to Build FrankenDjango

Every year, a new fad leads waves of developers to replace some particular core Django component. Here's a summary of some of the fads we've seen come and go.

Fad	Reasons
Replacing the database/ORM with a NoSQL database and corresponding ORM replacement.	Not okay: "I have an idea for a social network for ice cream haters. I just started building it last month. I need it to be web-scale!!!1!" Okay: "Our site has 50M users and I'm hitting the limits of what I can do with indexes, query optimization, caching, etc. We're also pushing the limits of our Postgres cluster. I've done a lot of research on this and am going to try storing a simple denormalized view of our activity feed data in Redis to see if it helps."
Replacing Django's template engine with Jinja2, Mako, or something else.	Not okay: "I read on Hacker News that Jinja2 is faster. I don't know anything about caching or optimization, but I need Jinja2!" Sometimes okay: "I hate having logic in Python modules. I just want logic in my templates!" Sometimes okay: "I have a small number of views which generate 1MB+ HTML pages designed for Google to index!"

Table 14.1: Fad-based Reasons to Replace Components of Django

14.2 Case Study: Replacing the Django Template Engine

Let's take a closer look at one of the most common examples of replacing core Django components: replacing the Django template engine with **Jinja2**.

14.2.1 Excuses, Excuses

The excuse for doing this used to be performance. That excuse is no longer quite as valid. A lot of work has gone into improving the performance of Django's templating system, and newer benchmarks

indicate that performance is greatly improved.

A common excuse for replacing the Django template engine is to give you more flexibility. This is a poor excuse because your template layer should be as thin as possible. Case in point, adding ‘flexibility’ to templates also means adding complexity.

14.2.2 What if I'm Hitting the Limits of Templates?

Are you really? You might just be putting your logic in the wrong places:

- If you are putting tons of logic into templates, template tags, and filters, consider moving that logic into model methods or helper utilities.
- Whatever can't be placed into model methods might go into views.
- Template tags and filters should be a last resort. We covered this in more detail in [chapter 13](#) ‘Template Tags and Filters’.

14.2.3 What About My Unusual Use Case?

Okay, but what if I need to generate a 1 MB+ HTML page for Google to index?

Interestingly enough, this is the only use case we know of for replacing Django 1.5 templates. The size of these pages can and will crash browsers, so it's really meant for machines to read from each other. These giant pages require tens of thousands of loops to render the final HTML, and this is a place where Jinja2 (or other template engines) might provide a noticeable performance benefit.

However, besides these exceptions, we've found we don't need Jinja2. So rather than replace Django templates across the site, we use Jinja2 in only the affected view:

EXAMPLE 14.1

```
# flavors/views.py
import os
from django.conf import settings
from django.http import HttpResponse

from jinja2 import Environment, FileSystemLoader
```

```
from syrup.models import Syrup

JINJA2_TEMPLATES_DIR = os.path.join(
    settings.PROJECT_ROOT,
    "templates",
    "jinja2"
)
JINJA2_LOADER = FileSystemLoader(JINJA2_TEMPLATES_DIR)
JINJA2_ENV = Environment(loader=JINJA2_LOADER)
TEMPLATE = JINJA2_ENV.get_template("big_syrup_list.html")

def big_syrup_list(request):
    object_list = Syrup.objects.filter()
    content = TEMPLATE.render(object_list=object_list)
    return HttpResponse(content)
```

As we demonstrate, it's pretty easy to bring in the additional performance of Jinja2 without removing Django templates from a project.

14.3 Summary

Always use the right tool for the right job. We prefer to go with stock Django components, just like we prefer using a scoop when serving ice cream. However, there are times when other tools make sense.

Just don't follow the fad of using a fork for ice cream!

15 | Working With the Django Admin

When people ask, “*What are the benefits of Django over other web frameworks?*” the admin is what usually comes to mind.

Imagine if every gallon of ice cream came with an admin interface. You’d be able to not just see the list of ingredients, but also add/edit/delete ingredients. If someone was messing around with your ice cream in a way that you didn’t like, you could limit or revoke their access.

Pretty surreal, isn’t it? Well, that’s what web developers coming from another background feel like when they first use the Django admin interface. It gives you so much power over your web application automatically, with little work required.

15.1 It's Not for End Users

The Django admin interface is designed for site administrators, not end users. It’s a place for your site administrators to add/edit/delete data and perform site management tasks.

Although it’s possible to stretch it into something that your end users could use, you really shouldn’t. It’s just not designed for use by every site visitor.

15.2 Admin Customization vs. New Views

It’s usually not worth it to heavily customize the Django admin. Sometimes, creating a simple view or form from scratch results in the same desired functionality with a lot less work.

We’ve always had better results with creating custom management dashboards for client projects than we have with modifying the admin to fit clients’ needs.

15.3 Viewing String Representations of Objects

The default admin page for a Django app looks something like this:



Figure 15.1: Admin list page for an ice cream bar app.

That's because the default string representation of an `IceCreamBar` object is “IceCreamBar object”.

It would be helpful to display something better here. We recommend that you do the following as standard practice:

- 1 Always implement the `__unicode__()` method for each of your Django models. This will give you a better default string representation in the admin and everywhere else.
- 2 If you want to change the admin list display in a way that isn't quite a string representation of the object, then use `list_display`.

Implementing `__unicode__()` is simple:

EXAMPLE 15.1

```
from django.db import models

class IceCreamBar(models.Model):
```

```

name = models.CharField(max_length=100)
shell = models.CharField(max_length=100)
filling = models.CharField(max_length=100)
has_stick = models.BooleanField(default=True)

def __unicode__(self):
    return self.name

```

The result:

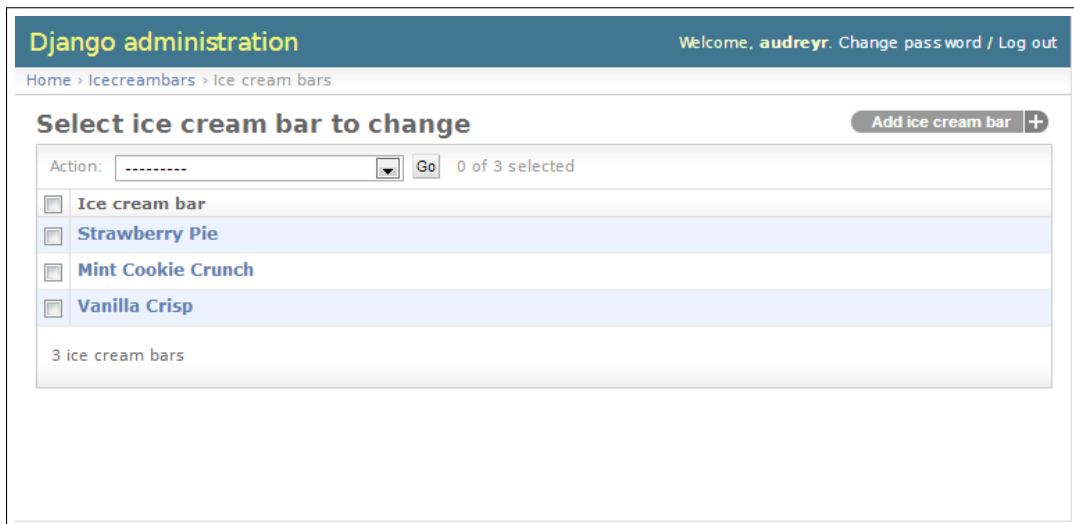


Figure 15.2: Improved admin list page with better string representation of our objects.

It's more than that, though. When you're in the shell, you see the better string representation:

EXAMPLE 15.2

```

>>> IceCreamBar.objects.all()
[<IceCreamBar: Vanilla Crisp>, <IceCreamBar: Mint Cookie Crunch>,
<IceCreamBar: Strawberry Pie>]

```

The `__unicode__()` method is called whenever you call `unicode()` on an object. This occurs in the

Django shell, templates, and by extension the Django admin. Therefore, try to make the results of `__unicode__()` nice, readable representation of Django model instances.

Django also provides the `__unicode__()` method as a default if we don't add a `__str__()` method. In fact, in all our years working with Django, neither author has ever bothered writing a `__str__()` method.

If you still want to show data for additional fields on the app's admin list page, you can then use `list_display`:

EXAMPLE 15.3

```
from django.contrib import admin

from .models import IceCreamBar

class IceCreamBarAdmin(admin.ModelAdmin):
    list_display = ("name", "shell", "filling",)

admin.site.register(IceCreamBar, IceCreamBarAdmin)
```

The result with the specified fields:

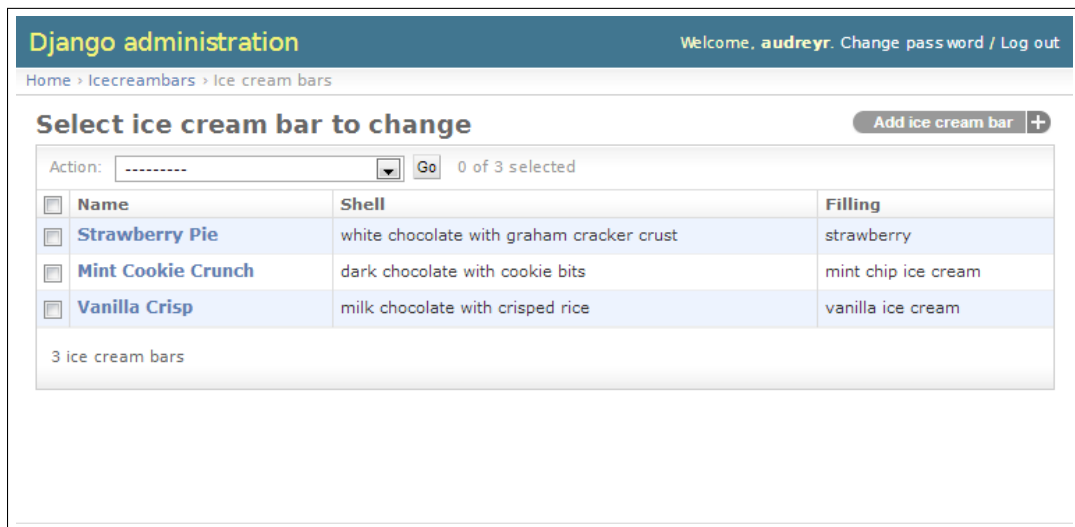


Figure 15.3: Improved admin list page with better string representation of our objects.

15.4 Adding Callables to ModelAdmin Classes

You can use callables such as methods and functions to add view functionality to the Django `django.contrib.admin.ModelAdmin` class. This allows you to really modify the list and display screens to suit your ice cream project needs.

For example, it's not uncommon to want to see the exact URL of a model instance in the Django admin. If you define a `get_absolute_url()` method for your model, what Django provides in the admin is link to a redirect view whose URL is very different from the actual object URL. Also, there are cases where the `get_absolute_url()` method is meaningless (REST APIs come to mind).

In the example below, we demonstrate how to use a simple callable to provide a link to our target URL:

EXAMPLE 15.4

```
from django.contrib import admin
from django.core.urlresolvers import reverse

from icecreambars.models import IceCreamBar

class IceCreamBarAdmin(admin.ModelAdmin):

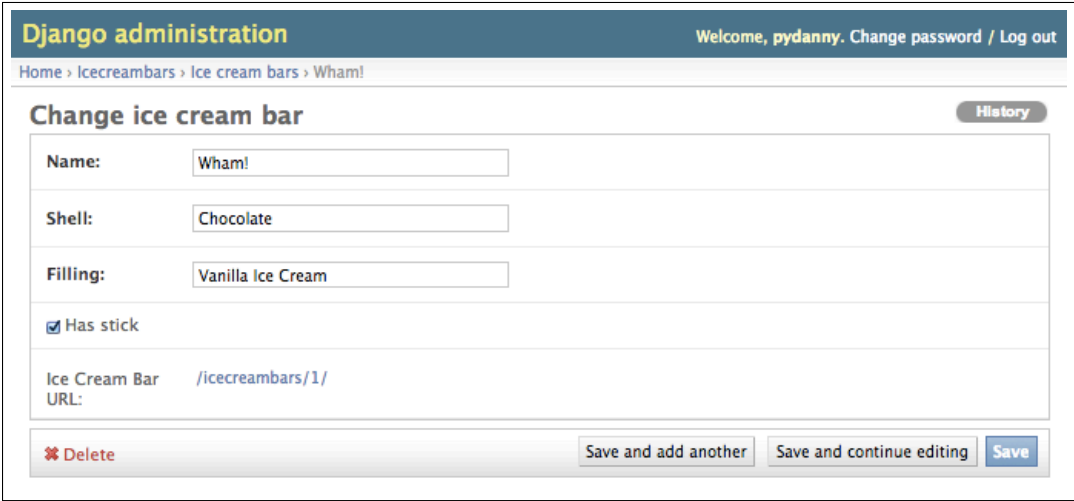
    list_display = ("name", "shell", "filling",)
    readonly_fields = ("show_url",)

    def show_url(self, instance):
        url = reverse("ice_cream_bar_detail",
                     kwargs={"pk": instance.pk})
        response = """<a href="{0}">{0}</a>""".format(url)
        return response

    show_url.short_description = "Ice Cream Bar URL"
    # Displays HTML tags
    # Never set allow_tags to True against user submitted data!!!
    show_url.allow_tags = True

admin.site.register(IceCreamBar, IceCreamBarAdmin)
```

Since a picture is worth a thousand words, here is what our callable does for us:



The screenshot shows the Django Admin interface for editing an 'Ice cream bar'. The header includes 'Django administration' and a user welcome message. The breadcrumb trail is 'Home > Icecreambars > Ice cream bars > Wham!'. The main heading is 'Change ice cream bar' with a 'History' button. The form contains three text input fields: 'Name' with 'Wham!', 'Shell' with 'Chocolate', and 'Filling' with 'Vanilla Ice Cream'. Below these is a checked checkbox for 'Has stick'. At the bottom, it displays 'Ice Cream Bar' with the URL '/icecreambars/1/'. Action buttons at the bottom are 'Delete', 'Save and add another', 'Save and continue editing', and 'Save'.

Figure 15.4: Displaying URL in the Django Admin.

WARNING: Use the `allow_tags` attribute With Caution

The `allow_tags` attribute, which is set to `False` by default, can be a security issue. When `allow_tags` is set to `True`, HTML tags are allowed to be displayed in the admin.

Our hard rule is `allow_tags` can only be used on system generated data like primary keys, dates, and calculated values. Data such as character and text fields are completely out, as is any other user entered data.

15.5 Django's Admin Documentation Generator

One of the more interesting developer tools that Django provides is the `django.contrib.admindocs` package. Created in an era before the advent of the documentation tools that we cover in [chapter 19](#) *Documentation: Be Obsessed*, it remains a useful tool.

It's useful because it introspects the Django framework to display docstrings for project components like models, views, custom template tags, and custom filters. Even if a project's components don't contain any docstrings, simply seeing a list of harder-to-introspect items like oddly named custom

template tags and custom filters can be really useful in exploring the architecture of a complicated, existing application.

Using `django.contrib.admindocs` is easy, but we like to to reorder the steps described in the formal documentation:

- ❶ `pip install docutils` into your project's **virtualenv**.
- ❷ Add `django.contrib.admindocs` to your `INSTALLED_APPS`.
- ❸ Add `(r'^admin/doc/', include('django.contrib.admindocs.urls'))` to your root `URLConf`. Make sure it's included before the `r'^admin/'` entry, so that requests to `/admin/doc/` don't get handled by the latter entry.
- ❹ *Optional*: Linking to templates requires the `ADMIN_FOR` setting to be configured.
- ❺ *Optional*: Using the `admindocs` bookmarklets requires the `XViewMiddleware` to be installed.

Once you have this in place, go to `/admin/doc/` and explore. You may notice a lot of your project's code lacks any sort of documentation. This is addressed in the formal documentation on `django.contrib.admindocs`: <http://2scoops.co/1.5-admindocs> and our own chapter on [chapter 19](#), *Documentation: Be Obsessed*.

15.6 Securing the Django Admin and Django Admin Docs

It's worth the effort to take the few extra steps to prevent hackers from accessing the admin, since the admin gives you so much power over your site. See [chapter 21](#), *Security Best Practices* for details, specifically the following sections:

- [section 21.15](#) 'Securing the Django Admin'
- [section 21.16](#) 'Securing Admin Docs'

15.7 Summary

In this chapter we covered the following:

- Who should be using the Django admin.
- When to use the Django admin and when to roll a new dashboard.
- String representation of objects.

- Adding callables to Django admin classes.
- Using Django's admin docs.
- Encouraging you to secure the Django admin.

16 | Dealing With the User Model

The best practices for this have changed significantly in Django 1.5. The “right way” before Django 1.5 was a bit confusing, and there’s still confusion around pre-1.5, so it’s especially important that what we describe here is only applied to Django 1.5.

So let’s briefly go over best practices for Django 1.5 or higher.

16.1 Use Django's Tools for Finding the User Model

From Django 1.5 onwards, the advised way to get to the user class is as follows:

EXAMPLE 16.1

```
# Stock user model definition
>>> from django.contrib.auth import get_user_model
>>> get_user_model()
<class 'django.contrib.auth.models.User'>

# When the project has a custom user model definition
>>> from django.contrib.auth import get_user_model
>>> get_user_model()
<class 'profiles.models.UserProfile'>
```

It is now possible to get two different User model definitions depending on the project configuration. This doesn’t mean that a project can have two different User models; it means that every project can customize its own User model. This is new in Django 1.5 and a radical departure from earlier versions of Django.

16.1.1 Use settings.AUTH_USER_MODEL for Foreign Keys to User

From Django 1.5 onwards, the official preferred way to attach ForeignKey, OneToOneField, or ManyToManyField to User is as follows:

EXAMPLE 16.2

```
from django.conf import settings
from django.db import models

class IceCreamStore(models.Model):

    owner = models.OneToOneField(settings.AUTH_USER_MODEL)
    title = models.CharField(max_length=255)
```

Yes, it looks a bit strange, but that's what the official Django docs advise.

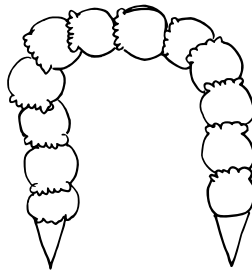


Figure 16.1: This looks strange too.

16.2 Custom User Fields for Projects Starting at Django 1.5

In Django 1.5, as long as you incorporate the necessary required methods and attributes, you can create your own user model with its own fields. You can still do things the old pre-Django 1.5 way, but you're not stuck with having a User model with just email, first_name, last_name, and username fields for identity.

WARNING: Migrating From Pre-1.5 User models to 1.5's Custom User Models.

