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, Alpha Version, 20130125
by Daniel Greenfeld and Audrey Roy

Table of Contents

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 iPaqs 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-wracking critiques from Django core developers whom we greatly respect. We've had this book reviewed by more

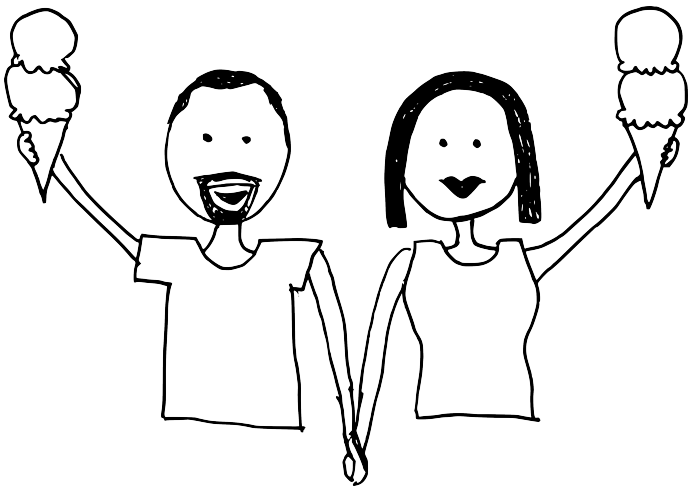
technical reviewers than the average technical book, and we've poured countless hours into revisions. That being said, there is always the possibility of errors or omissions. There is also the possibility that better practices may emerge than those described here.

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!



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 a grounding in Python (http://learnpythonthehardway.org/) and have at least gone through the 5-page Django tutorial: https://docs.djangoproject.com/en/1.5/intro/tutorial01/.

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:

```
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:

```
# 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.
- ***Bold and 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.

THIRD-PARTY PACKAGES: Recommendations For Your Projects

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: Third-Party Packages We Use*.

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 Functions, Thin Views, Stupid Templates

When deciding where to put a piece of code, we like to follow the "Fat Models, Helper Functions, 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 the chapters on Views, *Templates* and *Template Tags*.

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 the chapter on *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.

Coding Style

1

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.

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.

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: *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 "A Foolish Consistency is the Hobgoblin of Little Minds" (http://2scoops.org/hobgoblin-of-little-minds/).

The Word on Imports

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

1. Standard library imports
2. Related third-party imports
3. Local application or library specific imports

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

```
# 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:

1. Standard library imports.
2. Imports from core Django.
3. Imports from third-party apps.
4. Imports from the apps that you created as part of your Django project. (You'll read more about apps in the *Fundamentals of App Design* chapter.)

Use 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, **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 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!

```
# cones/views.py
# Hard coding of package name
from django.views.generic import CreateView

# DON'T DO THIS: Hardcoding of the 'cones' package
from cones.models import WaffleCone
from cones.forms import WaffleConeForm

class WaffleConeCreateView(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 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 relative imports. Here's the corrected example:

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

# Relative imports of the 'cones' package
from .models import WaffleCone
from .forms import WaffleConeForm

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

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

Additional reading:

- http://www.python.org/dev/peps/pep-0328/

Avoid Using Import *

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

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

Never do the following:

```
# 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.

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 `CharField` 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:

```
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.

TODO: What happens when you walk into an ice cream shop and import * - illustration.

Django Coding Style Guidelines

Follow the Official Django Coding Style Standards

Django has its own set of style guidelines that extend PEP 8. See:

https://docs.djangoproject.com/en/1.5/internals/contributing/writing-code/coding-style/

Rather than do it here, in many chapters we'll explore the specifics of the Django coding style guidelines as they apply in that chapter.

Other Good Conventions

Although these conventions 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. 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.
- Use underscores rather than dashes in template block names.

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***.

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.

The Optimal Django Environment Setup 2

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

Use the Same Database Locally and in Production

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 databases for development and production:

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.

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.

Different Databases Have Different Field Types and 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.

Specifically, in the case of SQLite3 versus most other relational databases, SQLite3 has dynamic, weak typing instead of strong typing. Yet the Django ORM has features that allow your code to interact with SQLite3 in a more strongly typed manner.

That may sound great, 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. PostgreSQL may take some work to get running, but we find it's well worth the effort.

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 www.postgresql.org/download/windows/
- Linux: Install via your package manager, or follow the instructions at www.postgresql.org/download/linux/

Use Pip and Virtualenv

Pip is a tool for managing and installing 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. It's great for situations where, say, you're working on one project that requires Django 1.4 and another that requires Django 1.5.

We strongly urge you to use both. Follow the instructions at:

- pip: www.pip-installer.org
- virtualenv: www.virtualenv.org

Most experienced **Djangonauts** can't live without `pip` and `virtualenv`.

TIP: Virtualenvwrapper

For developers using Mac OSX or Linux, who are experienced with the shell, in addition to pip and virtualenv, we also highly recommend virtualenvwrapper: http://virtualenvwrapper.readthedocs.org

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:

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

With virtualenvwrapper, you'd only have to type:

```
$ workon twoscoops
```

We find that this makes our lives easier, but it's not an absolute necessity.

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 the *Settings and Requirements Files* chapter.

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/).

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.

How To Lay Out Django Projects

3

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.

Django 1.5's Default Project Layout

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

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

Here's the resulting project layout:

```
mysite/
    manage.py
    my_app/
        __init__.py
        models.py
        tests.py
        views.py
    mysite/
        __init__.py
        settings.py
        urls.py
        wsgi.py
```

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:

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

Let's go over each level in detail:

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***, ***docs/*** directory, ***design/*** directory, ***.gitignore***, ***requirements.txt*** files, and other high-level files that are required for deployment.

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.

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).

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:

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

Let's do an in-depth review of this layout. Turn the page to learn more about what's going on here!

As you can see, in the ***icratings_project/*** directory, which is the `<repository_root>`, we have the following files and directories. We describe them in the table below:

icratings_project/

File or Directory	Purpose
.gitignore	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 .hgignore file.)
README.rst *docs/*	Developer-facing project documentation. You'll read more about this in the *Documentation* chapter.
Makefile	Contains simple deployment tasks and macros. For more complex deployments you may want to rely on tools like Fabric.
requirements.txt	A list of Python packages required by your project, including the Django 1.5 package. You'll read more about this in the chapter on *Django's Secret Sauce: Third-Party Packages*.
icratings/	The <django_project_root> of the project.

When anyone visits this project, they are provided with a high level view of the project. We've found this allows us to work easily with each other 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.

On the next page, we'll cover the `<django_project_root>` directory.

Inside the ***icratings_project/icratings*** directory, at the `<django_project_root>`, we place the following files/directories:

icratings_project/icratings/

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

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.

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.

If you're using virtualenvwrapper, that directory defaults to ***~/.virtualenvs/*** and the virtualenv would be located at:

```
~/.virtualenvs/icratings/
```

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:

```
$ pip freeze
```

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 r*equirements.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!

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:

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

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

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.

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.

Fundamentals of Django App Design

4

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 the chapter on *Django's Secret Sauce: Third-Party Packages*.

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 a 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.

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.