At the time of writing, the best practices for this are still being determined. We suggest that you carefully try out option #1 below, as it should work with a minimum of effort. For Django 1.5-style custom User model definitions, we recommend option #2 and option #3 for new projects only.

This is because custom User model definitions for option #2 and option #3 adds new User tables to the database that will not have the existing project data. Unless project-specific steps are taken to address matters, migration means ORM connections to related objects will be lost.

When best practices for migrating between User model types have been established by the community, we'll publish errata and update future editions of this book. In the meantime, we look forward to any suggestions for good practices or patterns to follow.

16.2.1 Option 1: Linking Back From a Related Model

This code is very similar to pre-Django 1.5 projects. You continue to use User (called preferably via `django.contrib.auth.get_user_model()`) and keep your related fields in a separate model (e.g. Profile). Here's an example:

EXAMPLE 16.3

```
from django.conf import settings
from django.db import models

class UserProfile(models.Model):

    # If you do this you need to either have a post_save signal or
    #     redirect to a profile_edit view on initial login.
    user = models.OneToOneField(settings.AUTH_USER_MODEL)
    favorite_ice_cream = models.CharField(max_length=30)
```

TIP: For Now, You Can Still Use the `user.get_profile()` Method.

The `user.get_profile()` method is deprecated as of Django 1.5. Instead, we advise using a standard Django ORM join instead, for example `user.userprofile.favorite_ice_cream`.

16.2.2 Option 2: Subclass `AbstractUser`

Choose this option if you like Django's `User` model fields the way they are, but need extra fields.

WARNING: Third-Party Packages Should Not Be Defining the User Model

Unless the express purpose of the third-party package is to provide a new `User` model, third-party packages should never use option #2 to add fields to the `User` model.

Here's an example of how to subclass `AbstractUser`:

```
EXAMPLE 16.4
# profiles/models.py
from django.contrib.auth.models import AbstractUser
from django.db import models
from django.utils.translation import ugettext_lazy as _

class KarmaUser(AbstractUser):
    karma = models.PositiveIntegerField(_("karma"),
                                      default=0,
                                      blank=True)
```

It's much more elegant than the pre-1.5 way, isn't it?

The other thing you have to do is set this in your settings:

EXAMPLE 16.5

```
AUTH_USER_MODEL = "profiles.KarmaUser"
```

16.2.3 Option 3: Subclass AbstractBaseUser

AbstractBaseUser is the bare-bones option with only 3 fields: `password`, `last_login`, and `is_active`.

Choose this option if:

- You're unhappy with the fields that the User model provides by default, such as `first_name` and `last_name`.
- You prefer to subclass from an extremely bare-bones clean slate but want to take advantage of the AbstractBaseUser sane default approach to storing passwords.

WARNING: Third-Party Packages Should Not Be Defining the User Model

Unless the express purpose of the third-party package is to provide a new User model, third-party packages should never use option #3 to add fields to the User model.

Let's try it out with a custom User model for the Two Scoops project. Here are our requirements:

- We need an email address.
- We need to handle permissions per the traditional `django.contrib.auth.models` use of `PermissionsMixin`; providing standard behavior for the Django admin.
- We don't need the first or last name of a user.
- We need to know their favorite ice cream topping.

Looking over the Django 1.5 documentation on customizing the User model, we notice there is a full example (<http://2scoops.co/1.5-customizing-user>). It doesn't do exactly what we want, but we can modify it. Specifically:

- We'll need to add `PermissionsMixin` to our custom User model.

- We'll need to implement a favorite toppings field.
- We'll need to ensure that the `admin.py` fully supports our custom `User` model. Unlike the example in the documentation, we do want to track groups and permissions.

Let's do it! We'll call our new `User` model, `TwoScoopsUser`.

Before we start writing our new `TwoScoopsUser` model, we need to write a custom `TwoScoopsUserManager`. This is generally required for custom `User` models as the auth system expects certain methods on the default manager, but the manager for the default user class expects fields we are not providing.

EXAMPLE 16.6

```
# profiles/models.py
from django.db import models

from django.contrib.auth.models import (
    BaseUserManager, AbstractBaseUser, PermissionsMixin
)

class TwoScoopsUserManager(BaseUserManager):
    def create_user(self, email, favorite_topping,
                    password=None):
        """
        Creates and saves a User with the given email,
        favorite topping, and password.
        """
        if not email:
            msg = "Users must have an email address"
            raise ValueError(msg)

        if not favorite_topping:
            msg = "Users must have a favorite topping"
            raise ValueError(msg)

        user = self.model(
            email=TwoScoopsUserManager.normalize_email(email),
            favorite_topping=favorite_topping,
```

```
)

user.set_password(password)
user.save(using=self._db)
return user

def create_superuser(self,
                      email,
                      favorite_topping,
                      password):
    """
    Creates and saves a superuser with the given email,
    favorite topping and password.
    """
    user = self.create_user(email,
                            password=password,
                            favorite_topping=favorite_topping
    )
    user.is_admin = True
    user.is_staff = True
    user.is_superuser = True
    user.save(using=self._db)
    return user
```

With our TwoScoopsUserManager complete, we can write the TwoScoopsUser class.

EXAMPLE 16.7

```
# profiles/models.py (after the TwoScoopsUserManager)
class TwoScoopsUser(AbstractBaseUser, PermissionsMixin):
    """ Inherits from both the AbstractBaseUser and
    PermissionMixin.
    """
    email = models.EmailField(
        verbose_name="email address",
        max_length=255,
        unique=True,
```

```
        db_index=True,
    )
    favorite_topping = models.CharField(max_length=255)

    USERNAME_FIELD = "email"
    REQUIRED_FIELDS = ["favorite_topping", ]

    is_active = models.BooleanField(default=True)
    is_admin = models.BooleanField(default=False)
    is_staff = models.BooleanField(default=False)

    objects = TwoScoopsUserManager()

    def get_full_name(self):
        # The user is identified by their email and
        #     favorite topping
        return "%s prefers %s" % (self.email,
                                self.favorite_topping)

    def get_short_name(self):
        # The user is identified by their email address
        return self.email

    def __unicode__(self):
        return self.email
```

Boom! There's no `first_name` or `last_name`, which is probably what you wanted if you're choosing this option. The permissions are in place and most importantly, users have a favorite ice cream topping!

Like the first option, don't forget to set this in your settings:

```
EXAMPLE 16.8
# settings/base.py
AUTH_USER_MODEL = "profiles.TwoScoopsUser"
```


Upon syncdb, this will create a new TwoScoopsUser table and various other references. We ask that you try this on a new database rather than an existing one.

Once the table has been created, we can create a superuser locally via the shell:

```
python manage.py createsuperuser
```

With our new superuser account in hand, let's create the *profiles/admin.py* so we can see the results.

Again, we follow the lead of the example in the Django documentation. We modify it to include the permissions and favorite toppings fields. The results:

EXAMPLE 16.9

```
# profiles/admin.py
from django import forms
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin
from django.contrib.auth.forms import ReadOnlyPasswordHashField

from .models import TwoScoopsUser

class TwoScoopsUserCreationForm(forms.ModelForm):
    """A form for creating new users. Includes all the
        required fields, plus a repeated password.
    """
    password1 = forms.CharField(label="Password",
                                widget=forms.PasswordInput)
    password2 = forms.CharField(label="Password confirmation",
                                widget=forms.PasswordInput)

    class Meta:
        model = TwoScoopsUser
        fields = ("email", "favorite_topping")

    def clean_password2(self):
        # Check that the two password entries match
        password1 = self.cleaned_data.get("password1")
        password2 = self.cleaned_data.get("password2")
        if password1 and password2 and password1 != password2:
```

```
        msg = "Passwords don't match"
        raise forms.ValidationError(msg)
    return password2

def save(self, commit=True):
    # Save the provided password in hashed format
    user = super(TwoScoopsUserCreationForm,
                  self).save(commit=False)
    user.set_password(self.cleaned_data["password1"])
    if commit:
        user.save()
    return user

class TwoScoopsUserChangeForm(forms.ModelForm):
    """ A form for updating users. Includes all the fields
        on the user, but replaces the password field with
        admin's password hash display field.
    """
    password = ReadOnlyPasswordHashField()

    class Meta:
        model = TwoScoopsUser

    def clean_password(self):
        # Regardless of what the user provides, return the
        # initial value. This is done here, rather than on
        # the field, because the field does not have access
        # to the initial value
        return self.initial["password"]

class TwoScoopsUserAdmin(UserAdmin):
    # Set the add/modify forms
    add_form = TwoScoopsUserCreationForm
    form = TwoScoopsUserChangeForm

    # The fields to be used in displaying the User model.
    # These override the definitions on the base UserAdmin
```

```

# that reference specific fields on auth.User.
list_display = ("email", "is_staff", "favorite_topping")
list_filter = ("is_staff", "is_superuser",
               "is_active", "groups")
search_fields = ("email", "favorite_topping")
ordering = ("email",)
filter_horizontal = ("groups", "user_permissions",)
fieldsets = (
    (None, {"fields": ("email", "password")}),
    ("Personal info", {"fields":
                       ("favorite_topping",)}),
    ("Permissions", {"fields": ("is_active",
                                "is_staff",
                                "is_superuser",
                                "groups",
                                "user_permissions")}),
    ("Important dates", {"fields": ("last_login",)}),
)
add_fieldsets = (
    (None, {
        "classes": ("wide",),
        "fields": ("email", "favorite_topping",
                  "password1", "password2")}
    ),
)

# Register the new TwoScoopsUserAdmin
admin.site.register(TwoScoopsUser, TwoScoopsUserAdmin)

```

Now if you go to your admin home and login, you'll be able to create and modify the `TwoScoopsUser` model records.

16.3 Summary

The new `User` model makes this an exciting time to be involved in Django. We are getting to participate in a major infrastructure change with wide-ranging implications. We are the ones who get

to pioneer the best practices.

In this chapter we covered the new method to find the User model and define our own custom ones. Depending on the needs of a project, they can either continue with the current way of doing things or customize the actual user model.

The next chapter is a dive into the world of third-party packages.

17 | Django's Secret Sauce: Third-Party Packages

The real power of Django is more than just the framework and documentation available at <http://djangoproject.com>. It's the vast, growing selection of third-party Django and Python packages provided by the open source community. There are many, many third-party packages available for your Django projects which can do an incredible amount of work for you. These packages have been written by people from all walks of life, and they power much of the world today.

Much of professional Django and Python development is about the incorporation of third-party packages into Django projects. If you try to write every single tool that you need from scratch, you'll have a hard time getting things done.

This is especially true for us in the consulting world, where client projects consist of many of the same or similar building blocks.

17.1 Examples of Third-Party Packages

[Appendix A: Packages Mentioned In This Book](#) covers all of the packages mentioned throughout Two Scoops of Django. This list is a great starting point if you're looking for highly-useful packages to consider adding to your projects.

Note that not all of those packages are Django-specific, which means that you can use some of them in other Python projects. (Generally, Django-specific packages generally have names prefixed with “django-”, but there are many exceptions.)

17.2 Know About the Python Package Index

The **Python Package Index (PyPI)**, located at <http://pypi.python.org/pypi>, is a repository of software for the Python programming language. As of the start of 2013, it lists over 27,000 packages, including Django itself.

For the vast majority of Python community, no open-source project release is considered official until it occurs on the Python Package Index.

The Python Package Index is much more than just a directory. Think of it as the world's largest center for Python package information and files. Whenever you use **pip** to install a particular release of Django, pip downloads the files from the Python Package Index. Most Python and Django packages are downloadable from the Python Package Index in addition to pip.

17.3 Know About DjangoPackages.com

Django Packages (<http://djangopackages.com>) is a directory of reusable apps, sites, tools and more for your Django projects. Unlike PyPI, it doesn't store the packages themselves, instead providing a mix of hard metrics gathered from the Python Package Index, GitHub, Bitbucket, ReadTheDocs, and "soft" data entered by user.

Django Packages is best known as a comparison site for evaluating package features. On Django Packages, packages are organized into handy grids so they can be compared against each other.

Django Packages also happens to have been created by the authors of this book, with contributions from numerous folks in the Python community. We continue to maintain and improve it as a helpful resource for Django users.

17.4 Know Your Resources

Django developers unaware of the critical resources of Django Packages and the Python Package Index are denying themselves one of the most important advantages of using Django and Python. If you are not aware of these tools, it's well worth the time you spend educating yourself.

As a Django (and Python) developer, make it your mission to use third-party libraries instead of reinventing the wheel whenever possible. The best libraries have been written, documented, and

tested by amazingly competent developers working around the world. Standing on the shoulders of these giants is the difference between amazing success and tragic downfall.

As you use various packages, study and learn from their code. You'll learn patterns and tricks that will make you a better developer.

17.5 Tools for Installing and Managing Packages

To take full advantage of all the packages available for your projects, having **virtualenv** and **pip** installed isn't something you can skip over. It's mandatory.

Refer to [chapter 2](#), *The Optimal Django Environment Setup*, for more details.

17.6 Package Requirements

As we mentioned earlier in [chapter 5](#), *Settings and Requirements Files*, we manage our Django/Python dependencies with requirements files. These files go into the *requirements/* directory that exists in the root of our projects.

TIP: Researching Third-Party Packages To Use

If you want to learn more about the dependencies we list in this and other chapters, please refer to [Appendix A: Packages Mentioned In This Book](#).

17.7 Wiring Up Django Packages: The Basics

When you find a third-party package that you want to use, follow these steps:

17.7.1 Step 1: Read the Documentation for the Package

Are you sure you want to use it? Make sure you know what you're getting into before you install any package.

17.7.2 Step 2: Add Package and Version Number to Your Requirements

If you recall from [chapter 5](#) *Settings and Requirements Files*, a `requirements/base.txt` file looks something like this (but probably longer):

EXAMPLE 17.1

```
Django==1.5.1
coverage==3.6
django-discover-runner==0.2.2
django-extensions==0.9
django-floppyforms==1.0
```

Note that each package is pinned to a specific version number. *Always* pin your package dependencies to version numbers.

What happens if you don't pin your dependencies? You are almost guaranteed to run into problems at some point when you try to reinstall or change your Django project. When new versions of packages are released, you can't expect them to be backwards-compatible.

Our sad example: Once we followed a software-as-a-service platform's instructions for using their library. As they didn't have their own Python client, but an early adopter had a working implementation on GitHub, those instructions told us to put the following into our `requirements/base.txt`:

BAD EXAMPLE 17.1

```
-e git+https://github.com/erly-adptr/py-junk.git#egg=py-jnk
```

Our mistake. We should have known better and pinned it to a particular git revision number.

Not the early adopter's fault at all, but they pushed up a broken commit to their repo. Once we had to fix a problem on a site very quickly, so we wrote a bugfix and tested it locally in development. It passed the tests. Then we deployed it to production in a process that grabs all dependency changes; of course the broken commit was interpreted as a valid change. Which meant, while fixing one bug, we crashed the site.

Not a fun day.

The purpose of using pinned releases is to add a little formality and process to our published work. Especially in Python, GitHub and other repos are a place for developers to publish their work-in-progress, not the final, stable work upon which our production-quality projects depend.

17.7.3 Step 3: Install the Requirements Into Your Virtualenv

Assuming you are already in a working virtualenv and are at the `<repo_root>` of your project, you `pip install` the appropriate requirements file for your setup, e.g. *requirements/dev.txt*.

If this is the first time you've done this for a particular virtualenv, it's going to take a while for it to grab all the dependencies and install them.

17.7.4 Step 4: Follow the Package's Installation Instructions Exactly

Resist the temptation to skip steps unless you're very familiar with the package. Since Django package developers love to get people to use their efforts, most of the time the installation instructions they've authored make it easy to get things running.

17.8 Troubleshooting Third-Party Packages

Sometimes you run into problems setting up a package. What should you do?

First, make a serious effort to determine and solve the problem yourself. Pore over the documentation and make sure you didn't miss a step. Search online to see if others have run into the same issue. Be willing to roll up your sleeves and look at the package source code, as you may have found a bug.

If it appears to be a bug, see if someone has already reported it in the package repository's issue tracker. Sometimes you'll find workarounds and fixes there. If it's a bug that no one has reported, go ahead and file it.

If you still get stuck, try asking for help in all the usual places: StackOverflow, IRC #django, the project's IRC channel if it has its own one, and your local Python user group. Be as descriptive and provide as much context as possible about your issue.

17.9 Releasing Your Own Django Packages

Whenever you write a particularly useful Django app, consider packaging it up for reuse in other projects.

The best way to get started is to follow Django's Advanced Tutorial: *How to Write Reusable Apps for the basics*: <https://docs.djangoproject.com/en/1.5/intro/reusable-apps/>

In addition to what is described in that tutorial, we recommend that you also:

- Create a public repo containing the code. Most Django packages are hosted on GitHub these days, so it's easiest to attract contributors there, but various alternatives exist (Sourceforge, Bitbucket, Launchpad, Gitorious, Assembla, etc.).
- Release the package on the Python Package Index (<http://pypi.python.org>). Follow the submission instructions: <http://2scoops.co/submit-to-pypi>
- Add the package to Django Packages: <http://djangopackages.com>.
- Use Read the Docs (<http://rtfd.org>) to host your **Sphinx** documentation.

TIP: Where Should I Create A Public Repo?

There are websites that offer free source code hosting and version control for open-source projects. As mentioned in [chapter 2](#), The Optimal Django Environment Setup, GitHub or Bitbucket are two popular options.

When choosing a hosted version control service, keep in mind that pip only supports Git, Mercurial, Bazaar, and Subversion.

17.10 What Makes a Good Django Package?

Here's a checklist for you to use when releasing a new open-source Django package. Much of this applies to Python packages that are not Django-specific. This checklist is also helpful for when you're evaluating a Django package to use in any of your projects.

This section is adapted from our DjangoCon 2011 talk, "*Django Package Thunderdome: Is Your Package Worthy?*": <http://2scoops.co/django-thunderdome-slides>

17.10.1 Purpose

Your package should do something useful and do it well. The name should be descriptive. The package's repo root folder should be prefixed with 'django-' to help make it easier to find.

If part of the package's purpose can be accomplished with a related Python package, then create a separate Python package and use it as a dependency.

17.10.2 Scope

Your package's scope should be tightly focused on one small task. This means that your application logic will be tighter, and users will have an easier time patching or replacing the package.

17.10.3 Documentation

A package without documentation is a pre-alpha package. Docstrings don't suffice as documentation.

As described in [chapter 19](#), *Documentation: Be Obsessed*, your docs should be written in **ReStructuredText**. A nicely-formatted version of your docs should be generated with Sphinx and hosted publicly. We encourage you to use <https://readthedocs.org/> with webhooks so that your formatted documentation automatically updates whenever you make a change.

If your package has dependencies, they should be documented. Your package's installation instructions should also be documented. The installation steps should be bulletproof.