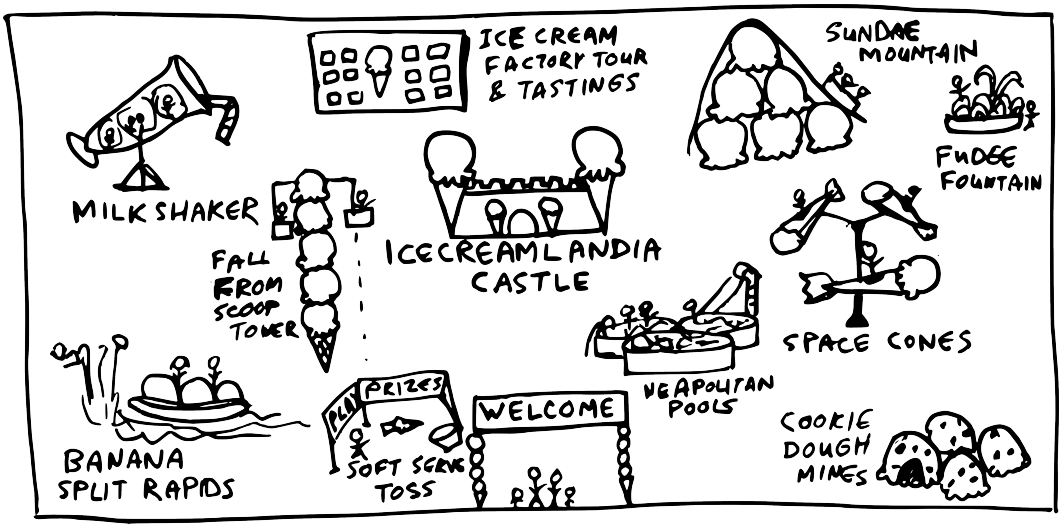
In the future, we might extend the site with apps like:

- 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*:



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.

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.

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.

Settings and Requirements Files

5

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.

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!

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* (http://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. It will typically contain something like the following:

```
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 the "Using Multiple Requirements Files" section.

Settings file	Purpose
base.py	Settings common to all instances of the project.
local.py	This is the settings file that you use when you're working on the project locally. Local development-specific settings include DEBUG mode, log level, and activation of developer tools like django-debug-toolbar. Developers sometimes name this file dev.py.
staging.py	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.
test.py	Settings for running tests including test runners, in-memory database definitions, and log settings.
production.py	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 prod.py.

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 ***local.py*** settings file:

```
$ django-admin.py shell --settings=twoscoops.settings.local
```

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

```
$ django-admin.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 --settings (or DJANGO_SETTINGS_MODULE Value)
Your local development server	twoscoops.settings.local
Your staging server	twoscoops.settings.staging
Your test server	twoscoops.settings.test
Your production server	twoscoops.settings.production

Notice How We Use django-admin.py Here

As described in the official Django documentation, 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/

If you run into "command not found" issues, see *Appendix B: Troubleshooting* for tips on how to resolve your issues.

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***:

```
# 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:

```
$ django-admin.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!

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***:

```
# 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:

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

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 their own copy-and-pasted version of ***local_settings.py.example*** for development, 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.

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.

How To Set Environment Variables Locally

On Mac and many Linux distributions that use **bash** for the shell, you can add lines like the following to your .bashrc, .bash_profile, or .profile:

```
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:

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

For more information, see http://en.wikipedia.org/wiki/Environment_variable.

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:

```
$ 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:

```
$ 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

```
$ 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:

```
>>> 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:

```
# 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.

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:

```
# 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 env variable" % var_name
        raise ImproperlyConfigured(error_msg)
```

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

```
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:

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

WARNING: Don't Import Anything From Django 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.

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 ***<repo_root>***. Then create '***.txt***' files that match the contents of your settings directory. The results should look something like:

```
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:

```
https://www.djangoproject.com/download/1.5c1/tarball/
psycopg2==2.4.5
South==0.7.6
```

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

```
-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:

```
-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*** :

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

Installing From Multiple Requirements Files

For local development:

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

For production:

```
$ 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:

```
$ pip freeze
```

Using multiple requirements files with Platforms as a Service (PaaS)

See the section on "Using a Platform as a Service" in the chapter on *Deploying Django Projects*.

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:

```
# settings/base.py

# Configuring MEDIA_ROOT
# DON'T DO THIS! Hardcoded to just one user's preferences
MEDIA_ROOT = "/Users/pydanny/code/twoscoops_project/media"

# Configuring STATIC_ROOT
# DON'T DO THIS! Hardcoded to just one user's preferences
STATIC_ROOT = "/Users/pydanny/code/twoscoops_project/
collected_static"

# Configuring TEMPLATE_DIRS
# DON'T DO THIS! Hardcoded to just one user's preferences
TEMPLATE_DIRS = (
    "/Users/pydanny/code/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 `PROJECT_ROOT` is with **Unipath** (http://pypi.python.org/pypi/Unipath/), a Python package that does elegant, clean path calculations:

```
# At the top of settings/base.py
from unipath import Path

PROJECT_ROOT = Path(__file__).ancestor(3)

MEDIA_ROOT = PROJECT_ROOT.child('media')
STATIC_ROOT = PROJECT_ROOT.child('static')
STATICFILES_DIRS = (
    PROJECT_ROOT.child('assets'),
)
TEMPLATE_DIRS = (
    PROJECT_ROOT.child('templates'),
)
```

If you really want to set your `PROJECT_ROOT` with the Python standard library's `os.path` module, though, this is one way to do it in a way that will account for paths:

```
# 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.

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.

Database/Model Best Practices

6

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.

THIRD-PARTY PACKAGES: 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.

Basics

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.

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 says (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 BDFL 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."

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 judgement. 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, the chapter on *Finding and Reducing Bottlenecks* has information on index analysis.

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 `OneToOneField` 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: `parent.child`	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.

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.

Here are some simple rules of thumb for know 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.

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:

```
# Code taken with permission from Carl Meyer's
# very useful django-model-utils
from django.db import models
from django.utils.timezone import now
from django.utils.translation import ugettext_lazy as _


class AutoCreatedField(models.DateTimeField):
    """
    A DateTimeField that automatically populates itself at
    object creation.

    By default, sets editable=False, default=now

    """
    def __init__(self, *args, **kwargs):
```

```
        kwargs.setdefault('editable', False)
        kwargs.setdefault('default', now)
        super(AutoCreatedField, self).__init__(*args, **kwargs)


class AutoLastModifiedField(AutoCreatedField):
    """
    A DateTimeField that updates itself on each save() of
    the model.

    By default, sets editable=False and default=now.

    """
    def pre_save(self, model_instance, add):
        value = now()
        setattr(model_instance, self.attname, value)
        return value

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

    """
    created = AutoCreatedField(_('created'))
    modified = AutoLastModifiedField(_('modified'))

    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**:

```
    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:

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

from model_utils 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**), 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:

- https://docs.djangoproject.com/en/dev/topics/db/models/#model-inheritance

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. Which means any problems that may be caused by deletion of migrations may not be detectable for some time.

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.

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 Django projects using Models requires a working knowledge. Since 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.

Cache Before Denormalizing

Often, setting up caching in the right places can save you the trouble of denormalizing your models.

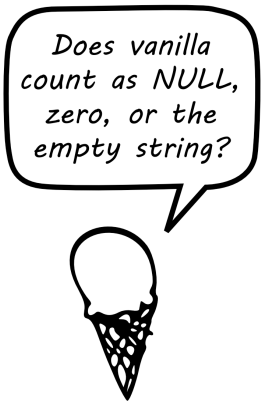
Denormalize Only If Absolutely Needed

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 *Finding and Reducing Bottlenecks* chapter can address, that's when research into the concepts and patterns of database denormalization should begin.

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 chart together to serve as a guide for standard usage of these model field arguments.

Field Type	Setting null=True	Setting blank=True
`CharField`, `TextField`, `SlugField`, `EmailField`, `CommaSeparatedInteger-Field`, etc.	**Don't do this.** Django's convention is to store empty values as the empty string, and to always retrieve NULL or empty values as the empty string for consistency.	**Okay.** 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.
`BooleanField`	**Don't do this.** Use `NullBooleanField` instead.	**Don't do this.**
`IntegerField`, `FloatField`, `DecimalField`, etc.	**Okay** if you want to be able to set the value to NULL in the database.	**Okay** if you want the corresponding form widget to accept empty values. If so, you will also want to set `null=True`.

Field Type	Setting null=True	Setting blank=True
`DateTimeField`, `DateField`, `TimeField`, etc.	**Okay** if you want to be able to set the value to `NULL` in the database.	**Okay** if you want the corresponding form widget to accept empty values, or if you are using `auto_now` or `auto_now_add`. If so, you will also want to set `null=True`.
`ForeignKey`, `ManyToManyField`, `OneToOneField`	**Okay** if you want to be able to set the value to `NULL` in the database.	**Okay** if you want the corresponding form widget (e.g. the select box) to accept empty values.
`IPAddressField`	**Okay** if you want to be able to set the value to `NULL` in the database.	**Not recommended.** 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.)

WARNING: IPAddressField in PostgreSQL

At the time of this writing, there is an open ticket (#5622) for:

Empty ipaddress raises an error (invalid input syntax for type inet: "") [sic]

Until this is resolved, we recommend using `null=True`, `blank=False` for `IPAddressField`.

See https://code.djangoproject.com/ticket/5622 for more details.

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. Which means 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:

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

class PublishedManager(models.Manager):

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


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:

```
>>> 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:

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

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

First, when you use model inheritance, what manager that is applied to the child model depends on if the parent model class is abstract or not. Children of abstract base classes receive their parent's model manager, and children of concrete base classes do not.

Second, and this represents unusual behavior for Python, the first manager applied to a model class is the one that Django treats as the default. This can cause what seems to be unpredictable behavior in your project, since the QuerySets returned by your model's manager might not be what you expect.

Which means, in your model class, always manually define `objects = models.Manager()` above any custom model manager. Django unfortunately breaks the normal Python pattern by assigning the first model manager defined in a model class to be the default.

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/

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.

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.

Function- and Class-Based Views

7

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

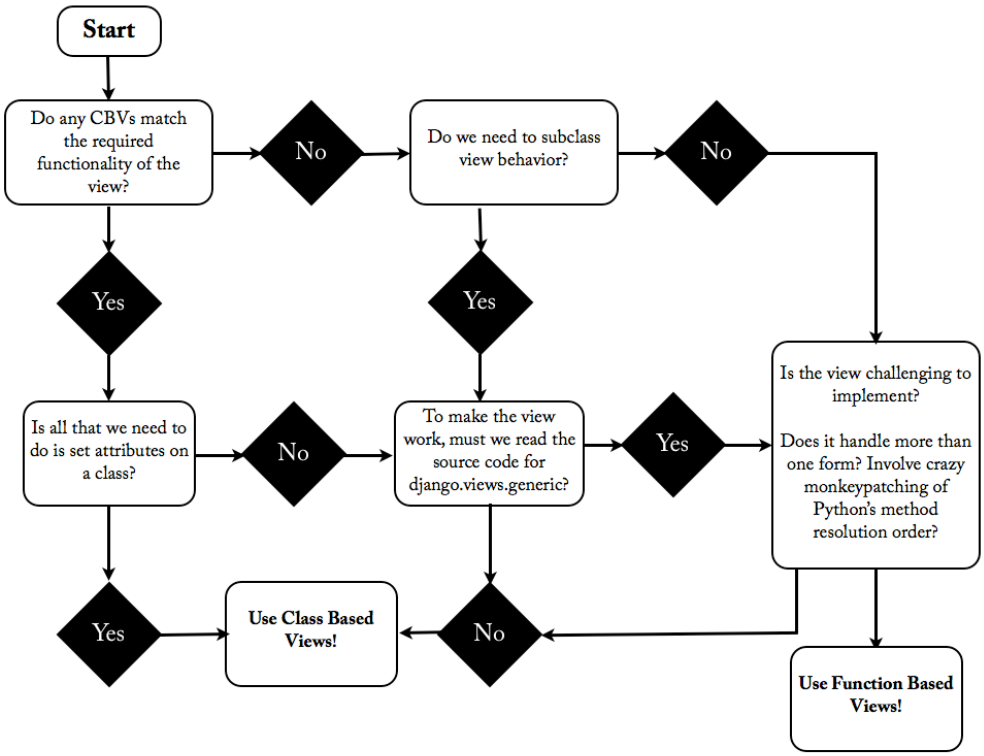
There was a bit of confusion about this due to the wording of the release notes and incorrect information on some blog posts. To clarify:

1. *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.
2. Function-based *generic* views such as `direct_to_template` and `object_list` were deprecated in Django 1.3 and removed in 1.5.

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. The flowchart follows our preference for using CBVs over FBVs.



We err on the side of using 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 is 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.

Keep View Logic Out of URLConfs

Responses are routed to views via URLConfs, in a module normally named ***urls.py***. Per Django's URL design philosophy (https://docs.djangoproject.com/en/1.5/misc/design-philosophies/#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!"

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:

1. The views modules should contain view logic.
2. The URL modules should contain URL logic.

Ever see code like this? Perhaps in the Django documentation?

```
# Similar to the Polls example
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 preference on the next page.

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:

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

from .models import Tasting

class TasteDetailView(DetailView):
    model = Tasting

class TasteResultsView(TasteDetailView):
    template_name = 'tastetests/results.html'
```

Then we define the urls:

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

```
from .views import TasteDetailView, 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 my 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**: Since when did URLConfs become a substitute for views? We should be able to call our views from one or more URLConfs, and our approach lets us do just that. Views should be views and URLConfs should be URLConfs.
- **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. Which means 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. Which means adding authentication, authorization, new content formats, or anything other business requirement tossed my way is much easier to handle.
- **Infinite flexibility yet again**: Our views, by having a formal definition in the views module, can implement their own custom logic.

What if we aren't using CBVs?

The same rules apply.

We've encountered debugging nightmares of projects using FBVs, 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.

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.

Best Practices for Class-Based Views

8

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'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:

- https://docs.djangoproject.com/en/1.5/topics/class-based-views/
- https://docs.djangoproject.com/en/1.5/topics/class-based-views/generic-display/
- https://docs.djangoproject.com/en/1.5/topics/class-based-views/generic-editing/
- https://docs.djangoproject.com/en/1.5/topics/class-based-views/mixins/
- https://docs.djangoproject.com/en/1.5/ref/class-based-views/
- Marc Tamlyn's CBV inspector at ccbv.co.uk

THIRD-PARTY PACKAGES: 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. Or Forms.
- Keep your views simple.
- Keep your **mixins** simpler.

Mixins

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 are a means of collecting functionality.

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:

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

Example of the rules in action:

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

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).