17.10.4 Tests

Your package should have tests. Tests improve reliability, make it easier to advance Python/Django versions, and make it easier for others to contribute effectively. Write up instructions on how to run your package's test suite. If you or any contributor can run your tests easily before submitting a pull request, then you're more likely to get better quality contributions.

17.10.5 Activity

Your package should receive regular updates from you or contributors if/when needed. When you update the code in your repo, you should consider uploading a minor or major release to the Python Package Index.

17.10.6 Community

Great open-source packages, including those for Django, often end up receiving contributions from other developers in the open source community. All contributors should receive attribution in a *CONTRIBUTORS.rst* or *AUTHORS.rst* file.

Be an active community leader if you have contributors or forks of your package. If your package is forked by other developers, pay attention to their work. Consider if there are ways that parts or all of their work can be merged into your fork. If the package's functionality diverges a lot from your package's purpose, be humble and consider asking the other developer to give their fork a new name.

17.10.7 Modularity

Your package should be as easily pluggable into any Django project that doesn't replace core components (templates, ORM, etc) with alternatives. Installation should be minimally invasive. Be careful not to confuse modularity with over-engineering, though.

17.10.8 Availability on PyPI

All major and minor releases of your package should be available for download from the Python Package Index. Developers who wish to use your package should not have to go to your repo to get a working version of it. Use proper version numbers per the next section.

17.10.9 Proper Version Numbers

Like Django and Python, we prefer to adhere to the strict version of **PEP 386** naming schema. In fact we follow the ‘**A.B.C**’ pattern. Let’s go through each element:

‘**A**’ represents the major version number. Increments should only happen with large changes that break backwards compatability from the previous major version. It’s not uncommon to see large API changes between versions.

‘**B**’ is the minor version number. Increments include less breaking changes, or deprecation notices about forthcoming changes.

‘**C**’ represents bugfix releases, and purists call this the ‘micro’ release. It’s not uncommon for developers to wait until a project has it’s first release at this level before trying the latest major or minor release of an existing project.

For alpha, beta, or release-candidates for a project, the convention is to place this information as a suffix to the upcoming version number. So you might have:

- Django 1.5
- django-crispy-forms 1.1b1

WARNING: Don't Upload Unfinished Code To PyPI

PyPI is meant to be the place where dependable, stable packages can be harnessed to build Python projects. *PyPI is not the place for Alpha, Beta, or Release Candidate code*, especially as pip and other tools will fetch the latest release by default.

Be nice to other developers and follow the convention of only placing proper releases on PyPI.

Additional Reading:

- <http://www.python.org/dev/peps/pep-0386>

17.10.10 License

Your package needs a license. Preferably, it should be licensed under the **BSD** or **MIT** licenses, which are generally accepted for being permissive enough for most commercial or noncommercial uses.

Create a *LICENSE.rst* file in your repo root, mention the license name at the top, and paste in the appropriate text from the approved list at the **Open Source Initiative** (OSI) <http://opensource.org/licenses/category> for the license that you choose.

TIP: Licenses Protect You and the World

In this era of casual litigation and patent trolls adding a software license isn't just a matter of protecting your ownership of the code. It's much, much more. If you don't license your code, or use an unapproved license not vetted by real lawyers, you run the risk of your work being used as a weapon by a patent troll, or in the case of financial or medical disaster, you could be held liable.

OSI-approved licenses all include a couple critical statements on **copyright**, **redistribution**, **disclaimer of warranty**, and **limitation of liability**.

17.10.11 Clarity of Code

The code in your Django package should be as clear and simple as possible, of course. Don't use weird, unusual Python/Django hacks without explaining what you are doing.

17.11 Summary

Django's real power is in the vast selection of third-party packages available to you for use in your Django projects.

Make sure that you have pip and virtualenv installed and know how to use them, since they're your best tools for installing packages on your system in a manageable way.

Get to know the packages that exist. The Python Package Index and Django Packages are a great starting point for finding information about packages.

Package maturity, documentation, tests, and code quality are good starting criteria when evaluating a Django package.

Finally, we've provided our base requirements file to give you ideas about the packages that we use.

Installation of stable packages is the foundation of Django projects big and small. Being able to use packages means sticking to specific releases, not just the trunk or master of a project. Barring a specific release, you can rely on a particular commit. Fixing problems that a package has with your project takes diligence and time, but remember to ask for help if you get stuck.

18 | Testing Stinks and Is a Waste of Money!

There, got you to this chapter.

Now you have to read it.

We'll try and make this chapter interesting.

18.1 Testing Saves Money, Jobs, and Lives

Daniel's Story: Ever hear the term “smoke test”?

Gretchen Davidian, a Management and Program Analyst at **NASA**, told me that when she was still an engineer, her job as a tester was to put equipment intended to get into space through such rigorous conditions that they would begin emitting smoke and eventually catch on fire.

That sounds exciting! Employment, money, and lives were on the line, and knowing Gretchen's attention to detail, I'm sure she set a lot of hardware on fire.

Keep in mind that for a lot of us developers the same risks are on the line as NASA. I recall in 2004 while working for a private company how a single miles-vs-kilometers mistake cost a company hundreds of thousands of dollars in a matter of hours. Quality Assurance (QA) staff lost their jobs, which meant money and health benefits. In other words, employment, money, and possibly lives can be lost without adequate tests. While the QA staff were very dedicated, everything was done via manually clicking through projects, and human error simply crept into the testing process.

Today, as Django moves into a wider and wider set of applications, the need for automated testing is just as important as it was for Gretchen at NASA and for the poor QA staff in 2004. Here are some cases where Django is used today that have similar quality requirements:

- Your application handles medical information.
- Your application provides life-critical resources to people in need.
- Your application works with other people's money now or will at some point in the future.

PACKAGE TIP: Useful Libraries For Testing Django Projects

We like to use **coverage.py** and **django-discover-runner**.

What these tools do is provide a clear insight into what parts of your code base are covered by tests. You also get a handy percentage of how much of your code is covered by tests. Even 100% test coverage doesn't guarantee a bug-free application, but it helps.

We want to thank Ned Batchelder for his incredible work in maintaining coverage.py. It's a superb project and useful for any Python related project.

18.2 How to Structure Tests

Let's say we've just created a new Django app. The first thing we do is delete the default but useless *tests.py* module that `django-admin.py startapp` creates.

In its place, we create a tests directory and place an *__init__.py* file in it so it becomes a valid Python module. Then, inside the new tests module, because most apps need them, we create *test_forms.py*, *test_models.py*, *test_views.py* modules. Tests that apply to forms go into *test_forms.py*, model tests go into *test_models.py*, and so on.

Here's what it looks like:

EXAMPLE 18.1

```
popsicles/
  tests/
    __init__.py
    test_forms.py
```

```
test_models.py
test_views.py
```

Also, if we have other files besides *forms.py*, *models.py* and *views.py* that need testing, we create corresponding test files and drop them into the *tests/* directory too.

TIP: Prefix Test Modules With `test_`

It's critically important that we always prefix test modules with *test_*, otherwise we can't configure `django-discover-runner` to discover just our test files.

When you don't prefix your test modules this way, `django-discover-runner` will attempt to evaluate *all modules*, including your settings files!

18.3 How to Write Unit Tests

It's not uncommon for developers to feel at the top of their game at the moment they are writing code. When they revisit that same code in months, weeks, days, or even hours and it's not uncommon for developers to feel as if that same code is of poor quality.

The same applies to writing unit tests.

Over the years, we've evolved a number of practices we like to follow when writing tests, including **unit tests**. Our goal is always to write the most meaningful tests in the shortest amount of time. Hence the following:

18.3.1 Each Test Method Tests One Thing

A test method must be extremely narrow in what it tests. A single test *should never* assert the behavior of multiple views, models, forms, or even multiple methods within a class. Instead, a single test should assert the behavior of a single view, model, form, method or function.

Of course, therein lies a conundrum. How does one run a test for a view, when views often require the use of models, forms, methods, and functions?

The trick is to be absolutely minimalistic when constructing the environment for a particular test, as shown in the example below:

EXAMPLE 18.2

```
import json

from django.core.urlresolvers import reverse
from django.test import TestCase

from flavors.models import Flavor

class FlavorAPITests(TestCase):

    def setUp(self):
        Flavor.objects.get_or_create(title="A Title", slug="a-slug")

    def test_list(self):
        url = reverse("flavor_object_api")
        response = self.client.get(url)
        self.assertEqual(response.status_code, 200)
        data = json.loads(response.content)
        self.assertEqual(len(data), 1)
```

In this test, taken from code testing the API we presented in [section 11.2](#), ‘Implementing a Simple JSON API’, [chapter 11](#), *Building REST APIs in Django*, we use the `setUp()` method to create the minimum possible number of records needed to run the test.

Here’s a much larger example, one based off of the REST API example that we provided in [chapter 11](#).

EXAMPLE 18.3

```
import json

from django.core.urlresolvers import reverse
```

```
from django.test import TestCase
from django.utils.http import urlencode

from flavors.models import Flavor

class DjangoRestFrameworkTests(TestCase):

    def setUp(self):
        Flavor.objects.get_or_create(title="title1", slug="slug1")
        Flavor.objects.get_or_create(title="title2", slug="slug2")

        self.create_read_url = reverse("flavor_rest_api")
        self.read_update_delete_url = \
            reverse("flavor_rest_api", kwargs={"slug": "slug1"})

    def test_list(self):
        response = self.client.get(self.create_read_url)

        # Are both titles in the content?
        self.assertContains(response, "title1")
        self.assertContains(response, "title2")

    def test_detail(self):
        response = self.client.get(self.read_update_delete_url)
        data = json.loads(response.content)
        content = {"id": 1, "title": "title1", "slug": "slug1",
                  "scoops_remaining": 0}

        self.assertEqual(data, content)

    def test_create(self):
        post = {"title": "title3", "slug": "slug3"}
        response = self.client.post(self.create_read_url, post)
        data = json.loads(response.content)
        self.assertEqual(response.status_code, 201)
        content = {"id": 3, "title": "title3", "slug": "slug3",
                  "scoops_remaining": 0}

        self.assertEqual(data, content)
```

```
self.assertEqual(Flavor.objects.count(), 3)

def test_delete(self):
    response = self.client.delete(self.read_update_delete_url)
    self.assertEqual(response.status_code, 204)
    self.assertEqual(Flavor.objects.count(), 1)

def test_update(self):
    # urlencode the PUT because self.client.put doesn't do it.
    put = urlencode(
        {"title": "Triple Peanut Butter Cup",
         "slug": "triple-peanut-butter-cup"}
    )
    # This only applies to Django 1.5+
    # Send as form data or Django's client.put uses
    # "application/octet-stream".
    response = self.client.put(self.read_update_delete_url, put,
                               content_type="application/x-www-form-urlencoded")
    self.assertEqual(response.status_code, 200)
    self.assertEqual(Flavor.objects.get(pk=1).title,
                     "Triple Peanut Butter Cup")
```

18.3.2 Don't Write Tests That Have to Be Tested

Tests should be written as simply as possible. If the code in a test or called to help run a test feels complicated or abstracted, then you have a problem. In fact, we ourselves are guilty of writing over-complicated utility test functions that required their own tests in the past. As you can imagine, this made debugging the actual tests a nightmare.

Don't Repeat Yourself Doesn't Apply to Writing Tests

The `setUp()` method is really useful for generating reusable data across all test methods in a test class. However, sometimes we need similar but different data between test methods, which is where we often fall into the trap of writing fancy test utilities. Or worse, we decide that rather than write

20 similar tests, we can write a single method that when passed certain arguments will handle all the work for us.

Our favorite method of handling these actions is to just dig in and write the same or similar code multiple times. In fact, we'll quietly admit to copy/pasting code between tests to expedite our work.

18.3.3 Don't Rely on Fixtures

We've learned over time that using fixtures is problematic. The problem is that fixtures are hard to maintain as a project's data changes over time. Modifying JSON-formatted files to match your last migration is hard, especially as it can be difficult to identify during the JSON load process where your JSON file(s) is either broken or a subtly inaccurate representation of the database.

Rather than wrestle with fixtures, we've found it's easier to write code that relies on the ORM. Other people like to use third-party packages.

PACKAGE TIP: Tools to Generate Test Data

The following are popular tools for test data generation:

- **factory_boy** A package that generates model test data.
- **model_mommy** Another package that generates model test data.
- **mock** Not explicitly for Django, this allows you to replace parts of your system with mock objects. This project made its way into the standard library as of Python 3.3.

18.3.4 Things That Should Be Tested

Everything! Seriously, you should test whatever you can, including:

Views: viewing of data, changing of data, and custom class-based view methods.

Models: creating/updating/deleting of models, model methods, model managers, model manager methods.

Forms: form methods, `clean()` methods, and custom fields.

Validators: really dig in and write multiple test methods against each custom validator you write. Pretend you are a malignant intruder attempting to damage the data in the site.

Signals: Since they act at a distance, signals can cause grief especially if you lack tests on them.

Filters: Since filters are essentially just functions accepting one or two arguments, writing tests for them should be easy.

Template Tags: Since template tags can do anything and can even accept template context, writing tests often becomes much more challenging. This means you really need to test them, since otherwise you may run into edge cases.

Miscellany: Context processors, middleware, email, and anything else not covered in this list.

The only things that shouldn't be tested are parts of your project that are already covered by tests in core Django and third-party packages. For example, a model's fields don't have to be tested if you're using Django's standard fields as-is. However, if you're creating a new type of field (e.g. by subclassing `FileField`), then you should write detailed tests for anything that could go wrong with your new field type.



Figure 18.1: Test as much of your project as you can, as if it were free ice cream.

18.4 Continuous Integration

For medium and large projects, we recommend setting up a continuous integration (CI) server to run the project's test suite whenever code is committed and pushed to the project repo.

If setting up a continuous integration (CI) server is too much of a hassle, you can use a hosted continuous integration service like Travis CI.

18.4.1 Resources for Continuous Integration

- http://en.wikipedia.org/wiki/Continuous_Integration
- <http://jenkins-ci.org/>
- <http://www.caktusgroup.com/blog/2010/03/08/django-and-hudson-ci-day-1/>
- <http://bartek.im/showoff-jenkins/>
- <http://ci.djangoproject.com/>
- <http://docs.python-guide.org/en/latest/scenarios/ci.html>

18.5 Who Cares? We Don't Have Time for Tests!

“Tests are the Programmer’s stone, transmuting fear into boredom.” –Kent Beck

Let’s say you are confident of your coding skill and decide to skip testing to increase your speed of development. Or maybe you feel lazy. It’s easy to argue that even with test generators and using tests instead of the shell, they can increase the time to get stuff done.

Oh, really?

What about when it’s time to upgrade?

That’s when the small amount of work you did up front to add tests saves you a lot of work.

For example, in the summer of 2010, Django 1.2 was the standard when we started Django Packages (<http://www.djangopackages.com>). Since then we’ve stayed current with new Django versions, which has been really useful. Because of our pretty good test coverage, moving up a version of Django (or the various dependencies) has been easy. Our path to upgrade:

- Increase the version in a local instance of Django Packages.
- Run the tests.
- Fix any errors that are thrown by the tests.
- Do some manual checking.

If Django Packages didn't have tests, any time we upgraded *anything* we would have to click through dozens and dozens of scenarios manually, which is error prone. Having tests means we can make changes and dependency upgrades with the confidence that our users (i.e. the Django community) won't have to deal with a buggy experience.

This is the benefit of having tests.

18.6 The Game of Test Coverage

A great, fun game to play is trying get **test coverage** as high as possible. Every work day we increase our test coverage is a victory, and every day the coverage goes down is a loss.

18.7 Setting Up the Test Coverage Game

Yes, we call test coverage a game. It's a good tool for developers to push themselves. It's also a nice metric that both developers and their clients/employers/investors can use to help evaluate the status of a project.

We advocate following these steps because most of the time we want to only test our own project's apps, not all Django and the myriad of third-party libraries that are the building blocks of our project. Testing those 'building blocks' takes an enormous amount of time, which is a waste because most are already tested or require additional setup of resources.

18.7.1 Step 1: Set Up a Test Runner

In our settings directory, we create a *settings/test.py* module and add the following:

EXAMPLE 18.4

```
""" Test settings and globals which
    allow us to run our test suite
    locally. """

from .base import *

##### TEST SETTINGS
```

```
TEST_RUNNER = "discover_runner.DiscoverRunner"
TEST_DISCOVER_TOP_LEVEL = PROJECT_ROOT
TEST_DISCOVER_ROOT = PROJECT_ROOT
TEST_DISCOVER_PATTERN = "test_*"

##### IN-MEMORY TEST DATABASE
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.sqlite3",
        "NAME": ":memory:",
        "USER": "",
        "PASSWORD": "",
        "HOST": "",
        "PORT": "",
    },
}
```

TIP: It's Okay to Use SQLite3 to Speed Up Tests

For tests we like to use an in-memory instance of SQLite3 to expedite the running of tests. We can have Django use **PostgreSQL** or **MySQL** (or other databases) but after years of writing tests for Django we've yet to catch problems caused by **SQLite3's** loose field typing.

18.7.2 Step 2: Run Tests and Generate Coverage Report

Let's try it out! In the command-line, at the *<project.root>*, type:

EXAMPLE 18.5

```
$ coverage run manage.py test --settings=twoscoops.settings.test
```

If we have nothing except for the default tests for two apps, we should get a response that looks like:

EXAMPLE 18.6

```
Creating test database for alias "default"...  
..  
-----  
Ran 2 tests in 0.008s  
  
OK  
  
Destroying test database for alias "default"...
```

This doesn't look like much, but what it means is that we've constrained our application to only run the tests that you want. Now it's time to go and look at and analyze our embarrassingly low test coverage numbers.

18.7.3 Step 3: Generate the report!

coverage.py provides a very useful method for generating HTML reports that don't just provide percentage numbers of what's been covered by tests, it also shows us the places where code is not tested. In the command-line, at the *<project-root>*:

EXAMPLE 18.7

```
coverage html --include="$SITE_URL*" --omit="admin.py"
```

Ahem...don't forget to change *<project-root>* to match your development machine's structure and don't forget the trailing asterisk (the "*" character)! For example, depending on where one does things, the *<path-to-project-root>* could be:

- */Users/audreyr/code/twoscoops/twoscoops/*
- */Users/pydanny/projects/twoscoops/twoscoops/*
- *c:\ twoscoops*

After this runs, in the *<project-root>* directory you'll see a new directory called *htmlcov/*. In the *htmlcov/* directory, open the *index.html* file using any browser.

What is seen in the browser is the test results for your test run. Unless you already wrote some tests, the total on the front page will be in the single digits, if not at 0%. Click into the various modules listed and you should see lots of code that's red-colored. *Red is bad.*

Let's go ahead and admit that our project has a low coverage total. If your project has a low coverage total, you need to admit it as well. It's okay just so long as we also resolve to improve the coverage total.