Django CBV Usage Table

Name	Purpose	Two Scoops Example
`View`	Base view or handy view that can be used for anything.	See the section called Implementing a Simple JSON API.
`RedirectView`	Redirect user to another URL	Send users who visit '/log-in/' to '/login/'.
`TemplateView`	Display a Django HTML template.	The '/about/' page of our site.
`ListView`	List objects	List ice cream flavors.
`DetailView`	Display an object	Details on an ice cream flavor.
`FormView`	Submit a form	The site's contact or email form.
`CreateView`	Create an object	Create a new ice cream flavor.

Name	Purpose	Two Scoops Example
`UpdateView`	Update an object	Update an existing ice cream flavor.
`DeleteView`	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 nice, public history of when flavors were added to the database.

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.

General Tips for Django CBVs

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

Constraining Django CBV Access to Authenticated Users

While the Django CBV documentation gives a functional example of using `django.contrib.auth.decorators.login_required` with CBVs, it's uncomfortable to use. Fortunately, django-braces provides a ready implementation that you can attach in moments. For example, we could do the following in all of the Django CBVs we've written so far:

```
# 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.

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. This return value should be a `django.http.HttpResponseRedirect`.

```
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(FlavorCreateForm, self).form_valid(form)
```

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`.

```
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(FlavorCreateForm, self).form_invalid(form)
```

Additional References:

- http://pydanny.com/tag/django-CBVs.html
- www.python.org/download/releases/2.3/mro/

Summary

This chapter covered:

- Mixins
- Which Django CBV should be used for which task
- General tips for CBV usage

The next chapter will use CBVs along with forms and models.

Common Patterns for Forms

9

Django forms are powerful, flexible, extensible, and robust. 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 these features.

This means that *even* if your Django project doesn't serve HTML, you are probably still using Django forms.

Interestingly enough, the design that these 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 Django's own CBVs (django-rest-framework), but 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.

THIRD-PARTY PACKAGES: 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 as a simple tool for rendering Django forms to Twitter bootstrap. This package plays well with django-floppyforms but conflicts with django-crispy-forms.

How Your Views Should Hook Things Together

Using our project's flavors app as an example, let's chart out some examples of how form-related views might fit together. First, let's define a flavors model that we'll use in various examples throughout this chapter:

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

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

    @models.permalink
    def get_absolute_url(self):
        return ('flavor_detail', (), {"slug": self.slug})
```

Views + ModelForm Example

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.

This is a breakdown of the three views corresponding to add and edit flavor forms, as well as confirmation pages for both forms.

FlavorCreateView (CreateView)	→	FlavorDetailView (DetailView)
FlavorUpdateView (UpdateView)	→	FlavorDetailView (DetailView)

In this example, we stick as closely as possible to the Django convention for naming things, which means the add and edit forms are correspondingly `FlavorCreateView` and

`FlavorUpdateView`, and the confirmation page is `FlavorDetailView`. Writing these views is easy, here is how we do it:

```
# 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):
    """ Allows even unauthenticated users. """
    model = Flavor
```

This looks great! Lots of stuff accomplished for a small bit of code! However, 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.

In the previous 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.

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

```
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):

    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
```

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!

Now, after the flavor is 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 update a flavor:

```
{# 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.

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.

Views + Form Example

In this example we'll create a simple Flavor search by creating a HTML form that doesn't add or edit objects. The form 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.

In this example all we need is a single view:

FlavorListView (`ListView`)	→	FlavorListView (`ListView`)

In this example we want to follow the standard internet convention for search pages, as well as accept a 'GET' request where the search parameter name is 'q'. We also 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***:

```
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 is None:
            # Return the base queryset
            return queryset
        # Return a filtered queryset
        return queryset.filter(title__icontains=q)
```

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

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. Which means the search form should like something like this:

```
{# 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.

Common Form Patterns

We like to have fun combining forms, models, and views because they allow us to get a lot of work done for little effort. In fact, once you get the hang of it, Django provides the ability to create an amazing amount of stable, robust functionality at a an amazing pace. We're going to go over five patterns that should be in every Django developer's toolbox.

Pattern 1: Simply Using a ModelForm With Default Validators

The simplest data changing form we can make is a ModelForm that uses several default validators as-is, without modification. In fact, we already relied on default validators in the first example of this chapter, "Views + ModelForm Example", of which the code below is a snippet:

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

from braces.models import LoginRequiredMixin
from .models import Flavor

class FlavorCreateView(LoginRequiredMixin, CreateView):
    model = Flavor

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

To summarize the snippet:

1. FlavorCreateView and FlavorUpdateView are assigned Flavor as their model.
2. Both views auto-generate a ModelForm based on the Flavor model.
3. Those ModelForms rely on the default field validation rules of the Flavor model.

Django gives us a lot of great defaults for data validation, but that's never enough. Which means the next pattern will demonstrate how to create a custom validator.

Pattern 2: Custom Validators on Form Fields in Model Forms

What if we wanted to be certain that every use of the title field across our project's edible item apps started with the word 'Delicious'? This would be a great reason to use a custom validator!

In this pattern we'll cover how to create custom validators and demonstrate how to add them to both abstract models and forms. As described near the start of this chapter, the `Flavor` model has a title field. For the purpose of this example, we'll assume we have a `WaffleCone` model which also has a title field.

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

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

def validate_delicious(value):
    """ Raise a ValidationError if the value doesn't start
        with the word 'delicious'
    """
    if not value.lower().startswith(u'delicious'):
        msg = u"Enter a value starting with 'Delicious'"
        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. Our `validate_delicious()` validator function does a simple string check, but validators can become quite complex.

TIP: Test Your Validators

Since validators are critical in keeping corruption out of Django project databases, it's a good idea to write tests for them.

In order to use our `validate_delicious()` validator function, we're going to first add it to an abstract model called `DeliciousTitleAbstractModel`, which we plan to use

across our project. In fact, we'll create a ***core/models.py*** module and place the `DeliciousTitleAbstractModel` there.

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

from .validators import validate_delicious

class DeliciousTitleAbstractModel(models.Model):

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

    class Meta:
        abstract = True
```

The last two lines of the above example code for ***core/models.py*** make `DeliciousTitleAbstractModel` an abstract model, which is what we want. Let's alter the original ***flavors/models.py*** Flavor code to use it as the parent class:

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

from core.models import DeliciousTitleAbstractModel

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

    @models.permalink
    def get_absolute_url(self):
        return ('flavor_detail', (), {"slug": self.slug})
```

This works with the Flavor model, and will work with any other food-based model such as a WaffleCone or Cake model. Any of these inheriting from the parent

`DeliciousTitleAbstractModel` class will throw a validation error if users attempt to have a title that doesn't start with 'Delicious'.

Now...

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

To support this behavior, we need to create a custom `FlavorForm` that utilizes our custom validator:

```
# 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 = [validate_delicious]
        self.fields['slug'].validators = [validate_delicious]

    class Meta:
        model = Flavor
```

One thing nice about both examples of validator usage in this pattern is that we haven't had to change the `validate_delicious()` code at all, 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 demonstrate on the next page:

```
# 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):

    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
```

The `FlavorCreateView` and `FlavorUpdateView` views now use the new FlavorForm to validate incoming data. The `Flavor` model can be the identical to how we began this chapter, or can be the altered one inheriting from `DeliciousTitleAbstractModel`.

Pattern 3: Override clean() in CBV / Form

In this example, we'll discuss the `clean()` method and how to validate against persistent data.

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()` and `<field_name>_clean()` methods. You might wonder why Django provides more hooks for validation, so here are our two favorite arguments:

- The `clean()` method is the place to validate two or more fields against each other.
- 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 make a simple example out of ice cream. 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 avoid users ordering flavors that are out of stock we'll put in a `clean_flavor()` method, our form might look like:

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

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

    flavor = forms.ChoiceField()
    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['flavor'].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_flavor, not filter().

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

For HTML powered views, the clean_flavor() 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.

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

        # Silly "too much chocolate" validation example
        if u'chocolate' in flavor.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!

Pattern 4: Overloading 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.

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:

```
# stores/models.py
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 ("store_detail", [self.pk, ])
```

The default ModelForm for this Model forces the user to enter the `title` and `block_address` field, but lets the user 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 (the authors) overloaded forms before we began to delve into their construction was to overload the entire `phone` and `description` fields in the edit form. Which made our code look like this:

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

from .models import IceCreamStore

class IceCreamStoreUpdateForm(forms.ModelForms):
    # Don't do this! You're duplicating the Model field!
    phone = forms.CharField(required=True)
    # Don't do this! You're duplicating the Model field!
    description = forms.TextField(required=True)

    class Meta:
        model = IceCreamStore
```

This form should look very familiar. Why?

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 DRY.

Want to know how gross? Using the above approach, if we adds a `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 useful little detail that's good to remember about Django Forms: *Instantiated form objects store fields in a dict-like attribute called* `fields`. Which means, instead of copy/pasting field definitions from models to forms, we simply apply new attributes to each field:

```
# 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.ModelForms):

    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 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. However, *none of this applies when inheriting the Meta class on forms*.

Which means we can't rely on inheritance to trim the line count in our Ice Cream Shop forms:

```
# 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(forms.ModelForm):

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

    class Meta:
        # Ignored in ModelForm inheritance, so we have to
        # define it again.
        model = IceCreamStore
        # show all the fields!
        fields = ("title", "block_address", "phone",
                "description", )
```

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

We use `Meta.fields` instead of `Meta.excludes` so we know exactly what fields we are exposing. See the chapter on *Security*.

We've got our form classes, so let's use them in the IceCreamStore 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.

Pattern 5: Simple Search Mixin View (1 Mixin, 2 CBV, 1 Form, 2 Models)

In this example we're going to cover how to use a single form and view that corresponds to two different models. First, we're going to assume that both models have a field called 'title' (this pattern also demonstrates why *naming standards in projects is a good thing*). We'll demonstrate how a single Class Based View 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:

```
# 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 is None:
            # No q is specified so we return queryset
            return queryset
        # return a filtered queryset
        return queryset.filter(title__istartswith=q)
```

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:

```
# 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

```
# 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
```

The form? Just define it in HTML for each `ListView`:

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

and

```
{# form to go into stores/store_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.

More Things To Know About Forms 10

> 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.

Use the POST Method in HTML Forms

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

```
<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.

Don't Disable Django's CSRF Protection

This is covered in the *Security* chapter's section on "CSRF protection". Also, please familiarize yourself with Django's documentation on the subject: https://docs.djangoproject.com/en/1.5/ref/contrib/csrf/

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:

1. If the form has bound data, `form.is_valid()` calls the `form.full_clean()` method.
2. `form.full_clean()` iterates through the form fields and each field validates itself:
    2.1. Data coming into the field is coerced into Python via the `to_python()` method or raises a `ValidationError`.
    2.2. Data is validated against field-specific rules, including custom validators. Failure raises a `ValidationError`.
    2.3. If there are any custom `clean_<field>()` methods in the form, they are called at this time.
3. `form.full_clean()` executes the `form.clean()` method.
4. If it's a ModelForm instance, `form._post_clean()` does the following:
    4.1. Sets ModelForm data to the Model instance, regardless of whether `form.is_valid()` is True or False.
    4.2. 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 it gets simpler in practice and all of this functionality lets us really understand what's going on with incoming data. Let's go over an example in the section below:

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

We like to call this the 'WHAT?!?' of form validation.

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

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

At first glance form data being set to the form instance might seem like a bug. However, since ModelForms don't save to the Model Instance until they are activated by the `form.save()` method, we like to think of this as a useful feature.

For example, perhaps you need to catch the details of attempts to use 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'll create a form failure history model in ***core/models.py***:

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

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

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

```
import json

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

from core.models import ModelFormFailureHistory
```

```
class FlavorActionMixin(object):

    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 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)
```

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.

Building REST APIs in Django

11

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.

THIRD-PARTY PACKAGES: For crafting APIs we recommend

- django-tastypie is the most senior in our list. It's a feature-rich, mature, powerful, stable tool for creating APIs from Django models by the developer behind django-haystack, the most commonly used Django search library.
- django-rest-framework builds off of Django CBVs and has a similar feature set to it's older sibling django-tastypie, adding a wonderful browsable API feature.
- django-braces isn't an API library in itself, but can be used to render content into JSON views. Also, because it's a lighter framework, it's arguably easier to incorporate non-Model data. Finally, it's trivial to incorporate directly into existing `views.py` modules.

We highly recommend the following reading:

- http://en.wikipedia.org/wiki/REST
- http://en.wikipedia.org/wiki/List_of_HTTP_status_codes
- http://jacobian.org/writing/rest-worst-practices/

Fundamentals of Basic REST API Design

When designing a REST API, use the appropriate HTTP method for each type of action:

Statement	HTTP method	Rough SQL equivalent
Looking at resource	GET	SELECT
Adding resource	PUT	INSERT
Updating resource	POST	UPDATE or INSERT
Deleting resource	DELETE	DELETE

These are HTTP status codes common to REST API design and how they relate the HTTP methods:

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

HTTP Status Code	Success/Failure	Meaning
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.

Implementing a Simple JSON API

Even on sites that are almost entirely HTML, it's not uncommon for developers to need simple JSON APIs tied to the standard Django authentication system for providing easy access to Django models or other resources. In this sort of limited case, **django-tastypie** and **django-rest-framework** are overkill, adding a lot of machinery for no reason.

This is when **django-braces** excels as a lightweight API library.

For example, using the `flavors` app example, let's say that after a flavor is created via the HTML form, we want to provide the capability to update the flavor via an AJAX call. We'll want to support the HTTP GET and POST methods, and incorporate the same, proven `FlavorForm` we use in other chapters:

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

from .models import Flavor

class FlavorForm(forms.ModelForm):

    class Meta:
        model = Flavor
        fields = ('title', 'slug', )
```

This is where the advantage of using `Meta.fields` over `Meta.excludes` becomes obvious. The form explicitly accepts values only for `title` and `slug`, meaning that we don't have to worry about other fields being modified, such as `created`, `modified`, or `owner`.

WARNING: Always Use Meta.fields. Never Use Meta.excludes.

The use of `Meta.excludes` is considered a security risk. We can't stress it strongly enough. Do not use `Meta.excludes`.

Let's create the first part of our view by writing a CBV that has a `get()` method which corresponds *exactly* with the HTTP GET method:

```
# flavors/views.py
from django.views.generic import View
from django.views.generic.detail import SingleObjectMixin

from braces.views import JSONResponseMixin

from .models import Flavor
from .forms import FlavorForm

class FlavorObjectApiView(JSONResponseMixin, SingleObjectMixin,
                                View):
    model = Flavor

    def get(self, request, *args, **kwargs):
        """ Returns a single JSON object in a JSON list
                representing the Flavor.
        """
        instance = [self.get_object()]
        return self.render_json_object_response(instance)
```

`SingleObjectMixin` provides the `get_object` method, which is what CBVs such as `django.views.generic.DetailView` use to get a single model instance based on either a slug or PK provided by the request. In our example, we place the result into a list to match Django's behavior for supplying JSON responses.