In fact, there is nothing wrong in saying publicly that you are working to improve a project's test coverage. Then, other developers (including ourselves) will cheer you on!

18.8 Playing the Game of Test Coverage

The game has a single rule:

Mandate that no commit can lower test coverage.

So if we go to add a feature and coverage is 65% when we start, we can't merge our thing in until coverage is at least 65% again. At the end of each day, if your test coverage goes up by any amount, you are winning.

Keep in mind that the gradual increase of test coverage can be a very good thing over huge jumps. Gradual increases can mean that we developers aren't putting in bogus tests to bump up coverage numbers; instead, we are improving the quality of the project.

18.9 Summary

All of this might seem silly, but testing can be very serious business. In a lot of developer groups this subject, while gamified, is taken very seriously. Lack of stability in a project can mean the loss of clients, contracts, and even employment.

In the next chapter we cover a common obsession of Python developers: documentation.

19 | Documentation: Be Obsessed

Given a choice between ice cream and writing great documentation, most Python developers would probably choose to write the documentation. That being said, writing documentation while eating ice cream is even better.

When you have great documentation tools like **reStructuredText** and **Sphinx**, you actually can't help it but want to add docs to your projects.

PACKAGE TIP: Install Sphinx Systemwide

We've found that simply installing *Sphinx* fetches for us all the pieces you need to document our Django (or Python) project. We recommend **pip** installing Sphinx systemwide, as you'll want to have it handy for every Django project.

19.1 Use reStructuredText for Python Docs

You'll want to learn and follow the standard Python best practices for documentation. These days, reStructuredText (**RST**) is the most common *markup* language used for documenting Python projects.

What follows are links to the formal reStructuredText specification and a couple sample projects which benefit from using it:

- <http://2scoops.co/restructured-text-specification>
- <https://docs.djangoproject.com/en/1.5/>
- <http://docs.python.org>

While it's possible to study the formal documentation for reStructuredText and learn at least the basics, here is a quick primer of some very useful commands you should learn.

EXAMPLE 19.1

Section Header

=====

****emphasis (bold/strong)****

italics

Simple link: <http://django.2scoops.org>

Fancier Link: `Two Scoops of Django`_

.. _`Two Scoops of Django`: <https://django.2scoops.org>

Subsection Header

#) An enumerated list item

#) Second item

* First bullet

* Second bullet

 * Indented Bullet

 * Note carriage return and indents

Literal code block::

```
def like():  
    print("I like Ice Cream")
```

```
for i in range(10):  
    like()
```


Python colored code block (requires pygments):

```
code-block:: python
```

```
# You need to "pip install pygments" to make this work.
```

```
for i in range(10):  
    like()
```

JavaScript colored code block:

```
code-block:: javascript
```

```
console.log("Don't use alert()");
```

19.1.1 Use Sphinx to Generate Documentation From *reStructuredText*

Sphinx is a tool for generating nice-looking docs from your *.rst* files. Output formats include **HTML**, **LaTeX**, manual pages, and plain text.

Follow the instructions to generate Sphinx docs: <http://sphinx-doc.org/>.

TIP: Build Your Sphinx Documentation At Least Weekly

You never know when bad cross-references or invalid formatting can break the Sphinx build. Rather than discover that the documentation is unbuildable at an awkward moment, just make a habit of creating it on a regular basis.

19.2 What Docs Should Django Projects Contain?

Developer-facing documentation refers to notes and guides that developers need in order to set up and maintain a project. This includes notes on installation, deployment, architecture, how to run tests or submit pull requests, and more. We’ve found that it really helps to place this documentation in all our projects, private or public. Here we provide a table that describes what we consider the absolute minimum documentation:

Filename or Directory	Reason	Notes
<i>README.rst</i>	Every Python project you begin should have a README.rst file in the repository root.	Provide at least a short paragraph describing what the project does. Also, link to the installation instructions in the docs/ directory.
<i>docs/</i>	Your project documentation should go in one, consistent location. This is the Python community standard.	A simple directory
<i>docs/deployment.rst</i>	This file lets you take a day off.	A point-by-point set of instructions on how to install/update the project into production, even if it's done via something powered by Ruby, Chef, Fabric, or a Makefile.
<i>docs/installation.rst</i>	This is really nice for new people coming into a project or when you get a new laptop and need to set up the project.	A point-by-point set of instructions on how to onboard yourself or another developer with the software setup for a project.
<i>docs/architecture.rst</i>	A guide for understanding what things evolved from as a project ages and grows in scope.	This is how you imagine a project to be in simple text and it can be as long or short as you want. Good for keeping focused at the beginning of an effort.

Table 19.1: Documentation Django Projects Should Contain

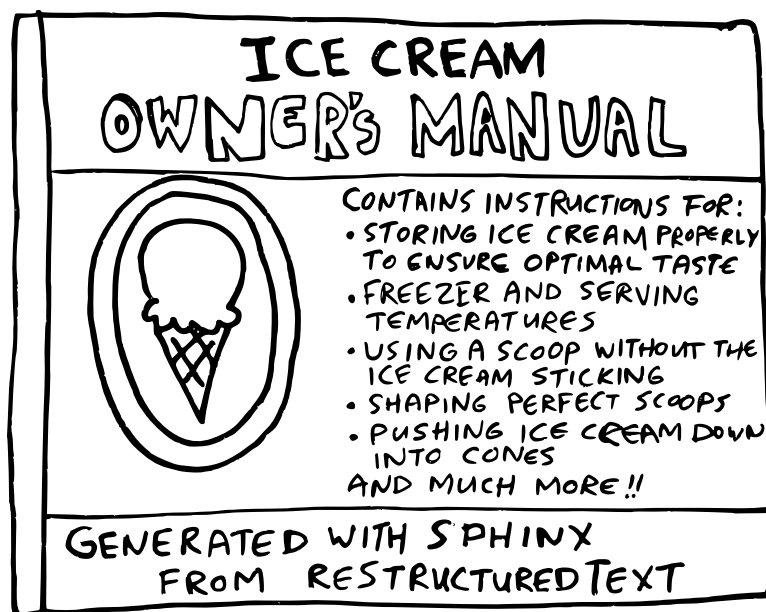


Figure 19.1: Even ice cream could benefit from documentation.

19.3 Wikis and Other Documentation Methods

For whatever reason, if you can't place developer-facing documentation in the project itself, you should have other options. While wikis, online document stores, and word processing documents don't have the feature of being placed in version control, they are better than no documentation.

Please consider creating documents within these other methods with the same names as the ones we suggested in the table on the previous page.

19.4 Summary

In this chapter we went over the following:

- The use of reStructuredText to write documentation in plaintext format.
- The use Sphinx to render your documentation in HTML or PDF formats.
- Advice on the documentation requirements for any Django project.

Next, we'll take a look at common bottlenecks in Django projects and ways to deal with them.

20 | Finding and Reducing Bottlenecks

This chapter covers a few basic strategies for identifying bottlenecks and speeding up your Django projects.

20.1 Should You Even Care?

Remember, premature optimization is bad. If your site is small- or medium-sized and the pages are loading fine, then it's okay to skip this chapter.

On the other hand, if your site's user base is growing steadily or you're about to land a strategic partnership with a popular brand, then read on.

20.2 Speed Up Query-Heavy Pages

This section describes how to reduce bottlenecks caused by having too many queries, as well as those caused by queries that aren't as snappy as they could be.

We also urge you to read up on database access optimization in the official Django docs: <http://2scoops.co/1.5-db-optimization>

20.2.1 Find Excessive Queries With Django Debug Toolbar

You can use **django-debug-toolbar** to help you determine where most of your queries are coming from. You'll find bottlenecks such as:

- Duplicate queries in a page.
- ORM calls that resolve to many more queries than you expected.
- Slow queries.

You probably have a rough idea of some of the URLs to start with. For example, which pages don't feel snappy when they load?

Install `django-debug-toolbar` locally if you don't have it yet. Configure it to include the `SQLDebugPanel`. Then run your project locally, open it in a web browser, and expand the debug toolbar. It'll show you how many queries the current page contains.

PACKAGE TIP: Packages For Profiling and Performance Analysis

django-debug-toolbar is a critical development tool and an invaluable aid in page-by-page analysis. We also recommend adding **django-cache-panel** to your project, but only configured to run when `settings/dev.py` module is called. This will increase visibility into what your cache is doing.

django-extensions comes with a tool called `RunProfileServer` that starts Django's `runserver` command with `hotshot`/profiling tools enabled.

newrelic is a third-party library provided by <http://newrelic.com>. They provide a free service that really helps in performance analysis of staging or production sites. Newrelic's for-pay service is amazing, and often worth the investment.

20.2.2 Reduce the Number of Queries

Once you know which pages contain an undesirable number of queries, figure out ways to reduce that number. Some of the things you can attempt:

- Try using `select_related()` in your ORM calls to combine queries. It follows `ForeignKey` relations and combines more data into a larger query. If using CBVs, `django-braces` makes this doing this trivial with the `SelectRelatedMixin`. But beware of queries that get too large!

- If the same query is being generated more than once per template, move the query into the Python view, add it to the context as a variable, and point the template ORM calls at this new context variable.
- Implement caching using a key/value store such as **Memcached**. Then write tests to assert the number of queries run in a view. See <http://2scoops.co/1.5-test-num-queries> for instructions.

20.2.3 Speed Up Common Queries

The length of time it takes for individual queries can also be a bottleneck. Here are some tips, but consider them just starting points:

- Make sure your indexes are helping speed up your most common slow queries. Look at the raw SQL generated by those queries, and index on the fields that you filter/sort on most frequently. Look at the generated WHERE and ORDER.BY clauses.
- Understand what your indexes are actually doing in production. Development machines will never perfectly replicate what happens in production, so learn how to analyze and understand what's really happening with your database.
- Look at the query plans generated by common queries.
- Turn on your database's slow query logging feature and see if any slow queries occur frequently.
- Use django-debug-toolbar in development to identify potentially-slow queries defensively, before they hit production.

Once you have good indexes, and once you've done enough analysis to know which queries to rewrite, here are some starting tips on how to go about rewriting them:

- ❶ Rewrite your logic to return smaller result sets when possible.
- ❷ Re-model your data in a way that allows indexes to work more effectively.
- ❸ Drop down to raw SQL in places where it would be more efficient than the generated query.

TIP: Use EXPLAIN ANALYZE / EXPLAIN

If you're using PostgreSQL, you can use `EXPLAIN ANALYZE` to get an extremely detailed query plan and analysis of any raw SQL query. For more information, see:

- <http://www.revsys.com/writings/postgresql-performance.html>
- <http://2scoops.co/craig-postgresql-perf2>

The MySQL equivalent is the `EXPLAIN` command, which isn't as detailed but is still helpful. For more information, see:

- <http://dev.mysql.com/doc/refman/5.6/en/explain.html>

20.3 Get the Most Out of Your Database

You can go a bit deeper beyond optimizing database access. Optimize the database itself! Much of this is database-specific and already covered in other books, so we won't go into too much detail here.

20.3.1 Know What Doesn't Belong in the Database

Frank Wiles of Revsys taught us that there are two things that should never go into any large site's relational database:

Logs. Don't add logs to the database. Logs may seem OK on the surface, especially in development. Yet adding this many writes to a production database will slow their performance. When the ability to easily perform complex queries against your logs is necessary, we recommend third-party services such as splunk.com or loggly.com or even use of document based NoSQL databases including MongoDB or CouchDB.

Ephemeral data. Don't store ephemeral data in the database. What this means is data that requires constant rewrites is not ideal for use in relational databases. This includes examples such as `django.contrib.sessions`, `django.contrib.messages`, and `metrics`. Instead, move this data to things like Memcached, Redis, Riak, and other non-relational stores.

TIP: Frank Wiles on Binary Data in Databases

Actually, Frank says that there three things to never store in a database, the third item being binary data. Storage of binary data in databases is addressed by `django.db.models.FileField`, which does the work of storing files on file servers like AWS CloudFront or S3 for you.

20.3.2 Getting the Most Out of PostgreSQL

If using **PostgreSQL**, be certain that it is set up correctly in production. As this is outside the scope of the book, we recommend the following articles:

- http://wiki.postgresql.org/wiki/Detailed_installation_guides
- http://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server
- <http://www.revsys.com/writings/postgresql-performance.html>
- <http://2scoops.co/craig-postgresql-perf>
- <http://2scoops.co/craig-postgresql-perf2>

For further information, you may want to read the book “*PostgreSQL 9.0 High Performance*”: <http://2scoops.co/high-perf-postgresql>

20.3.3 Getting the Most Out of MySQL

It’s easy to get **MySQL** running, but optimizing production installations requires experience and understanding. As this is outside the scope of this book, we recommend the following books by MySQL experts to help you:

- “*High Performance MySQL*” <http://2scoops.co/high-perf-mysql>

20.4 Cache Queries With Memcached or Redis

You can get a lot of mileage out of simply setting up Django’s built-in caching system with Memcached or Redis. You will have to install one of these tools, install a package that provides Python bindings for them, and configure your project.

You can easily set up the per-site cache, or you can cache the output of individual views or template fragments. You can also use Django's low-level cache API to cache Python objects.

Reference material:

- <https://docs.djangoproject.com/en/1.5/topics/cache/>
- <https://github.com/sebleier/django-redis-cache/>

20.5 Identify Specific Places to Cache

Deciding where to cache is like being first in a long line of impatient customers at Ben and Jerry's on free scoop day. You are under pressure to make a quick decision without being able to see what any of the flavors actually look like.

Here are things to think about:

- Which views/templates contain the most queries?
- Which URLs are being requested the most?
- When should a cache for a page be invalidated?

Let's go over the tools that will help you with these scenarios.

20.6 Consider Third-Party Caching Packages

Third-party packages will give you additional features such as:

- Caching of QuerySets.
- Cache invalidation settings/mechanisms.
- Different caching backends.
- Alternative or experimental approaches to caching.

A couple of the popular Django packages for caching are:

- `django-cache-machine`
- `johnny-cache`

See <http://www.djangopackages.com/grids/g/caching/> for more options.

WARNING: Third-Party Caching Libraries Aren't Always the Answer

Having tried many of the third-party Django cache libraries we have to ask our readers to test them very carefully and be prepared to drop them. They are cheap, quick wins, but can lead to some hair-raising debugging efforts at the worst possible times. Cache invalidation is hard, and in our experience magical cache libraries are better for projects with more static content. By-hand caching is a lot more work, but leads to better performance in the long run and doesn't risk those terrifying moments.

20.7 Compression and Minification of HTML, CSS, and JavaScript

When a browser renders a web page, it usually has to load HTML, CSS, JavaScript, and image files. Each of these files consumes the user's bandwidth, slowing down page loads. One way to reduce bandwidth consumption is via compression and minification. Django even provides tools for you: `GZipMiddleware` and the `{% spaceless %}` template tag. Through the at-large Python community, we can even use **WSGI** middleware that performs the same task.

The problem with making Django and Python do the work is that compression and minification take up system resources, which can create bottlenecks of their own. A better approach is to use **Apache** and **Nginx** web servers configured to compress the outgoing content. If you are maintaining your own web servers, this is absolutely the way to go.

A very common middle approach that we endorse is to use a third-party Django library to compress and minify the CSS and JavaScript in advance. Our preference is `django-pipeline` which comes recommended by Django core developer Jannis Leidel.

Tools and libraries to reference:

- Apache and Nginx compression modules
- `django-pipeline`
- `django-htmlmin`
- Django's built-in spaceless tag: <http://2scoops.co/1.5-spaceless-tag>

- Django's included GZip middleware: <http://2scoops.co/1.5-gzip-middleware>
- <http://www.djangopackages.com/grids/g/asset-managers/>

20.8 Use Upstream Caching or a Content Delivery Network

Upstream caches such as **Varnish** are very useful. They run in front of your web server and speed up web page or content serving significantly. See <http://varnish-cache.org/>.

Content Delivery Networks (CDNs) like Akamai and Amazon Cloudfront serve static media such as images, video, CSS, and JavaScript files. They usually have servers all over the world, which serve out your static content from the nearest location. Using a CDN rather than serving static content from your application servers can speed up your projects.

20.9 Other Resources

Advanced techniques on scaling, performance, tuning, and optimization are beyond the scope of this book, but here are some starting points.

On general best practices for web performance:

- YSlow's *Web Performance Best Practices and Rules*:
<http://developer.yahoo.com/yslow/>
- Google's *Web Performance Best Practices*:
https://developers.google.com/speed/docs/best-practices/rules_intro

On scaling large Django sites:

- "Django Performance Tips" article by Jacob Kaplan-Moss:
<http://jacobian.org/writing/django-performance-tips/>
- David Cramer often writes and speaks about scaling Django at Disqus. Read his blog and keep an eye out for his talks, Quora posts, comments, etc. <http://justcramer.com/>
- Watch videos and slides from past DjangoCons and PyCons about different developers' experiences. Scaling practices vary from year to year and from company to company:
<http://lanyrd.com/search/?q=django+scaling>

20.10 Summary

In this chapter we explored a number of bottleneck reduction strategies including:

- Whether you should even care about bottlenecks in the first place
- Profiling your pages and queries
- Optimizing queries
- Using your database wisely
- Caching queries
- Identifying what needs to be cached
- Compression of HTML, CSS, and JavaScript
- Exploring other resources

In the next chapter, we'll go over the basics of securing Django projects.

21 | Security Best Practices

When it comes to security, Django has a pretty good record. This is due to security tools provided by Django, solid documentation on the subject of security, and a thoughtful team of core developers who are extremely responsive to security issues. However, it's up to individual Django developers such as ourselves to understand how to properly secure Django-powered applications.

This chapter contains a list of things helpful for securing your Django application. This list is by no means complete. Consider it a starting point.

21.1 Harden Your Servers

Search online for instructions and checklists for server hardening. Server hardening measures include but are not limited to things like changing your SSH port and disabling/removing unnecessary services.

21.2 Know Django's Security Features

Django 1.5's security features include:

- Cross-site scripting (XSS) protection
- Cross-site request forgery (CSRF) protection
- SQL injection protection
- Clickjacking protection
- Support for SSL/HTTPS, including secure cookies
- Validation of files uploaded by users
- Secure password storage, using the PBKDF2 algorithm with a SHA256 hash by default

- Automatic HTML escaping

Most of Django’s security features “just work” out of the box without additional configuration, but there are certain things that you’ll need to configure. We’ve highlighted some of these details in this chapter, but please make sure that you read the official Django documentation on security as well: <https://docs.djangoproject.com/en/1.5/topics/security/>

21.3 Turn Off DEBUG Mode in Production

Your production site should not be running in DEBUG mode. Attackers can find out more than they need to know about your production setup from a helpful DEBUG mode stack trace page. For more information, see <https://docs.djangoproject.com/en/1.5/ref/settings/#debug>.

21.4 Keep Your Secret Keys Secret

If your SECRET_KEY setting is not secret, this means you risk everything from remote code execution to password hacking. Your API keys and other secrets should be carefully guarded as well. These keys should not even be kept in version control.

We cover the mechanics of how to keep your SECRET_KEY out of version control in [chapter 5](#), *Settings and Requirements Files*, [section 5.3](#), ‘Keep Secret Keys Out With Environment Variables.’

21.5 HTTPS Everywhere

It is always better to deploy a site behind HTTPS. Not having HTTPS means that malicious network users can sniff authentication credentials between your site and end users. In fact, all data sent between your site and end users is up for grabs.

Your entire site should be behind HTTPS. Your site’s static resources should also be served via HTTPS, otherwise visitors will get warnings about “insecure resources” which could scare them away from your site.

TIP: Jacob Kaplan-Moss on HTTPS vs HTTP

Django co-leader Jacob Kaplan-Moss says, “Your whole site should only be available via HTTPS, not HTTP at all. This prevents getting “firesheeped” (having a session cookie stolen when served over HTTP). The cost is usually minimal.”

If visitors try to access your site via HTTP, they should be redirected to HTTPS. This can be done either through configuration of your web server or with Django middleware. Performance-wise, it’s better to do this at the web server level, but if you don’t have control over your web server settings for this, then redirecting via Django middleware is fine.

PACKAGE TIP: Django Middleware Packages That Force HTTPS

Two packages that force HTTPS/SSL across your entire site through Django middleware:

- django-sslify <https://github.com/rdegges/django-sslify>
- django-secure <https://github.com/carljm/django-secure>

The difference is that django-sslify is more minimalist and does nothing but force HTTPS, whereas django-secure also helps you configure and check other security settings.

You should purchase an SSL certificate from a reputable source rather than creating a self-signed certificate. To set it up, follow the instructions for your particular web server or platform-as-a-service.

21.5.1 Use Secure Cookies

Your site’s cookies should also be served over HTTPS. You’ll need to set the following in your settings:

EXAMPLE 21.1

```
SESSION_COOKIE_SECURE = True
CSRF_COOKIE_SECURE = True
```

Read <https://docs.djangoproject.com/en/1.5/topics/security/#ssl-https> for more details.

21.5.2 Use HTTP Strict Transport Security (HSTS)

HSTS is usually configured at the web server level. Follow the instructions for your web server or platform-as-a-service.

If you have set up your own web servers, Wikipedia has sample HSTS configuration snippets that you can use: https://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security

When you enable HSTS, your site's web pages include a HTTP header that tells HSTS-compliant browsers to only connect to the site via secure connections:

- HSTS-compliant browsers will redirect HTTP links to HTTPS.
- If a secure connection isn't possible (e.g. the certificate is self-signed or expired), an error message will be shown and access will be disallowed.

To give you a better idea of how this works, here's an example of what a HTTP Strict Transport Security response header might look like:

EXAMPLE 21.2

```
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

Some HSTS configuration advice:

- ❶ You should use HSTS' `includeSubDomains` mode if you can. This prevents attacks involving using non-secured subdomains to write cookies for the parent domain.
- ❷ You should also set `max-age` to a large value like 31536000 (12 months) or 63072000 (24 months) if you can, but keep in mind that once you set it, you can't unset it.

WARNING: Choose Your HSTS Policy Duration Carefully

Remember that HSTS is a one-way switch. It's a declaration that for the next N seconds, your site will be HTTPS-only. Don't set a HSTS policy with a `max-age` longer than you are able to maintain. Browsers do not offer an easy way to unset it.

Note that HSTS should be enabled *in addition* to redirecting all pages to HTTPS as described earlier.

21.6 Use Django 1.5's Allowed Hosts Validation

As of Django 1.5, you must set `ALLOWED_HOSTS` in your settings to a list of allowed host/domain names. This is a security measure to prevent use of fake HTTP Host headers to submit requests.

We recommend that you avoid setting wildcard values here. For more information, read the Django documentation on `ALLOWED_HOSTS` and the `get_host()` method:

- <http://2scoops.co/1.5-allowed-hosts>
- http://2scoops.co/1.5-get_host

21.7 Always Use CSRF Protection With HTTP Forms That Modify Data

Django comes with cross-site request forgery protection (CSRF) built in, and usage of it is actually introduced in Part 4 of the Django introductory tutorial. It's easy to use, and Django even throws a friendly warning during development when you forget to use it.

In our experience, the only use case for turning off CSRF protection across a site is for creating machine-accessible APIs. API frameworks such as `django-tastypie` and `django-rest-framework` do this for you. If you are writing an API from scratch that accepts data changes, it's a good idea to become familiar with Django's CSRF documentation at <https://docs.djangoproject.com/en/1.5/ref/contrib/csrf/>.

TIP: HTML Search Forms

Since HTML search forms don't change data, they use the HTTP GET method and do not trigger Django's CSRF protection.

You should use Django's `CsrfViewMiddleware` as blanket protection across your site rather than manually decorating views with `csrf_protect`.

21.7.1 Posting Data via AJAX

You should use Django's CSRF protection even when posting data via AJAX. Do not make your AJAX views CSRF-exempt.

Instead, when posting via AJAX, you'll need to set an HTTP header called **X-CSRFToken**.

The official Django documentation includes a snippet that shows how to set this header for only POST requests, in conjunction with jQuery 1.5 or higher's cross-domain checking: <https://docs.djangoproject.com/en/1.5/ref/contrib/csrf/#ajax>

See our complete example of how to use this snippet in practice in [section 11.4](#), AJAX and the CSRF Token, in [chapter 11](#), Building REST APIs in Django.

Recommended reading:

- <https://docs.djangoproject.com/en/1.5/ref/contrib/csrf/>

21.8 Prevent Against Cross-Site Scripting (XSS) Attacks

XSS attacks usually occur when users enter malignant JavaScript that is then rendered into a template directly. This isn't the only method, but it is the most common. Fortunately for us, Django by default escapes a lot of specific characters meaning most attacks fail.

However, Django gives developers the ability to mark content strings as safe, meaning that Django's own safeguards are taken away. Also, if you allow users to set individual attributes of HTML tags, that gives them a venue for injecting malignant JavaScript.

There are other avenues of attack that can occur, so educating yourself is important.

Additional reading:

- <http://2scoops.co/1.5-docs-on-html-scraping>
- http://en.wikipedia.org/wiki/Cross-site_scripting

21.9 Defend Against Python Code Injection Attacks

We once were hired to help with a project that had some security issues. The requests coming into the site were being converted from `django.http.HttpRequest` objects directly into strings via creative use of the `str()` function, then saved to a database table. Periodically, these archived Django requests would be taken from the database and converted into Python dicts via the `eval()` function. This meant that arbitrary Python code could be run on the site at any time.

Needless to say, upon discovery the critical security flaw was quickly removed. This just goes to show that no matter how secure Python and Django might be, we always need to be aware that certain practices are incredibly dangerous.

21.9.1 Python Built-ins That Execute Code

Beware of the `eval()`, `exec()`, and `execfile()` built-ins. If your project allows arbitrary strings or files to be passed into any of these functions, you are leaving your system open to attack.

For more information, read “Eval really is dangerous” by Ned Batchelder: http://nedbatchelder.com/blog/201206/eval_really_is_dangerous.html

21.9.2 Python Standard Library Modules That Can Execute Code

“Never unpickle data received from an untrusted or unauthenticated source.”

– <http://docs.python.org/2/library/pickle.html>

You should not use the Python standard library’s `pickle` module to serialize and deserialize anything that might contain user input. For more information, read “Why Python Pickle is Insecure” by Nadia Alramli: <http://nadiana.com/python-pickle-insecure>.

21.9.3 Third-Party Libraries That Can Execute Code

When using PyYAML, only use `safe_load()`. While the use of **YAML** in the Python and Django communities is rare, it’s not uncommon to receive this format from other services. Therefore, if you are accepting YAML documents, only load them with the `yaml.safe_load()` method.

For reference, the `yaml.load()` method will let you create Python objects, *which is really bad*. As Ned Batchelder says, `yaml.load()` should be renamed to `yaml.dangerous_load()`: http://nedbatchelder.com/blog/201302/war_is_peace.html

21.10 Validate All User Input With Django Forms

Django's forms are a wonderful framework designed to validate Python dictionaries. While most of the time we use them to validate incoming HTTP requests containing POST, there is nothing limiting them to be used in this manner.

For example, let's say we have a Django app that updates its model via CSV files fetched from another project. To handle this sort of thing, it's not uncommon to see code like this (albeit in not as simplistic an example):

BAD EXAMPLE 21.1

```
import csv
import StringIO

from .models import Purchase

def add_csv_purchases(rows):

    rows = StringIO.StringIO(rows)
    records_added = 0

    # Generate a dict per row, with the first CSV row being the keys
    for row in csv.DictReader(rows, delimiter=","):
        # DON'T DO THIS: Tossing unvalidated data into your model.
        Purchase.objects.create(**row)
        records_added += 1
    return records_added
```

In fact, what you don't see is that we're not checking to see if sellers, stored as a string in the `Purchase` model, are actually valid sellers. We could add validation code to our `add_csv_purchases()` function, but let's face it, keeping complex validation code understandable as requirements and data changes over time is hard.

A better approach is to validate the incoming data with a Django Form like so:

EXAMPLE 21.3

```
import csv
import StringIO

from django import forms

from .models import Purchase, Seller

class PurchaseForm(forms.ModelForm):

    class Meta:
        model = Purchase

    def clean_seller(self):
        seller = self.cleaned_data["seller"]
        try:
            Seller.objects.get(name=seller)
        except Seller.DoesNotExist:
            msg = "{0} does not exist in purchase #{1}.".format(
                seller,
                self.cleaned_data["purchase_number"]
            )
            raise forms.ValidationError(msg)
        return seller

def add_csv_purchases(rows):

    rows = StringIO.StringIO(rows)

    records_added = 0
    errors = []
    # Generate a dict per row, with the first CSV row being the keys.
    for row in csv.DictReader(rows, delimiter=","):

        # Bind the row data to the PurchaseForm.
```

```
form = PurchaseForm(row)
# Check to see if the row data is valid.
if form.is_valid():
    # Row data is valid so save the record.
    form.save()
    records_added += 1
else:
    errors.append(form.errors)

return records_added, errors
```

What’s really nice about this practice is that rather than cooking up our own validation system for incoming data, we’re using the well-proven data testing framework built into Django.

21.11 Handle User-Uploaded Files Carefully

Django has two model fields that allow for user uploads: `FileField` and `ImageField`. They come with some built-in validation, but the Django docs also strongly advise you to “pay close attention to where you’re uploading them and what type of files they are, to avoid security holes.”

If you are only accepting uploads of certain file types, do whatever you can do to ensure that the user is only uploading files of those types. For example, you can:

- Use the **python-magic** library to check the uploaded file’s headers: <https://github.com/ahupp/python-magic>
- Validate the file with a Python library that specifically works with that file type. Unfortunately this isn’t documented, but if you dig through Django’s `ImageField` source code, you can see how Django uses PIL to validate that uploaded image files are in fact images.

WARNING: Custom Validators Aren't the Answer Here

Don't just write a custom validator and expect it to validate your uploaded files before dangerous things happen. Custom validators are run against field content after they've already been coerced to Python by the field's `to_python()` method.

If the contents of an uploaded file are malicious, any validation happening after `to_python()` is executed may be too late.

Uploaded files must be saved to a directory that does not allow them to be executed. Take extra care with your web server's configuration here, because a malicious user can try to attack your site by uploading an executable file like a CGI or PHP script and then accessing the URL.

Consult your web server's documentation for instructions on how to configure this, or consult the documentation for your platform-as-a-service for details about how static assets and user-uploaded files should be stored.

Further reading:

- <https://docs.djangoproject.com/en/1.5/ref/models/fields/#filefield>

21.12 Don't Use `ModelForms.Meta.exclude`

When using `ModelForms`, always use `Meta.fields`. Never use `Meta.exclude`. The use of `Meta.exclude` is considered a grave security risk. *We can't stress this strongly enough. Don't do it.*

One common reason we want to avoid the `Meta.exclude` attribute is that its behavior implicitly allows all model fields to be changed except for those that we specify. When using the `excludes` attribute, if the model changes after the form is written, we have to remember to change the form. If we forget to change the form to match the model changes, we risk catastrophe.

Let's use an example to show how this mistake could be made. We'll start with a simple ice cream store model:

EXAMPLE 21.4

```
# stores/models.py
from django.conf import settings
```

```
from django.db import models

class Store(models.Model):
    title = models.CharField(max_length=255)
    slug = models.SlugField()
    owner = models.ForeignKey(settings.AUTH_USER_MODEL)
    # Assume 10 more fields that cover address and contact info.
```

Here is the *wrong way* to define the ModelForm fields for this model:

```
BAD EXAMPLE 21.2

# DON'T DO THIS!
from django import forms

from .models import Store

class StoreForm(forms.ModelForm):

    class Meta:
        model = Store
        # DON'T DO THIS: Implicit definition of fields.
        # Too easy to make mistakes!
        excludes = ("pk", "slug", "modified", "created", "owner")
```

In contrast, this is the *right way* to define the same ModelForm's fields:

```
EXAMPLE 21.5

from django import forms

from .models import Store

class StoreForm(forms.ModelForm):

    class Meta:
        model = Store
```

```
# Explicitly specifying the fields we want
fields = (
    "title", "address_1", "address_2", "email",
    "usstate", "postal_code", "city",
)
```

The first code example, as it involves less typing, appears to be the better choice. It's not, as when you add a new model field you now you need to track the field in multiple locations (one model and one or more forms).

Let's demonstrate this in action. Perhaps after launch we decide we need to have a way of tracking store co-owners, who have all the same rights as the owner. They can access account information, change passwords, place orders, and specify banking information. The store model receives a new field as shown on the next page:

EXAMPLE 21.6

```
# stores/models.py
from django.conf import settings
from django.db import models