The `JSONResponseMixin` provided by django-braces comes with the `render_json_object_response()` method, which converts lists of Django model instances (i.e. QuerySets) into JSON and then puts the result into a `django.http.HttpResponse` object.

While the GET part of our view was interesting, the `post()` implementation for the HTTP POST method is fascinating:

```
    # add to the FlavorObjectApiView view
    def post(self, request, *args, **kwargs):
        """ Updates a single object in a JSON list
            representing the Flavor
        """
        instance = self.get_object()
        form = FlavorForm(self.request.POST,
                instance=instance)
        if form.is_valid():
            instance = form.save()
            return self.render_json_object_response([instance])

        response = self.render_json_response(form.errors)
        response.status_code = 400  # Bad request
        return response
```

Again, we start the method by using `SingleObjectMixin's get_object()` method to fetch the flavor object. Then, because this method is *only* called on HTTP POST requests, we can pass in a `self.request.POST` object into the `FlavorForm` along with the model instance. At this point, we reach a decision point:

- If `form.is_valid() == True`, we save the form and then convert the model instance to a JSON response.

- If `form.is_valid() == False`, we render the form.errors as a JSON response, change the status code to 400 Bad Request, and send the results back.

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

```
from django.conf.urls.defaults import patterns, url

from flavors import views

urlpatterns = patterns('',
    url(
        regex=r'^api/(?P<slug>[-\w]+)/$',
        view=views.FlavorObjectApiView.as_view(),
        name='flavor_object_api'
    ),
)
```

This is great! The next page covers moving the code to a Mixin, making it easy to reuse the code across multiple pages.

Reusing Our Simple JSON API

The `FlavorObjectApiView` is interesting, but there is a good chance that we may need to create several similar views. This is the use case for mixins and aptly illustrates the power of Django CBVs.

In order to reuse the code for `FlavorObjectApiView`, here we're going to replace the ***flavors*** app specific code with more generic code and move it to our ***core*** app.

```
# core/views.py
from django.views.generic.detail import SingleObjectMixin

from braces.views import JSONResponseMixin

class ObjectApiMixin(JSONResponseMixin, SingleObjectMixin):

    @property
    def model(self):
        msg = "model needs to be defined for ObjectApiView"
        raise NotImplementedError(msg)

    @property
    def form_class(self):
        msg = "form_class not defined for ObjectApiView"
        raise NotImplementedError(msg)

    def get(self, request, *args, **kwargs):
        """ Returns a single JSON object in a JSON list
            representing the model instance
        """
        instance = [self.get_object()]
        return self.render_json_object_response(instance)

    def post(self, request, *args, **kwargs):
        """ Updates a single object.
            If successful, returns 200 and serialized instance.
```

```
            If not, returns 400 and serialized form errors.
        """
        instance = self.get_object()

        form = self.form_class(self.request.POST,
                               instance=instance)
        if form.is_valid():
            instance = form.save()
            return self.render_json_object_response([instance])

        response = self.render_json_response(form.errors)
        response.status_code = 400
        return response
```

The most important change is the inclusion of two properties: `model` and `form_class`. These properties `raise NotImplemented` if called. We do this to provide more helpful error messages if we neglect to include the required `model` and `form_class` attributes during usage of this Mixin.

Speaking of usage: suppose we replace the current `FlavorObjectApiView` with the following. Imports are included here for the sake of clarity:

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

from core.views import ObjectApiMixin
from .models import Flavor
from .forms import FlavorForm

class FlavorObjectApiView(ObjectApiMixin, View):
    model = Flavor
    form_class = FlavorForm
```

Now, if we wanted to support more resources displayed this way, it's a matter of defining new views using existing models and form classes.

Please note that `ObjectApiMixin` does not inherit directly from `View`. We do this because mixins should never be able to run on their own and there might be other view classes that could run with `ObjectApiMixin`.

API Creation Libraries

If you need to expose a lot of model resources as RESTful APIs, you might consider using an API framework. As mentioned earlier, we can recommend both django-tastypie and django-rest-framework. They are powerful libraries for exposing resources for a variety of uses, mobile development to service oriented architecture and much more. These libraries and others include built-in functionality like throttling, pagination, documentation, OAuth support, and the power of their active communities is invaluable.

We prefer to use these tools when we require a dedicated REST API that works directly with Django models. For example, we need to have a API version of the site and need throttling to block malicious users, or are provided read-only access for all the models on our site.

On the other hand, these libraries do become challenging when you are trying to display data that is not based off of models, or just need to support a few API calls.

Summary

In this chapter we covered:

- Fundamentals of Basic REST API Design
- Implementing a Simple JSON API
- Reusing Our Simple JSON API
- API Creation Libraries

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

Templates: Best Practices

12

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?

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 critical expression of content, and therein lies the practices and patterns for this chapter.

Before you read this chapter, we recommend that you have read about the syntax of the Django template language: https://docs.djangoproject.com/en/dev/topics/templates/

Exploring Template Inheritance

Let's dive into some useful details about template inheritance. We're first going to explore a very simple ***base.html*** file then inherit it from another template. 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>`".

While this file is too simple to serve in a real project, it's ideal for getting across some of the options available with template inheritance. See below:

```
{% 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>
```

Our example relies on just three template tags, which we'll summarize in a table on the next page.

base.html template tags table

Template Tag	Purpose
{% load %}	Loads the staticfiles built-in template tag library
{% block %}	Since base.html 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.

To demonstrate the ***base.html*** file in use, we'll use a simple ***about.html*** template. This file will extend or inherit the base.html template in order to display the following:

- 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.

```
{% 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 %}
```

Let's assume we've got a view ready to render this template. We call it and the resulting rendered html is what you see on the next page...

```
<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
{% extends %}	Informs Django that about.html is inheriting or extending from base.html
{% block %}	Since about.html is a child template, block overrides the content provided by base.html. For example, this means our title will render as <title>Audrey and Daniel</title>.
{{ block.super }}	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 about.html template, this will render Two Scoops .

Take 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 }}`.

{{ block.super }} gives the power of control

Let's imagine that we have a template which must inherit everything from the ***base.html*** but must replace 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 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:

1. A template using both the ***project.css*** link and a custom CSS link.
2. A *dashboard* template with a custom CSS link to ***dashboard.css***. The template excludes the project.css.
3. A template that uses just the standard ***project.css*** link.

Example 1: Template using both project.css and a custom link

```
{% extends "base.html" %}
{% block stylesheets %}
    {{ block.super }} {# this brings in project.css #}
    <link rel="stylesheet" type="text/css"
        href="{% static "css/custom" %}" />
{% endblock %}
```

Example 2: Dashboard template that excludes the project.css link.

```
{% 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 %}
```

Example 3: Template just linking the project.css file.

```
{% extends "base.html" %}
{% comment %}
    By not using {% block stylesheets %}, this template uses 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.

Flat Is Better Than Nested

Did you know that the Zen of Python says "flat is better than nested"? Don't forget that when you're coding your Django templates.

TIP: The Zen of Python

At the command line, do the following:

```
$ python
>>> 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.