class Store(models.Model):
    title = models.CharField(max_length=255)
    slug = models.SlugField()
    owner = models.ForeignKey(settings.AUTH_USER_MODEL)
    co_owners = models.ManyToManyField(settings.AUTH_USER_MODEL)
    # Assume 10 more fields that cover address and contact info.
```

The first form code example which we warned against using relies on us to remember to alter it to include the new `co_owners` field. If we forget, then anyone accessing that store's HTML form can add or remove co-owners. While we might remember a single form, what if we have more than one `ModelForm` for a model? In complex applications this is not uncommon.

On the other hand, in the second example, where we used `Meta.fields` we know exactly what fields each form is designed to handle. Changing the model doesn't alter what the form exposes, and we can sleep soundly knowing that our ice cream store data is more secure.

21.13 Beware of SQL Injection Attacks

The Django ORM generates properly-escaped SQL which will protect your site from users attempting to execute malignant, arbitrary SQL code.

Django allows you to bypass its ORM and access the database more directly through raw SQL. When using this feature, be especially careful to escape your SQL code properly.

21.14 Never Store Credit Card Data

Unless you have a strong understanding of the PCI-DSS security standards (<https://www.pcisecuritystandards.org/>) and adequate time/resources/funds to validate your PCI compliance, storing credit card data is too much of a liability and should be avoided.

Instead, we recommend using third-party services like Stripe, Balanced Payments, PayPal, and others that handle storing this information for you, and allow you to reference the data via special tokens. Most of these services have great tutorials, are very Python and Django friendly, and are well worth the time and effort to incorporate into your project.

TIP: Read the Source Code of Open Source E-Commerce Solutions

If you are planning to use one of the existing open source Django e-commerce solutions, examine how the solution handles payments. If credit card data is being stored in the database, even encrypted, then please consider using another solution.

21.15 Secure the Django Admin

Since the Django admin gives your site admins special powers that ordinary users don't have, it's good practice to make it extra secure.

21.15.1 Change the Default Admin URL

By default, the admin URL is *yoursite.com/admin/*. Change it to something that's long and difficult to guess.

TIP: Jacob Kaplan-Moss Talks About Changing the Admin URL

Django project co-leader Jacob Kaplan-Moss says (paraphrased) that it's an easy additional layer of security to come up with a different name (or even different domain) for the admin.

It also prevents attackers from easily profiling your site. For example, attackers can tell which version of Django you're using, sometimes down to the point-release level, by examining the content of *admin/* on a project.

21.15.2 Use django-admin-honeypot

If you're particularly concerned about people trying to break into your Django site, *django-admin-honeypot* is a package that puts a fake Django admin login screen at *admin/* and logs information about anyone who attempts to log in.

See <https://github.com/dmpayton/django-admin-honeypot> for more information.

21.15.3 Only Allow Admin Access via HTTPS

This is already implied in the “Use SSL/HTTPS in Production” section, but we want to especially emphasize here that your admin needs to be SSL-secured. If your site allows straight HTTP access, you will need to run the admin on a properly secured domain, adding to the complexity of your deployment. Not only will you need a second deployment procedure, but you'll need to include logic in your URLConf in order to remove the admin from HTTP access. In the experience of the authors, it's much easier to put the whole site on SSL/HTTPS.

Without SSL, if you log into your Django admin on an open WiFi network, it's trivial for someone to sniff your admin username/password.

21.15.4 Limit Admin Access Based on IP

Configure your web server to only allow access to the Django admin to certain IP addresses. Look up the instructions for your particular web server.

An acceptable alternative is to put this logic into middleware. It's better to do it at the web server level because every middleware component adds an extra layer of logic wrapping your views, but in some cases this can be your only option. For example, your platform-as-a-service might not give you fine-grain control over web server configuration.

21.15.5 Use the `allow_tags` Attribute With Caution

The `allow_tags` attribute, which is set to `False` by default, can be a security issue. What `allow_tags` does is when set to `True` is allow HTML tags to be displayed in the admin.

Our hard rule is `allow_tags` can only be used on system generated data like primary keys, dates, and calculated values. Data such as character and text fields are completely out, as is any other user entered data.

21.16 Secure the Admin Docs

Since the Django admin docs give your site admins a view into how the project is constructed, it's good practice to keep them extra secure just like the Django admin. Borrowing from the previous section on the Django admin, we advocate the following:

- Changing the admin docs URL to something besides *yoursite.com/admin/*.
- Only allowing admin docs access via HTTPS.
- Limiting admin docs access based on IP.

21.17 Monitor Your Sites

Check your web servers' access and error logs regularly. Install monitoring tools and check on them frequently. Keep an eye out for suspicious activity.

21.18 Keep Your Dependencies Up-to-Date

You should always update your projects to work with the latest stable release of Django. This is particularly important when a release includes security fixes. Subscribe to the official Django weblog (<https://www.djangoproject.com/weblog/>) and keep an eye out for updates and security information.

It's also good to keep your dependencies up-to-date, and to watch for important security announcements relating to them.

21.19 Put Up a Vulnerability Reporting Page

It's a good idea to publish information on your site about how users can report security vulnerabilities to you.

GitHub's "Responsible Disclosure of Security Vulnerabilities" page is a good example of this and rewards reporters of issues by publishing their names: <https://help.github.com/articles/responsible-disclosure-of-security-vulnerabilities>

21.20 Keep Up-to-Date on General Security Practices

We end this chapter with a common-sense warning.

First, keep in mind that security practices are constantly evolving, both in the Django community and beyond. Check Twitter, Hacker News (<http://news.ycombinator.com/>), and various security blogs regularly.

Second, remember that security best practices extend well beyond those practices specific to Django. You should research the security issues of every part of your web application stack, and you should follow the corresponding sources to stay up to date.

TIP: Good Books and Articles on Security

Paul McMillan, Django core developer and security expert, recommends the following books:

- “*The Tangled Web: A Guide to Securing Modern Web Applications*”:
<http://2scoops.co/book-tangled-web>
- “*The Web Application Hacker’s Handbook*”: <http://2scoops.co/book-web-app-hackers>

In addition, we recommend the following reference sites:

- <https://code.google.com/p/browsersec/wiki/Main>
- https://wiki.mozilla.org/WebAppSec/Secure_Coding_Guidelines

21.21 Summary

Please use this chapter as a starting point for Django security, not the ultimate reference guide. See the Django documentation’s list for additional security topics:

<http://2scoops.co/1.5-additional-security-topics>

Django comes with a good security record due to the diligence of its community and attention to detail. Security is one of those areas where it’s a particularly good idea to ask for help. If you find yourself confused about anything, ask questions and turn to others in the Django community for help.

22 | Logging: What's It For, Anyway?

Logging is like rocky road ice cream. Either you can't live without it, or you forget about it and wonder once in awhile why it exists.

Anyone who's ever worked on a large production project with intense demands understands the importance of using the different log levels appropriately, creating module-specific loggers, meticulously logging information about important events, and including extra detail about the application's state when those events are logged.

While logging might not seem glamorous, remember that it is one of the secrets to building extremely stable, robust web applications that scale and handle unusual loads gracefully. Logging can be used not only to debug application errors, but also to track interesting performance metrics.

Logging unusual activity and checking logs regularly is also important for ensuring the security of your server. In the previous chapter, we covered the importance of checking your server access and error logs regularly. Keep in mind that application logs can be used in similar ways, whether to track failed login attempts or unusual application-level activity.

22.1 Application Logs vs. Other Logs

This chapter focuses on application logs. Any log file containing data logged from your Python web application is considered an application log.

In addition to your application logs, you should be aware that there are other types of logs, and that using and checking all of your server logs is necessary. Your server logs, database logs, network logs, etc. all provide vital insight into your production system, so consider them all equally important.

22.2 Why Bother With Logging?

Logging is your go-to tool in situations where a stack trace and existing debugging tools aren't enough. Whenever you have different moving parts interacting with each other or the possibility of unpredictable situations, logging gives you insight into what's going on.

The different log levels available to you are `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL`. Let's now explore when it's appropriate to use each logging level.

22.3 When to Use Each Log Level

In places other than your production environment, you might as well use all the log levels. Log levels are controlled in your project's settings modules, so we can fine tune this recommendation as needed to account for load testing and large scale user tests.

In your production environment, we recommend using every log level except for `DEBUG`.

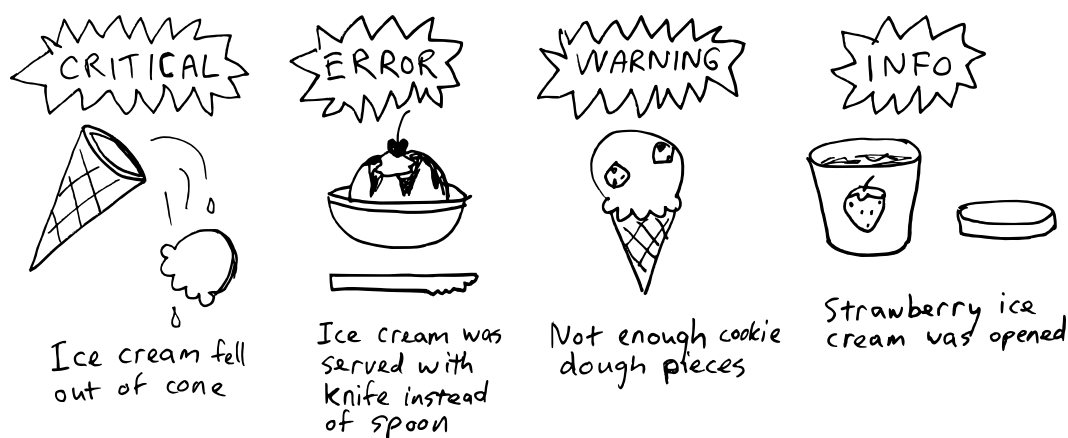


Figure 22.1: Appropriate usage of `CRITICAL`/`ERROR`/`WARNING`/`INFO` logging in ice cream.

Since the same `CRITICAL`, `ERROR`, `WARNING`, and `INFO` logs are captured whether in production or development, introspection of buggy code requires less modification of code. This is important to remember, as debug code added by developers working to fix one problem can create new ones.

The rest of this section covers how each log level is used.

22.3.1 Log Catastrophes With CRITICAL

Use the CRITICAL log level only when something catastrophic occurs that requires urgent attention.

For example, if your code relies on an internal web service being available, and if that web service is part of your site's core functionality, then you might log at the CRITICAL level anytime that the web service is inaccessible.

This log level is never used in core Django code, but you should certainly use it in your code anywhere that an extremely serious problem can occur.

22.3.2 Log Production Errors With ERROR

Let's look at core Django for an example of when ERROR level logging is appropriate. In core Django, the ERROR log level is used very sparingly. There is one very important place where it is used: whenever code raises an exception that is not caught, the event gets logged by Django using the following code:

EXAMPLE 22.1

```
# Taken directly from core Django code.
# Used here to illustrate an example only, so don't
# copy this into your project.
logger.error("Internal Server Error: %s", request.path,
            exc_info=exc_info,
            extra={
                "status_code": 500,
                "request": request
            }
        )
```

How does Django put this to good use? Well, when `DEBUG=False` is in your settings, everyone listed in `ADMINS` immediately gets emailed the following:

- A description of the error
- A complete Python traceback from where the error occurred
- Information about the HTTP request that caused the error

If you've ever received one of those email notifications, you know how useful `ERROR` logs are when you need them most.

Similarly, we recommend that you use the `ERROR` log level whenever you need to log an error that is worthy of being emailed to you or your site admins. When your code catches the exception, log as much information as you can to be able to resolve the problem.

For example, an exception may be thrown when one of your views cannot access a needed third-party API. When the exception is caught, you can log a helpful message and the API's failure response, if any.

22.3.3 Log Lower-Priority Problems With `WARNING`

This level is good for logging events that are unusual and potentially bad, but not as bad as `ERROR`-level events.

For example, if you are using `django-admin-honeypot` to set up a fake *admin/* login form, you might want to log intruders' login attempts to this level.

Django uses the log level in several parts of `CsrfViewMiddleware`, to log events that result in a **403 Forbidden** error. For example, when an incoming `POST` request is missing its `csrf_token`, the event gets logged as follows:

EXAMPLE 22.2

```
# Taken directly from core Django code.
# Used here to illustrate an example only, so don't
# copy this into your project.
logger.warning("Forbidden (%s): %s",
               REASON_NO_CSRF_COOKIE, request.path,
               extra={
                   "status_code": 403,
                   "request": request,
               })
```

22.3.4 Log Useful State Information With INFO

We recommend using this level to log any details that may be particularly important when analysis is needed. These include:

- Startup and shutdown of important components not logged elsewhere
- State changes that occur in response to important events
- Changes to permissions, e.g. when users are granted admin access

In addition to this, the INFO level is great for logging any general information that may help in performance analysis. It's a good level to use while hunting down problematic bottlenecks in your application and doing profiling.

22.3.5 Log Debug-Related Messages to DEBUG

In development, we recommend using DEBUG and occasionally INFO level logging wherever you'd consider throwing a `print` statement into your code for debugging purposes.

Getting used to logging this way isn't hard. Instead of this:

BAD EXAMPLE 22.1

```
from django.views.generic import TemplateView

from .helpers import pint_counter

class PintView(TemplateView):

    def get_context_data(self, *args, **kwargs):
        context = super(PintView, self).get_context_data(**kwargs)
        pints_remaining = pint_counter()
        print("Only %d pints of ice cream left." % (pints_remaining))
        return context
```

We do this:

EXAMPLE 22.3

```
import logging

from django.views.generic import TemplateView

from .helpers import pint_counter

logger = logging.getLogger(__name__)

class PintView(TemplateView):

    def get_context_data(self, *args, **kwargs):
        context = super(PintView, self).get_context_data(**kwargs)
        pints_remaining = pint_counter()
        logger.debug("Only %d pints of ice cream left." % pints_remaining)
        return context
```

Sprinkling `print` statements across your projects results in problems and technical debt:

- Depending on the web server, a forgotten `print` statement can bring your site down.
- *Print statements are not recorded.* If you don't see them, then you miss what they were trying to say.
- When it's time for the Django world to migrate to Python 3, old-style `print` statements like `print IceCream.objects.flavor()` will break your code.

Unlike `print` statements, logging allows different report levels and different response methods. This means that:

- We can write `DEBUG` level statements, leave them in our code, and never have to worry about them doing anything when we move code to production.
- The response method can provide the response as email, log files, console and `stdout`. It can even report as pushed HTTP requests to applications such as *Sentry*!

Note that there's no need to go overboard with debug-level logging. It's great to add `logging.debug()` statements while you're debugging, but there's no need to clutter your code with logging every single line.



Figure 22.2: Appropriate usage of DEBUG logging in ice cream.

22.4 Log Tracebacks When Catching Exceptions

Whenever you log an exception, it's usually helpful to log the stack trace of the exception. Python's logging module supports this:

- ❶ `Logger.exception()` automatically includes the traceback and logs at ERROR level.
- ❷ For other log levels, use the optional `exc_info` keyword argument.

Here's an example of adding a traceback to a WARNING level log message:

EXAMPLE 22.4

```
import logging
import requests

logger = logging.getLogger(__name__)

def get_additional_data():
    try:
        r = requests.get("http://example.com/something-optional/")
    except requests.HTTPError, e:
```

```
logger.exception(e)
logger.debug("Could not get additional data", exc_info=True)
return r
```

22.5 One Logger Per Module That Uses Logging

Whenever you use logging in another module, don't import and reuse a logger from elsewhere. Instead, define a new logger specific to the module like this:

```
EXAMPLE 22.5
# You can place this snippet at the top
# of models.py, views.py, or any other
# file where you need to log.
import logging

logger = logging.getLogger(__name__)
```

What this gives you is the ability to turn on and off only the specific loggers that you currently need. If you're running into a strange issue in production that you can't replicate locally, you can temporarily turn on DEBUG logging for just the module related to the issue. Then, when you identify the problem, you can turn that logger back off in production.

22.6 Log Locally to Rotating Files

When you create a new Django project with `startproject`, your default settings file is configured to email ERROR and higher log messages to whomever you list in `ADMINS`. This occurs via a handler called `AdminEmailHandler` that comes with Django.

In addition to this, we recommend also writing logs of level INFO and higher to rotating log files on disk. On-disk log files are helpful in case the network goes down or emails can't be sent to your admins for some reason. Log rotation keeps your logs from growing to fill your available disk space.

A common way to set up log rotation is to use the UNIX **logrotate** utility with `logging.handlers.WatchedFileHandler`.

Note that if you are using a platform-as-a-service, you might not be able to set up rotating log files. In this case, you may need to use an external logging service such as Loggly (<http://loggly.com/>).

22.7 Other Logging Tips

- Control the logging in settings files per the Django documentation on logging: <https://docs.djangoproject.com/en/1.5/topics/logging/>
- While debugging, use the Python logger at DEBUG level.
- After running tests at DEBUG level, try running them at INFO and WARNING levels. The reduction in information you see may help you identify upcoming deprecations for third-party libraries.
- Don't wait until it's too late to add logging. You'll be grateful for your logs if and when your site fails.
- You can do useful things with the emails you receive when ERROR or higher level events occur. For example, you can configure a PagerDuty (<http://www.pagerduty.com/>) account to alert you and your team repeatedly until you've taken action.
- Colorizing your log output helps you spot important messages at a glance. This blog post by **logutils** author Vinay Sajip explains more, with some great screenshots: <http://2scoops.co/colorizing-logging-output>.

PACKAGE TIP: Logutils Provides Useful Handlers

The **logutils** package by Vinay Sajip comes with a number of very interesting logging handlers. Features include:

- Colorizing of console streams under Windows, Linux and Mac OS X.
- The ability to log to queues. Useful in situations where you want to queue up log messages to a slow handler like SMTPHandler.
- Classes that allow you to write unit tests for log messages.
- An enhanced HTTPHandler that supports secure connections over HTTPS.

Some of the more basic features of logutils are so useful that they have been absorbed into the Python standard library!

22.8 Necessary Reading Material

- <https://docs.djangoproject.com/en/1.5/topics/logging/>
- <http://docs.python.org/2/library/logging.html>
- <http://docs.python.org/2/library/logging.config.html>
- <http://docs.python.org/2/library/logging.handlers.html>
- <http://docs.python.org/2/howto/logging-cookbook.html>

22.9 Useful Third-Party Tools

- Sentry (<https://www.getsentry.com/>) aggregates errors for you.
- loggly.com (<http://loggly.com/>) simplifies log management and provides excellent query tools.

22.10 Summary

Django projects can easily take advantage of the rich logging functionality that comes with Python. Combine logging with handlers and analysis tools, and suddenly you have real power. You can use logging to help you improve the stability and performance of your projects.

In the next chapter we'll discuss signals, which become much easier to follow, debug, and understand with the help of logging.

23 | Signals: Use Cases and Avoidance Techniques

The Short Answer: Use signals as a last resort.

The Long Answer: Often when new Django-nauts first discover signals, they get signal-happy. They start sprinkling signals everywhere they can and feeling like real experts at Django.

After coding this way for a while, projects start to turn into confusing, knotted hairballs that can't be untangled. Signals are being dispatched everywhere and hopefully getting received somewhere, but at that point it's hard to tell what exactly is going on.

Many developers also confuse signals with asynchronous message queues such as what Celery (<http://www.celeryproject.org/>) provides. Make no mistake, *signals are synchronous and blocking*, and calling performance heavy processes via signals provide absolutely no benefit from a performance or scaling perspective. In fact, moving such unnecessary processes to signals is considered code obfuscation.

Signals can be useful, but they should be used as a last resort, only when there's no good way to avoid using them.

23.1 When to Use and Avoid Signals

Do not use signals when:

- The signal relates to one particular model and can be moved into one of that model's methods, possibly called by `save()`.

- The signal can be replaced with a custom model manager method.
- The signal relates to a particular view and can be moved into that view.

It might be okay to use signals when:

- Your signal receiver needs to make changes to more than one model.
- You want to dispatch the same signal from multiple apps and have them handled the same way by a common receiver.
- You want to invalidate a cache after a model save.
- You have an unusual scenario that needs a callback, and there's no other way to handle it besides using a signal. For example, you want to trigger something based on the `save()` or `init()` of a third-party app's model. You can't modify the third-party code and extending it might be impossible, so a signal provides a trigger for a callback.

23.2 Signal Avoidance Techniques

Let's go over some scenarios where you can simplify your code and remove some of the signals that you don't need.

23.2.1 Using Custom Model Manager Methods Instead of Signals

Let's imagine that our site handles user-submitted ice cream-themed events, and each ice cream event goes through an approval process. These events are set with a status of "Unreviewed" upon creation. The problem is that we want our site administrators to get an email for each event submission so they know to review and post things quickly.

We could have done this with a signal, but unless we put in extra logic in the `post_save()` code, even administrator created events would generate emails.

An easier way to handle this use case is to create a custom model manager method and use that in your views. This way, if an event is created by an administrator, they don't have to go through the review process.

Since a code example is worth a thousand words, here is how we would create such a method:

EXAMPLE 23.1

```
# events/managers.py
from django.db import models

class EventManager(models.Manager):

    def create_event(self, title, start, end, creator):
        event = self.model(title=title,
                           start=start,
                           end=end,
                           creator=creator)
        event.notify_admins()
        return event
```

Now that we have our custom manager with its custom manager method in place, let's attach it to our model (which comes with a `notify_admins()` method):

EXAMPLE 23.2

```
# events/models.py
from django.conf import settings
from django.core.mail import mail_admins
from django.db import models

from model_utils.models import TimeStampedModel

from .managers import EventManager

class Event(TimeStampedModel):

    STATUS_UNREVIEWED, STATUS_REVIEWED = (0, 1)
    STATUS_CHOICES = (
        (STATUS_UNREVIEWED, "Unreviewed"),
        (STATUS_REVIEWED, "Reviewed"),
    )

    title = models.CharField(max_length=100)
```

```
start = models.DateTimeField()
end = models.DateTimeField()
status = models.IntegerField(choices=STATUS_CHOICES,
                              default=STATUS_UNREVIEWED)
creator = models.ForeignKey(settings.AUTH_USER_MODEL)

objects = EventManager()

def notify_admins(self):
    # create the subject and message
    subject = "{user} submitted a new event!".format(
        user=self.creator.get_full_name())
    message = """"TITLE: {title}
START: {start}
END: {end}""".format(title=self.title, start=self.start,
                      end=self.end)

    # Send to the admins!
    mail_admins(subject=subject,
                message=message,
                fail_silently=False)
```

Using this follows a similar pattern to using the User model. To generate an event, instead of calling `create()`, we call a `create_event()` method.

EXAMPLE 23.3

```
>>> from django.contrib.auth import get_user_model
>>> from django.utils import timezone
>>> from events.models import Event
>>> user = get_user_model().get(username="audreyr")
>>> now = timezone.now()
>>> event = Event.objects.create_event(
...     title="International Ice Cream Tasting Competition",
...     start=now,
...     end=now,
...     user = user
```