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.

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:

2-tier template architecture example

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

3-tier template architecture example

```
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
```

As can be seen, in the 2-tier architecture, everything inherits from the root base.html file. In the 3-tier architecture, apps inherit from a ***base_<app_name>.html*** template. This is extremely useful when we want HTML to look or behave differently for a particular section of the site that groups functionality.

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:

```
{% 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:

```
{# 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 the *Finding and Reducing Bottlenecks* chapter for more details.

Useful Things to Consider

The following are a series of smaller things we keep in mind during template development.

Our Naming Practices

- **We don't use the 'dash' character in blocks to link values together.** We prefer the use of underscore because many text editors won't let you double click select the associated words.
- **We rely on clear, intuitive names for blocks.** `{% block extra_js %}` is good.
- **We include the name of the block tag in the endblock.** Never write just `{% endblock %}`, include the whole `{% endblock extra_js %}`.
- **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 %}`.

Limit Looping

TODO: Looping over large querysets in your templates is usually a bad idea. Show example snippet of looping and checking for various "if" conditions, and how that code can be improved by moving it to a model manager.

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:

```
# settings/local.py
TEMPLATE_STRING_IF_INVALID = 'INVALID EXPRESSION: %s'
```

Use URL Names Instead of Hardcoded Paths

A common developer mistake is to hardcode URLs in templates like this:

```
<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:

```
<a href="{% url 'flavors_list' %}">
```

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, not `{{ object_list }}` and `{{ object }}`. Which means both of the the following template examples will work:

```
{# toppings/topping_list.html #}
{# Using implicit names #}
<ol>
{% for object in object_list %}
    <li>{{ object }} </li>
{% endfor %}
</ol>
```

```
{# toppings/topping_list.html #}
{# Using explicit names #}
<ol>
{% for topping in topping_list %}
   <li>{{ topping }} </li>
{% endfor %}
</ol>
```

Avoid Coupling Styles Too Tightly to Python Code

Aim to be able 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.

Using Javascript Templates in Django Templates

Many developers enjoy the use of Javascript templates libraries such as handlebars.js, since they are often a component of sophisticated Javascript frameworks. Unfortunately, Javascript templates often conflict with how Django templates render context variables. Fortunately, version 1.5, Django provides the `{% verbatim %}` built-in template tag. Simply wrap the Javascript language and generate verbatim content:

```
{% verbatim %}
    Ice Cream {{if melted}}melted{{/if}}.
{% endverbatim %}
```

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 packages template directory should go into app directly. We'll explore this in the chapter on *How to Release Your Own Django Packages*.

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 the *Tradeoffs of Replacing Core Components* chapter for a case study about replacing the Django template engine with Jinja2.

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.

In the next chapter we'll examine Template Tags and Filters.

Template Tags and Filters

13

"Please stop writing so many template tags. They are a pain to debug."

-- Audrey Roy, while debugging Daniel Greenfeld's code.

Django provides dozens of default template tags and filters, all of which all share several 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 templatetags.

Our Problems With Template Tags and Filters

Odds are that the opening quote for this chapter was a pretty good sign that we find template tags and filters problematic. We've done a lot of work with them, and have found that they can be the source of unpleasantness:

- Template tags and to a lesser extent filters of any complexity can be challenging to debug.
- Template tags and filters can have a significant performance cost.
- It can be difficult to consistently apply the same effect as a template tag on alternative output formats used by an API or in PDF/CSV generation.

These days, we're very cautious about adding new template tags and filters, and 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 which adds the method/property.

When should you write new template tags or filters?
Only when you must. When there is no other option. Our preference for writing them is that they are only responsible for rendering of HTML.

Custom template tags should be written with caution. Why? Well, they only work in templates, the writing of compiler/renderers is complex because 'they can do anything', and you have to make sure that the template tag is thread-safe. See https://docs.djangoproject.com/en/dev/howto/custom-template-tags/#thread-safety-considerations

You could use the `{% include "app_name/_box_items.html" %}` built-in template tag. The only downside of this approach is that unless you also pass in the `'only'` argument, the included template is passed the entire context of the parent, which can be confusing in complex template hierarchies.

Filters are okay. They are essentially just functions with decorators that make usable inside of Django templates. This means can be called as normal functions (although we prefer to have our filters call functions imported from helper modules).

TIP: We Do Use Template Tags

Interestingly enough, Daniel has been involved at least three prominent libraries that make extensive use of template tags.

Naming Your Template Tag Modules

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 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.

Loading Your Template Tag Modules

In your template, right after `{% extends "base.html" %}` is where you load them:

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

{% load flavors_tags %}
```

Simplicity itself! Explicit loading of functionality! Hooray!

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:

```
# 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 templatetag library into each and every template loaded by `django.template.Template`. Which 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 templatetag 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. Which means it's typically placed in an `__init__` module or template tag, 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.

Tradeoffs of Replacing Core Components

14

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. Even Instagram says these days on Forbes.com it's completely unnecessary: http://www.forbes.com/sites/limyunghui/2012/04/09/inspiring-insights-by-instagram-ceo-kevin-systrom-the-man-who-built-a-1-billion-startup/2/

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'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?

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!" Not 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!"

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.

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.

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 put into model methods might go into views.
- Template tags and filters should be a last resort. We'll cover this in more detail in the *Models, Revisited* and the *Template Tags and Filters* chapters, so don't worry if you don't fully understand this yet.

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:

```
# flavors/views.py
import os
from django.conf import settings
from django.http import HttpResponse

from jinja2 import Template, 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)

def big_syrup_list(request):
    template = JINJA2_ENV.get_template('big_syrup_list.html')
    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.

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!

Working With the Django Admin

15

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 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.

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 that.

Admin Customization vs. New Views

It's usually not worth it to heavily customize the 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.

Secure It Well

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 the *Security* chapter for details.

Dealing With the User Model

16

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.

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:

```
# Stock user model definition
>>> from django.contrib.auth import get_user_model
>>> get_user_model()
<class 'django.contrib.auth.models.User'>

# 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. Which is new in Django 1.5 and a radical departure from earlier versions of Django.

Use settings.AUTH_USER_MODEL for foreign keys to User
From Django 1.5 onwards, the advised way to attach ForeignKey, OneToOneField, or ManyToManyField to user is as follows:

```
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)
```

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 it's 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 we wrote this book, 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.

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:

```
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`.

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`:

```
# 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:

```
AUTH_USER_MODEL = 'profiles.KarmaUser'
```

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 (https://docs.djangoproject.com/en/1.5/topics/auth/customizing/#a-full-example). It doesn't do exactly what we want, but we can modify. Specifically:

We'll need to add `PermissionMixin` 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.