```
)
```

23.2.2 Validate Your Model Elsewhere

If you're using a `pre_save` signal to trigger input cleanup for a specific model, try writing a custom validator for your field(s) instead.

If validating through a `ModelForm`, try overriding your model's `clean()` method instead.

23.2.3 Override Your Model's Save or Delete Method Instead

If you're using `pre_save` and `post_save` signals to trigger logic that only applies to one particular model, you might not need those signals. You can often simply move the signal logic into your model's `save()` method.

The same applies to overriding `delete()` instead of using `pre_delete` and `post_delete` signals.

24 | What About Those Random Utilities?

24.1 Create a Core App for Your Utilities

Sometimes we end up writing shared classes or little general-purpose utilities that are useful everywhere. These bits and pieces don't belong in any particular app. We don't just stick them into a sort-of-related random app, because we have a hard time finding them when we need them. We also don't like placing them as “random” modules in the root of the project.

Our way of handling our utilities is to place them into a Django app called *core* that contains modules which contains functions and objects for use across a project. (Other developers follow a similar pattern and call this sort of app *common*, *generic*, *util*, or *utils*.)

For example, perhaps our project has both a custom model manager and a custom view mixin used by several different apps. Our *core* app would therefore look like:

EXAMPLE 24.1

```
core/  
    __init__.py  
    managers.py # contains the custom model manager  
    models.py  
    views.py # Contains the custom view mixin
```

TIP: Django App Boilerplate: The `models.py` Module

Don't forget that in order to make a Python module be considered a Django App, a `models.py` module is required! However, you only need to make the core module a Django app if you need to do one or more of the following:

- Have non-abstract models in *core*.
- Have admin auto-discovery working in *core*.
- Have template tags and filters.

Now, if we want to import our custom model manager and/or view mixin, we import using the same pattern of imports we use for everything else:

EXAMPLE 24.2

```
from core.managers import PublishedManager
from core.views import IceCreamMixin
```

24.2 Django's Own Swiss Army Knife

The Swiss Army Knife is a multi-purpose tool that is compact and useful. Django has a number of useful helper functions that don't have a better home than the `django.utils` package. It's tempting to dig into the code in `django.utils` and start using things, but don't. Most of those modules are designed for internal use and their behavior or inclusion can change between Django version. Instead, read <https://docs.djangoproject.com/en/1.5/ref/utils/> to see which modules in there are stable.

TIP: Malcolm Tredinnick On Django's Utils Package.

Django core developer Malcolm Tredinnick liked to think of `django.utils` as being in the same theme as Batman's utility belt: indispensable tools that are used everywhere internally.

There are some gems in there that have turned into best practices:

24.2.1 `django.contrib.humanize`

This is a set of localized template filters designed to give user presented data a more ‘human’ touch. For example it includes a filter called ‘`intcomma`’ that converts integers to strings containing commas (or periods depending on locale) every three digits. While `django.contrib.humanize`’s filters they are useful for making template output more attractive, you can also import each filter individually as a function. This is quite handy when processing any sort of text, especially when used in conjunction with REST APIs.

24.2.2 `django.utils.html.remove_tags(value, tags)`

When you need to accept content from users and want to strip out a list of tags, this function removes those tags for you while keeping all other content untouched.

24.2.3 `django.utils.html.strip_tags(value)`

When you need to accept content from users and have to strip out anything that could be HTML, this function removes those tags for you while keeping all the existing text between tags.

24.2.4 `django.utils.text.slugify(value)`

Whatever you do, don’t write your own version of the `slugify()` function; as any inconsistency from what Django does with this function will cause subtle yet nasty problems. Instead, use the same function that Django uses and `slugify()` consistently.

PACKAGE TIP: slugify and languages besides English

Our friend Tomasz Paczkowski pointed out that we should note that `slugify()` can cause problems with localization:

EXAMPLE 24.3

```
>>> from django.utils.text import slugify
>>> slugify(u"straße") # German
u"strae"
```

Fortunately, **unicode-slugify**, is a Mozilla foundation supported project that addresses the issue:

EXAMPLE 24.4

```
>>> from slugify import slugify
>>> slugify(u"straße") # Again with German
u"straße"
```

24.2.5 `django.utils.timezone`

It's good practice for you to have time zone support enabled. Chances are that your users live in more than one time zone.

When you use Django's time zone support, date and time information is stored in the database uniformly in UTC format and converted to local time zones as needed.

24.2.6 `django.utils.translation`

Much of the non-English speaking world appreciates use of this tool, as it provides Django's i18n support.

24.3 Summary

We follow the practice of putting often reused files into utility packages. We enjoy being able to remember where we placed our code. Many Python libraries often follow this pattern, and we covered some of the more useful utility functions provided by Django.

25 | Deploying Django Projects

Deployment of Django projects is an in-depth topic that could fill an entire book on its own. Here, we touch upon deployment at a high level.

25.1 Using Your Own Web Servers

You should deploy your Django projects with **WSGI**.

Django 1.5's `startproject` command now sets up a *wsgi.py* file for you. This file contains the default configuration for deploying your Django project to any WSGI server.

The most commonly-used WSGI deployment setups are:

- ❶ **Gunicorn** behind a **Nginx** proxy.
- ❷ **Apache** with **mod_wsgi**.

Here's a quick summary comparing the two.

Setup	Advantages	Disadvantages
Gunicorn (sometimes with Nginx)	Gunicorn is written in pure Python. Supposedly this option has slightly better memory usage, but your mileage may vary. Has built-in Django integration.	Documentation is brief for nginx (but growing). Not as time-tested, so you may run into confusing configuration edge cases and the occasional bug.

Setup	Advantages	Disadvantages
Apache with mod_wsgi	Has been around for a long time and is tried and tested. Very stable. Lots of great documentation, to the point of being kind of overwhelming.	Apache configuration can be overly complex and painful for some. Lots of crazy conf files.

Table 25.1: Gunicorn vs Apache

WARNING: Do not use mod_python

As of June 16th, 2010 the **mod_python** project is officially dead. The last active maintainer of mod_python and the official Django documentation explicitly warns against using mod_python and we concur.

There’s a lot of debate over which option is faster. Don’t trust benchmarks blindly, as many of them are based on serving out tiny “Hello World” pages, which of course will have different performance from your own web application.

Ultimately, though, both choices are in use in various high volume Django sites around the world. Configuration of any high volume production server can be very difficult, and if your site is busy enough it’s worth investing time in learning one of these options very well.

The disadvantage of setting up your own web servers is the added overhead of extra sysadmin work. It’s like making ice cream from scratch rather than just buying and eating it. Sometimes you just want to buy ice cream so that you can focus on the enjoyment of eating it.

25.2 Using a Platform as a Service

If you’re working on a small side project or are a founder of a small startup, you’ll definitely save time by using a **Platform as a Service (PaaS)** instead of setting up your own servers. Even large projects can benefit from the advantages of using them.

First, a public service message:

TIP: Never Get Locked Into a Single Hosting Provider

There are amazing services which will host your code, databases, media assets, and also provide a lot of wonderful accessories services. These services, however, can go through changes that can destroy your project. These changes include crippling price increases, performance degradation, unacceptable terms of service changes, untenable service license agreements, sudden decreases in availability, or can simply go out of business.

This means it's in your best interests to do your best to avoid being forced into architectural decisions based on the needs of your hosting provider. Be ready to be able to move from one provider to another without major restructuring of your project.

We make certain none of our projects are intrinsically tied to any hosting solution, meaning we are not locked into a single vendor.

As a WSGI-compliant framework, Django is supported on a lot of Platform as a Service providers. If you go with a PaaS, choose one that can scale with little or no effort as your traffic/data grows.

The most commonly-used ones as of this writing that specialize in automatic/practically automatic scaling are:

- Heroku (<http://heroku.com>) is a popular option in the Python community because of its wealth of documentation and easy ability to scale. If you choose this option, read <http://www.deploydjango.com/> and <http://www.theherokuhackersguide.com/> by Randall Degges.
- Gondor.io (<http://gondor.io>) - Developed and managed by two Django core developers, James Tauber and Brian Rosner, Gondor.io was designed for people who want to deploy their Python sites early and often.
- DotCloud (<http://dotcloud.com>) is a Python powered Platform as a Service with a sandbox tier that lets you deploy an unlimited number of applications.

TIP: Read the Platform as a Service Documentation

We originally wanted to provide a quick and easy mini-deployment section for each of the services we listed. However, we didn't want this book to become quickly outdated, so instead we ask the reader to follow the deployment instructions listed on each site.

See each of these services' individual documentation for important details about how your requirements files, environment variables, and settings files should be set up when using a Platform as a Service. For example, most of these services insist on the placement of a requirements.txt file in the root of the project.

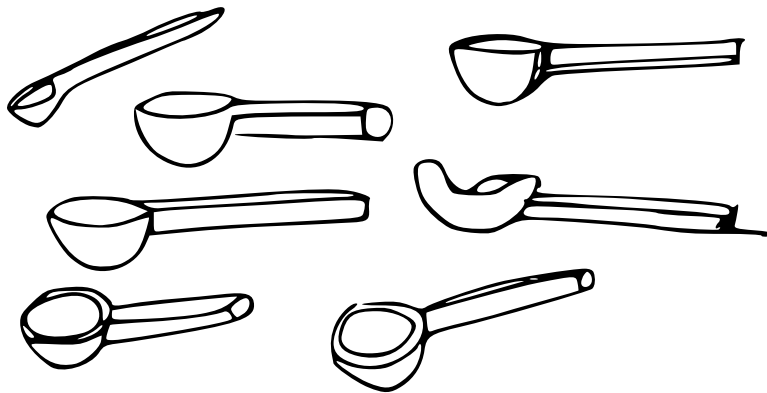


Figure 25.1: How ice cream is deployed to cones and bowls.

25.3 Summary

In this chapter we gave some guidelines and advice for deploying Django projects. We also suggested the use of Platforms as a Service, and also advised not to alter your application structure too much to accommodate a provider.

26 | Where and How to Ask Django Questions

All developers get stuck at one point or another on something that's impossible to figure out alone. When you get stuck, don't give up!

26.1 What to Do When You're Stuck

Follow these steps to increase your chances of success:

- ❶ Troubleshoot on your own as much as possible. For example, if you're having issues with a package that you just installed, make sure the package has been installed into your virtualenv properly, and that your virtualenv is active.
- ❷ Read through the documentation in detail, to make sure you didn't miss something.
- ❸ See if someone else has had the same issue. Check Google, mailing lists, and StackOverflow.
- ❹ Can't find anything? Now ask on StackOverflow. Construct a tiny example that illustrates the problem. Be as descriptive as possible about your environment, the package version that you installed, and the steps that you took.
- ❺ Still don't get an answer after a couple of days? Try asking on the django-users mailing list or in IRC.

26.2 How to Ask Great Django Questions in IRC

IRC stands for **Internet Relay Chat**. There are channels like #python and #django on the Freenode IRC network, where you can meet other developers and get help.

A warning to those who are new to IRC: sometimes when you ask a question in a busy IRC channel, you get ignored. Sometimes you even get trolled by cranky developers. Don't get discouraged or take it personally!

The IRC #python and #django channels are run entirely by volunteers. You can and should help out and answer questions there too, whenever you have a few free minutes.

- ❶ When you ask something in IRC, be sure that you've already done your homework. Use it as a last resort for when StackOverflow doesn't suffice.
- ❷ Paste a relevant code snippet and traceback into <https://gist.github.com/> (or another pastebin).
- ❸ Ask your question with as much detail and context as possible. *Paste the link to your code snippet/traceback.* Be friendly and honest.

TIP: Use a Pastebin!

Don't ever paste code longer than a few characters into IRC. Seriously, don't do it. You'll annoy people. Use a pastebin!

- ❹ When others offer advice or help, thank them graciously and make them feel appreciated. A little gratitude goes a long way. A lot of gratitude could make someone's day. Think about how you would feel if you were volunteering to help for free.

26.3 Insider Tip: Be Active in the Community

The biggest secret to getting help when you need it is simple: be an active participant in the Python and Django communities.

The more you help others, the more you get to know people in the community. The more you put in, the more you get back.

26.3.1 10 Easy Ways to Participate

- ❶ Attend Python and Django user group meetings. Join all the local groups that you can find on <http://wiki.python.org/moin/LocalUserGroups>. Search meetup.com for Python and join all the groups near you.

- ② Attend Python and Django conferences in your region and country. Learn from the experts. Stay for the entire duration of the sprints and contribute to open-source projects. You'll meet other developers and learn a lot.
- ③ Contribute to open source Django packages and to Django itself. Find issues and volunteer to help with them. File issues if you find bugs.
- ④ Join #python and #django on IRC Freenode and help out.
- ⑤ Find and join other smaller niche Python IRC channels. There's #py-ladies, and there are also foreign-language Python IRC channels listed on <http://www.python.org/community/irc/>.
- ⑥ Answer Django questions on StackOverflow.
- ⑦ Meet other fellow Djangonauts on Twitter. Be friendly and get to know everyone.
- ⑧ Join the Django group on LinkedIn, comment on posts, and occasionally post things that are useful to others.
- ⑨ Volunteer for diversity efforts. Get involved with PyLadies and help make the Python community more welcoming to women.
- ⑩ Subscribe to Planet Django, an aggregated feed of blog posts about Django. Comment on blogs and get to know the community. <http://www.planetdjango.org/>

26.4 Summary

One of the strengths of Django is the human factor of the community behind the framework. Assume a friendly, open stance when you need guidance and odds are the community will rise to the task of helping you. They won't do your job for you, but in general they will reach out and attempt to answer questions or point you in the right direction.

27 | Closing Thoughts

While we've covered a lot of ground here, this is also just the tip of the ice cream cone. We plan to add more material and revise the existing material as time goes on, with a new edition released whenever a new version of Django is released.

If this book does well, we may write other books in the future.

We'd genuinely love to hear from you. Email us and let us know:

- Did you find any of the topics unclear or confusing?
- Any errors or omissions that we should know about?
- What additional topics would you like us to cover in a future edition of this book?

We hope that this has been a useful and worthwhile read for you. Cheers to your success with your Django projects!

Daniel Greenfeld
pydanny@cartwheelweb.com
Twitter: [@pydanny](#)

Audrey Roy
audreyr@cartwheelweb.com
Twitter: [@audreyr](#)

Appendix A: Packages Mentioned In This Book

This is a list of the third-party Python and Django packages that we've described or mentioned in this book. We've also snuck in a few really useful packages that we don't mention in the book but that we feel are extremely useful.

As for the packages that we're currently using in our own projects: the list has some overlap with this list but is always changing. Please don't use this as the definitive list of what you should and should not be using.

Core

Django <http://djangoproject.com>

The web framework for perfectionists with deadlines.

Pillow <http://pypi.python.org/pypi/Pillow>

Friendly installer for the Python Imaging Library.

South <http://south.readthedocs.org>

Easy database migrations for Django

Sphinx <http://sphinx-doc.org/>

Documentation tool

django-braces <http://django-braces.readthedocs.org>

Drop-in mixins that really empower Django's Class Based Views

django-debug-toolbar <http://pypi.python.org/pypi/django-debug-toolbar>

Display panels used for debugging Django HTML views.

django-model-utils <http://pypi.python.org/pypi/django-model-utils>

Useful model utilities including a time stamped model.

virtualenv <http://virtualenv.org>

Virtual Environments for Python

Asynchronous

celery <http://www.celeryproject.org/>

Distributed Task Queue

django-celery <http://docs.celeryproject.org/en/latest/django/>

Celery integration for Django

rq <https://pypi.python.org/pypi/rq>

A simple, lightweight, library for creating background jobs, and processing them.

Deployment

dj-database-url <http://pypi.python.org/pypi/dj-database-url>

This simple Django utility allows you to easily use Heroku for database access.

django-heroku-memcacheify <http://pypi.python.org/pypi/django-heroku-memcacheify>

Easy Memcached settings configuration for Heroku.

Forms

django-crispy-forms <http://django-crispy-forms.readthedocs.org/>

Rendering controls for Django forms

django-floppyforms <http://django-floppyforms.readthedocs.org/>

Form field, widget, and layout that can work with django-crispy-forms.

Logging

newrelic <http://newrelic.com>

Realtime logging and aggregation platform

sentry <http://getsentry.com>

Exceptional error aggregation

Project Templates

django-skel <https://github.com/rdegges/django-skel>

Django project template optimized for Heroku deployments

django-twoscoops-project <https://github.com/twoscoops/django-twoscoops-project>

The sample project layout from the book.

REST APIs

django-rest-framework <http://django-rest-framework.org/>

Expose Model and non-Model resources as a RESTful API.

django-tastypie <http://django-tastypie.readthedocs.org>

Expose Model and non-Model resources as a RESTful API.

Security

django-secure <http://pypi.python.org/pypi/django-secure>

Helps you lock down your site's security using practices advocated by security specialists

django-sslify <https://github.com/rdegges/django-sslify>

Forcing HTTPs across your Django site

Testing

coverage <http://coverage.readthedocs.org/>

Checks how much of your code is covered with tests

django-discover-runner <http://pypi.python.org/pypi/django-discover-runner>

Test runner based off unittest2

factory boy https://pypi.python.org/pypi/factory_boy

A package that generates model test data.

model mommy https://pypi.python.org/pypi/model_mommy

Another package that generates model test data.

mock <https://pypi.python.org/pypi/mock>

Not explicitly for Django, this allows you to replace parts of your system with mock objects.

This project made its way into the standard library as of Python 3.3.

User Registration

django-registration <http://django-registration.readthedocs.org/>

Email and username registration made easy, but it lacks sample templates.

django-social-auth <http://django-social-auth.readthedocs.org/>

Easy social authentication and registration for Twitter, Facebook, Google, and lots more.

Miscellaneous

django-extensions <http://django-extensions.readthedocs.org/>

Provides shell_plus management command and a lot of other utilities.

django-haystack <http://django-haystack.readthedocs.org/>

Full-text search that works with SOLR, Elasticsearch, and more.

django-pipeline <http://django-pipeline.readthedocs.org/>

Compression of CSS and JS. Use with cssmin and jsmin packages.

python-dateutil <https://pypi.python.org/pypi/python-dateutil>

Provides powerful extensions to Python's datetime module.

psycopg2 <http://pypi.python.org/pypi/psycopg2>

PostgreSQL database adapter

requests <http://docs.python-requests.org>

Easy-to-use HTTP library that replaces Python's urllib2 library.

unicode-slugify <https://github.com/mozilla/unicode-slugify>

A Mozilla supported slugifier that supports unicode characters.

virtualenvwrapper <http://www.doughellmann.com/projects/virtualenvwrapper/>

Makes virtualenv better for Mac OS X and Linux!

Appendix B: Troubleshooting

This appendix contains tips for troubleshooting common Django installation issues.

Identifying the Issue

Often, the issue is one of:

- That Django isn't on your system path, or
- That you're running the wrong version of Django

Run this at the command line:

EXAMPLE 27.1

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

If you're running Django 1.5, you should see the following output:

EXAMPLE 27.2

```
1.5
```

Don't see the same output? Well, at least you now know your problem. Read on to find a solution.

Our Recommended Solutions

There are all sorts of different ways to resolve Django installation issues (e.g. manually editing your PATH environment variable), but the following tips will help you fix your setup in a way that is

consistent with what we describe in chapter on The Optimal Django Environment Setup.

Check Your Virtualenv Installation

Is **virtualenv** installed properly on your computer? At the command line, try creating a test virtual environment and activating it.

If you're on a Mac or Linux system, verify that this works:

EXAMPLE 27.3

```
$ virtualenv testenv
$ source testenv/bin/activate
```

If you're on Windows, verify that this works:

EXAMPLE 27.4

```
C:\code> virtualenv testenv
C:\code> testenv\Scripts\activate
```

Your virtualenv should have been activated, and your command line prompt should now have the name of the virtualenv prepended to it.

On Mac or Linux, this will look something like:

EXAMPLE 27.5

```
(testenv) $
```

On Windows, this will look something like:

EXAMPLE 27.6

```
(testenv) >
```

Did you run into any problems? If so, study the Virtualenv documentation (<http://virtualenv.org>) and fix your installation of Virtualenv.

If not, then continue on.

Check if Your Virtualenv Has Django 1.5 Installed

With your virtualenv activated, check your version of Django again:

EXAMPLE 27.7

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

If you still don't see 1.5, then try using pip to install Django 1.5 into testenv:

EXAMPLE 27.8

```
(testenv) $ pip install Django==1.5
```

Did it work? Check your version of Django again. If not, check that you have **pip** installed correctly as per the official documentation (<http://pip-installer.org>).

Check for Other Problems

Follow the instructions in the official Django docs for troubleshooting problems related to running django-admin.py: <https://docs.djangoproject.com/en/1.5/faq/troubleshooting/>

Appendix C: Additional Resources

This appendix lists additional resources that are applicable to modern Django. While there is more content available than what is listed here, much of it is out of date. Therefore, we will only list content that is current and applicable to Django 1.5.

Getting Started with Django

<http://gettingstartedwithdjango.com/>

Partially funded by the Django Software Foundation this is a free video lesson series for Django 1.5. The creator, Kenneth Love, was a technical reviewer for this book, and many of the practices advocated in the video series match those presented in this book.

Lincoln Loop's Django Best Practices

<http://lincolnloop.com/django-best-practices/>

This is a really good reference of practices very similar to those espoused in this book.

ccbv.co.uk

<http://ccbv.co.uk/>

Detailed descriptions, with full methods and attributes, for each of Django's class-based generic views.

Official Django 1.5 Documentation

<https://docs.djangoproject.com/en/1.5/>

The official Django documentation has seen a significant amount of improvement with the release of version 1.5. If you've used a previous version of Django, make sure that you are reading the correct edition of the documentation.

pydanny's blog

<http://pydanny.com/tag/django.html>

A good amount of this blog is about Django and is very current. As the author of this blog is also one of this book's authors, the style of the blog loosely resembles the content of this book.

Effective Django

<http://effectivedjango.com>

Nathan Yergler's excellent combined of the notes and examples developed for talks prepared for PyCon 2012, PyOhio 2012, and PyCon 2013, and for Eventbrite web engineering.

Acknowledgments

This book was not written in a vacuum. We would like to express our thanks to everyone who had a part in putting it together.

The Python and Django Community

The Python and Django communities are an amazing family of friends and mentors. Thanks to the combined community we met each other, fell in love, and were inspired to write this book.

Technical Reviewers

We can't begin to express our gratitude to our technical reviewers. Without them this book would have been littered with inaccuracies and broken code. Special thanks to Malcolm Tredinnick for providing an amazing wealth of technical editing and oversight, Kenneth Love for his constant editing and support, Jacob Kaplan-Moss for his honesty, Randall Degges for pushing us to do it, Lynn Root for her pinpoint observations, and Jeff Triplett for keeping our stuff agnostic.

Malcolm Tredinnick lived in Sydney, Australia and spent much of his time travelling internationally. He was a Python user for over 15 years and Django user since just after it was released to the public in mid-2005, becoming a Django core developer in 2006. A user of many programming languages, he felt that Django was one of the better web libraries/frameworks that he used professionally and was glad to see its incredibly broad adoption over the years. In 2012 when he found out that we were co-hosting the first PyCon Philippines, he immediately volunteered to fly out, give two talks, and co-run the sprints. Sadly, he passed away in March of 2013, just two months after this book was released. His leadership and generosity in the Python and Django community will always be remembered.

Kenneth Love is a full-stack, freelance web developer who focuses mostly on Python and Django. He works for himself at Gigantuan and, with his long-time development partner Chris Jones, at Brack3t. Kenneth created the ‘Getting Started with Django’ tutorial series for getting people new to Django up to speed with best practices and techniques. He also created the `django-braces` package which brings several handy mixins to the generic class-based views in Django.

Lynn Root is an engineer for Red Hat on the freeIPA team, known for breaking VMs and being loud about it. Living in San Francisco, she is the founder & leader of PyLadiesSF, and the de facto missionary for the PyLadies word. Lastly, she has an unhealthy obsession for coffee, Twitter, and socializing.

Barry Morrison is a self-proclaimed geek, lover of all technology. Multidiscipline Systems Administrator with more than 10 years of professional experience with Windows, Linux and storage in both the Public and Private sectors. He is also a Python and Django aficionado and Arduino tinkerer.

Jacob Kaplan-Moss is the co-BDFL of Django and a partner at Revolution Systems which provides support services around Django and related open source technologies. Jacob previously worked at World Online, where Django was invented, where he was the lead developer of Ellington, a commercial Web publishing platform for media companies.

Jeff Triplett is an engineer, photographer, trail runner, and KU Basketball fan who works for Revolution Systems in Lawrence, Kansas where he helps businesses and startups scale. He has been working with Django since early 2006 and he previously worked at the Lawrence Journal-World, a Kansas newspaper company, in their interactive division on Ellington aka ‘The CMS’ which was the original foundation for Django.

Lennart Regebro created his first website in 1994, and has been working full time with open source web development since 2001. He is an independent contractor based in Kraków, Poland and the author of ‘Porting to Python 3’.

Randall Degges is a happy programmer with a passion for building API services for developers. He is an owner and Chief Hacker at Telephony Research, where he uses Python to build high performance web systems. Randall authored *The Heroku Hacker’s Guide*, the only Heroku book yet published. In addition to writing and contributing to many open source libraries, Randall also maintains a popular programming blog.

Sean Bradley is a developer who believes Bach’s *Art of the Fugue* and Knuth’s *Art of Computer Programming* are different chapters from the same bible. He is founder of Bravoflix, the first online video subscription service in the U.S. dedicated exclusively to the performing arts, and founder of BlogBlimp, a technology consultancy with a passion for Python. In addition, Sean runs Concert Talent, a production company providing corporate entertainment, engagement

marketing, comprehensive educational outreach, as well as international talent management and logistical support for major recording and touring artists. When he isn't busy coding, he's performing on stages in China, spending time above the treeline in the Sierras, and rebooting music education as a steering committee member for the Los Angeles Arts Consortium.

Chapter Reviewers

The following are people who gave us an amazing amount of help and support with specific chapters during the writing of this book. We would like to thank Preston Holmes for his contributions to the User model chapter, Tom Christie for his sage observations to the REST API chapter, and Donald Stufft for his support on the Security chapter.

Preston Holmes is a recovering scientist now working in education. Passionate about open source and Python, he is one of Django's newest core developers. Preston was involved in the development of some of the early tools for the web with Userland Frontier.

Tom Christie is a committed open source hacker and Djangonaut, living and working in the wonderful seaside city of Brighton, UK. Along with Web development he also has a background in speech recognition and network engineering. He is the author of the 'Django REST framework' package.

Donald Stufft is a Software Engineer at Nebula, Inc. In addition to working on securing OpenStack and Nebula's platform, he is an open source junkie and involved in several OSS projects. He is the creator of Crate.io (a Python packaging index) and Slumber (a generic wrapper around RESTful apis). He lives in Philadelphia with his wife and daughter.

Alpha Reviewers

During the Alpha period an amazing number of people sent us corrections and cleanups. This list includes: Brian Shumate, Myles Braithwaite, Robert Węglarek, Lee Hinde, Gabe Jackson, Jax, Baptiste Mispelon, Matt Johnson, Kevin Londo, Esteban Gaviota, Kelly Nicholes, Jamie Norrish, Amar Šahinović, Patti Chen, Jason Novinger, Dominik Aumayr, Hrayr Artunyan, Simon Charettes, Joe Golton, Nicola Marangon, Farhan Syed, Florian Apolloner, Rohit Aggarwa, Vinod Kurup, Mickey Cheong, Martin Bächtold, Phil Davis, Michael Reczek, Prahlad Nrsimha Das, Peter Heise, Russ Ferriday, Carlos Cardoso, David Sauve, Maik Hoepfel, Timothy Goshinski, Florian Apolloner, and Francisco Barros, João Oliveira, Zed Shaw, and Jannis Leidel.

Beta Reviewers

During the Beta period an awesome number of people sent us corrections, cleanups, bug fixes, and suggestions. This includes: Francisco Barros, Florian Apolloner, David Beazley, Alex Gaynor, Jonathan Hartley, Stefane Fermigier, Deric Crago, Nicola Marangon, Bernardo Brik, Zed Shaw, Zoltán Árokszállási, Charles Denton, Marc Tamlyn, Martin Bächtold, Carlos Cardoso, William Adams, Kelly Nichols, Nick August, Tim Baxter, Joe Golton, Branko Vukelic, John Goodleaf, Graham Dumpleton, Richard Cochrane, Mike Dewhirst, Jonas Obrist, Anthony Burke, Timothy Goshinski, Felix Ingram, Steve Klass, Vinay Sajip, Olav Andreas Lindekleiv, Kal Sze, John Jensen, Jonathan Miller, Richard Corden, Dan Poirier, Patrick Jacobs, R. Michael Herberge, and Dan Loewenherz.

Final Reviewers

During the Final period the following individuals sent us corrections, cleanups, bug fixes, and suggestions. This includes: Chris Jones, Davide Rizzo, Tiberiu Ana, Dave Castillo, Jason Bittel, Erik Romijn, Darren Ma, Dolugen Buuraldaa, Anthony Burke, Hamish Downer, Wee Liat, Álex González, Wee Liat, Jim Kalafut, Harold Ekstrom, Felipe Coelho, Andrew Jordan, Karol Breguła, Charl Botha, Fabio Natali, Tayfun Sen, Garry Cairns, Dave Murphy, Chris Foresman, Josh Schreuder, Marcin Pietranik, Vraj Mohan, Yan Kalchevskiy, Jason Best, Richard Donkin, Peter Valdez, Jacinda Shelly, and Jamie Norrish.

If your name is not on this list but should be, please send us an email so that we can make corrections!

Typesetting

We originally typeset the alpha version the book with iWork Pages. When we ran into the limitations of Pages, we converted the book to LaTeX. We thank Laura Gelsomino for helping us with all of our LaTeX issues and for improving upon the book layout. We also thank Matt Harrison for his guidance and support in the generation of the kindle and ePub versions.

Laura Gelsomino is an economist keen about art and writing, and with a soft spot for computers, who found the meeting point between her interests the day she discovered LaTeX. Since that day, she habitually finds any excuse to vent her aesthetic sense on any text she can lay her hands on, beginning with her economic models.

Matt Harrison has over a decade of Python experience across the domains of search, build management and testing, business intelligence and storage. He has taught hundreds how to program in Python at PyCON, OSCON and other conferences and user groups. Most recently he was worked to analyze data to optimize profits, vendor negotiations, and manufacturing yields.

Index

- `--settings`, 30
- `<configuration_root>`, 16, 17
- `<django_project_root>`, 17, 19
- `<repository_root>`, 16–19, 40
- `__unicode__()`, 150
- `{% block %}`, 128, 130, 133
- `{% extends %}`, 130
- `{% includes %}`, 133
- `{% load %}`, 128
- `{% static %}`, 128
- `{{ block.super }}`, 129–132
- abstract base classes, 49, 51, 52, 59
- `AbstractBaseUser`, 159
- `AbstractUser`, 158
- Acknowledgments, 269–273
 - Alpha Reviewers, 271
 - Beta Reviewers, 272
 - Chapter Reviewers, 271
 - Final Reviewers, 272
 - Technical Reviewers, 269–271
 - Typesetting, 272–273
- AJAX, 112
- `allow_tags` warning, 152, 224
- Apache, 205, 249
- `assets/`, 20
- CBVs, *see* Class-Based Views
- Class-Based Views, 61–84, 105, 133, 200
- `clean()` methods, 92, 93, 102
- Coding Style, 1–8
- Content Delivery Networks, 206
- Continuous Integration, 186–187
- `coverage.py`, 180
- CSRF, 112, 213, 214
- custom field validators, 87–89, 92
- database normalization, 54
- DEBUG, 30, 32, 33, 210
- Django Packages, 187, 188
- Django’s Admin, 147
- Django’s admin, 154
- `django-admin.py`, 14, 32
- `django-admin.py startproject`, 16–17, 21
- `django-debug-toolbar`, 199
- `django-discover-runner`, 180, 181
- `django.contrib.admin`, 147–154, 222
- `django.contrib.admindocs`, 152–154, 224
- `django.contrib.humanize`, 245
- `django.utils.html.remove_tags()`, 245
- `django.utils.html.strip_tags()`, 245
- `django.utils.html.timezone`, 246
- `django.utils.translation`, 246
- `DJANGO_SETTINGS_MODULE`, 31
- Documentation, 193–197
- `docutils`, 153

- Don't Repeat Yourself, [27](#), [29](#), [64](#), [65](#), [95](#), [141](#), [184](#)
- environment variables, [34–39](#)
- `eval()`, [215](#)
- `exec()`, [215](#)
- `execfile()`, [215](#)
- FBVs, [61–67](#), *see* Function-Based Views
- filters, [137–138](#), [186](#)
- fixtures, [9–10](#)
- `form.is_valid()`, [102](#)
- `form_invalid()`, [75](#)
- `form_valid()`, [74](#), [79–81](#), [102](#)
- Forms, [6](#), [74–83](#), [85–104](#)
- `get_absolute_url()`, [151](#)
- `get_env_setting()`, [38](#), [39](#)
- Git, [14](#)
- Gunicorn, [249](#)
- HSTS, [212](#)
- HTTPS, [210](#), [211](#)
- `ImproperlyConfigured`, [39](#)
- indexes, [49](#), [59](#)
- `intcomma`, [245](#)
- `IPAddressField`, [56](#)
- `is_valid()`, [102](#)
- Jinja2, [135](#), [144–146](#)
- kept out of version control, [28](#)
- license, [ii](#)
- Linux, [11](#), [20](#), [21](#), [35](#), [264](#)
- `local_settings` anti-pattern, [28–29](#), [34](#)
- logging, [227–236](#)
 - CRITICAL, [228](#), [229](#)
 - DEBUG, [228](#), [231–233](#)
 - ERROR, [228](#), [229](#)
 - exceptions, [233–234](#)
 - INFO, [228](#), [231](#)
 - WARNING, [228](#), [230](#)
- `logrotate`, [234](#)
- Loose coupling, [64](#), [65](#)
- Mac OS X, [11](#), [20](#), [21](#), [35](#), [264](#)
- `manage.py`, [14](#), [32](#)
- Memcached, [203](#)
- Mercurial, [14](#)
- messages, [80](#)
- `Meta.exclude`, [219–221](#)
- `Meta.fields`, [219–221](#)
- method resolution order, [71](#)
- mixins, [70](#), [71](#), [80](#), [81](#), [100](#), [244](#)
- `mod_python` warning, [250](#)
- `mod_wsgi`, [249](#)
- model managers, [56–58](#), [238](#), [239](#)
- `ModelForms`, [85](#), [87](#), [89–91](#), [94](#), [102](#), [103](#), [219–221](#), [241](#)
- multi-table inheritance, [49–52](#)
- Multiple Settings Files, [29–34](#)
- MySQL, [10](#), [122](#), [189](#), [202](#), [203](#)
- NASA, [xix](#), [179](#)
- Nginx, [205](#), [249](#)
- Open Source Initiative, [176](#)
- Open Source Licenses, [176](#)
- pastebin, [254](#)
- pickle, [215](#)
- pip, [11](#), [13](#), [14](#), [153](#), [168](#), [169](#), [172](#), [175](#), [176](#), [193](#), [265](#)
- Platform as a Service, [42](#), [250–252](#)

- PostgreSQL, [9–11](#), [49](#), [56](#), [122](#), [189](#), [202](#), [203](#)
- PowerShell, [36](#)
- print(), [231–232](#)
- PROJECT_ROOT, [43](#), [44](#)
- proxy models, [49](#), [51](#)
- Python Package Index, [168](#), [176](#)
- PYTHONPATH, [14](#)
- PyYAML security warning, [215](#), [216](#)

- README.rst, [16](#), [22](#)
- Redis, [203](#)
- requirements, [40–42](#)
- requirements.txt, [16](#), [21](#)
- requirements/, [40](#)
- REST APIs, [105–114](#)
- reStructuredText, [173](#), [193](#), [194](#)

- SECRET_KEY, [28–29](#), [34–39](#), [210](#)
- select_related(), [124](#), [200](#)
- settings, [19](#), [27–45](#)
- settings/base.py, [38](#)
- settings/local.py, [30](#), [32](#), [35](#)
- signals, [237](#), [238](#), [241](#)
- site_assets/, [20](#)
- slugify(), [137](#), [245](#), [246](#)
- Sphinx, [172](#), [173](#), [193](#), [195](#)
- SQLite3, [9](#), [10](#), [189](#)
- SSL, [211](#)
- STATICFILES_DIRS, [20](#)

- template tags, [8](#), [137–142](#), [186](#)
- TEMPLATE_STRING_IF_INVALID, [135](#)
- templates, [16](#), [20–22](#), [42](#), [44](#), [79](#), [81](#), [83](#), [115–135](#), [145](#)
- test coverage, [187–191](#)
- testing, [179–191](#)
- TimeStampedModel, [51](#)

- twoscoops_project, [24](#)

- unicode(), [150](#)
- unit tests, [181](#)
- Upstream Caches, [206](#)
- URLConfs, [17](#), [19](#), [63–67](#), [134](#), [153](#)
- User model, [155–166](#)

- validation, [102](#)
- Vanilla Steak, [72](#)
- Varnish, [206](#)
- virtualenv, [11](#), [12](#), [14](#), [20–21](#), [35](#), [36](#), [153](#), [169](#), [171](#), [176](#), [264](#), [265](#)
- virtualenvwrapper, [12](#), [36](#)

- Windows, [11](#), [20](#), [36](#), [264](#)
- WSGI, [19](#), [205](#), [249](#)

- Zen of Python, [118](#), [142](#)