```
# 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.

```
# 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:

```
# settings/base.py
AUTH_USER_MODEL = 'profiles.TwoScoopsUser'
```

Upon `syncdb` this will create a new User 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
Email address: hello@twoscoopsofdjango.com
Favorite topping: rainbow sprinkles
Password:
Password (again):
Superuser created successfully.
```

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:

```
# 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.

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, can either continue with the current way of doing things or customize the actual user model.

The next chapter begins our three chapter series on Testing, starting with *Testing Stinks and Is a Waste of Money*.

Django's Secret Sauce: Third-Party Packages

17

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 open-source packages available for your Django projects. Third-party packages do an incredible amount of work for you. These packages have been written by people from all walks of life, and 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.

Examples of Third-Party Packages

Appendix A: Packages Mentioned In This Book covers all of the packages mentioned throughout this book. 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 have names prefixed with "django-", but there are many exceptions.)

Know About the Python Package Index

The **Python Package Index** (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 (**PyPI**) 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 PyPI. Most Python and Django packages are downloadable from PyPI as well via pip.

Know about DjangoPackages.com

Django Packages (www.djangopackages.com) is a directory of reusable apps, sites, tools and more for your Django projects. Unlike the Python Package Index, it doesn't store the packages themselves, instead providing a mix of hard metrics gathered from PyPI, 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.

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.

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 The Optimal Django Environment Setup for more details.

Package Requirements

As we mentioned earlier in 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 reference *Appendix A: Third-Party Packages We Use*.

Wiring Up Django Packages: The Basics

When you find a third-party package that you want to use, follow these steps:

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.

2. Add Package and Version Number to Your Requirements

If you recall from the chapter on *Django's Secret Sauce: Third-Party Packages*, a `requirements/_base.txt` file looks something like this (but probably longer):

```
https://www.djangoproject.com/download/1.5c1/tarball/
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'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*:

```
-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 projects depend.

3. Install the Requirements Into Your Virtualenv

Assuming you already in a working virtualenv and are at the `<repo_root>` of your project:

```
$ pip install -r requirements/_base.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.

4. Follow the Package's Installation Instructions Exactly

Resist the temptation to skip steps unless you're very familiar with the package. Since Django 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.

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.

How To Create and Release 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:

1. 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.).
2. Release the package on PyPI, the Python package index (http://pypi.python.org). Follow the PyPI submission instructions: http://2scoops.org/submit-to-pypi
3. Add the package to Django Packages: http://www.djangopackages.com
4. 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 *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.

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://www.slideshare.net/audreyr/django-package-thunderdome-by-audrey-roy-daniel-greenfeld

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.

Scope

Your package's scope should be tightly focus 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.

Documentation

A package without documentation is a pre-alpha package. Docstrings don't suffice as documentation.

As described in the *Documentation: Be Obsessed* chapter, 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 Sphinx 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.

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.

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.

Community

Great Django packages often end up receiving contributions from other developers in the Django community. All contributors should receive attribution in a `CONTRIBUTORS.rst` file and/or in a `README.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.

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.

Availability on PyPI

All major and minor releases of your package should be available for download from PyPI (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.

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 http://opensource.org/licenses/category for the license that you choose.

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.

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.

Testing Stinks and Is a Waste of Money!

18

There, got you to this chapter.

Now you have to read it.

We'll try and make this chapter interesting.

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 intelligence and 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 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. 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.

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 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 2012, 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:

1. Increase the version in a local instance of Django Packages.
2. Run the tests.
3. Fix any errors that are thrown by the tests.
4. 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. We doubt any lives are on the line with Django Packages, but as it does have a decent profile in the developer community, which means any time a significant error comes up in production we certainly hear about it on Twitter and email.

This is the benefit of having tests.

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.

THIRD-PARTY PACKAGES

We prefer 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% 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 it's useful for any Python related project.

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 resource.

Step 1: Set Up A Test Runner

In our settings directory, we create a *test.py* module and add the following:

```
"""Local test settings and globals which allows us to run our
test suite locally."""

from settings.base import *

########## TEST SETTINGS
TEST_RUNNER = 'discover_runner.DiscoverRunner'
TEST_DISCOVER_TOP_LEVEL = PROJECT_ROOT
TEST_DISCOVER_ROOT = PROJECT_ROOT
TEST_DISCOVER_PATTERN = "*"

########## 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.

Step 2: Run Tests and Generate Coverage Report

Let's try it out! In the command-line, at the <project_root>, type:

```
$ 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:

```
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.

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*>:

```
$ coverage html --include="<project-root>*" --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 colored red.

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!

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 it 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 a 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.

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 ***forms.py***, ***models.py***, ***views.py*** modules. Tests that apply to forms go into ***forms.py***, model tests go into ***models.py***, and so on.

Here's what it looks like:

```
popsicles/
  tests/
      __init__.py
      forms.py
      models.py
      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.

Summary

All of this might seem silly, but testing can be very serious business. In a lot of developer groups this game 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*.

Documentation: Be Obsessed

19

Given a choice between ice cream and writing great documentation, most Python developers would probably choose to write the documentation, believe it or not. That's how obsessed the Python community is with documentation.

When you have great documentation tools like Sphinx, you actually can't help it but want to add docs to your projects.

Formatting Your Docs

You'll want to learn and follow the standard Python best practices for documentation.

Use reStructuredText Markup To Write Up Python Docs

These days, reStructuredText (RST) is the most common markup language used for documenting Python projects. It looks like this:

```
Section Header
==============

Subsection Header
-----------------

#) An enumerated list item

#) Second item
```

Study the documentation for reStructuredText and learn at least the basics: http://docutils.sourceforge.net/rst.html

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.

We recommend pip installing Sphinx systemwide, as you'll want to have it handy for every Django project.

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.

What Docs Should Your Django Project Contain?

Developer facing documentation are notes and guides that developers need in order to maintain a project. This includes notes on installation, deployment, architecture, how to run tests or submit pull requests, and more. We've found it really helps to place this documentation in all our projects, private or public.

On the next page we provide a table that describes what we consider the absolute minimum documentation.

Required Project Documentation

Filename or Directory	Reason	Description or Instructions
README.rst	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.
docs/	Your project documentation should go in one, consistent location. This is the Python community standard.	A simple directory
docs/deployment.rst	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.
docs/installation.rst	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.
docs/architecture.rst	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.

Using a Wiki or 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.

Summary

In this chapter we went over the following:

- Introduced the use of reStructuredText and Markdown to write documentation in plaintext format.
- Introduced the use Sphinx to render your documentation in HTML or PDF formats.
- Advised on the required project documentation.

In the next chapter we dive into the *Finding and Reducing Bottlenecks*.

Finding and Reducing Bottlenecks 20

This chapter covers a few basic strategies for identifying bottlenecks and speeding up your Django projects.

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.

Get the Most Out of Your Database

First, Frank Wiles of Revsys taught us that there are two things that should never go into any large site's relational database:

1. **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.
2. **Don't add ephemeral data.** 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

Frank actually taught us three things to never store in a database, but storing of binary data in databases is addressed by `django.db.models.FileField`, which does the work of storage of binary data on file servers like AWS CloudFront or S3 for you.

Second, 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.

Getting the Most Out of PostgreSQL

If using PostgreSQL, be certain that PostgreSQL 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

We also recommend the following books:

- PostgreSQL 9.0 High Performance

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 my MySQL experts to help you:

- High Performance MySQL

TIP: How to tell what your database is really doing:

For PostgreSQL, a great and current (as of Django 1.5) guide for achieving this understanding is Craig Kerstiens post on the subject:

- www.craigkerstiens.com/2012/10/01/understanding-postgres-performance/
- www.craigkerstiens.com/2013/01/10/more-on-postgres-performance/

For MySQL, we recommend using and understanding the EXPLAIN command:

- http://dev.mysql.com/doc/refman/5.6/en/explain.html

Use django-debug-toolbar To Find Query-Heavy Pages

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.

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 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 contain.

Once you are looking into the number of queries being done, figure out ways to reduce them. Some of the things you can attempt:

- Try using `select_related()` in your ORM calls, as 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`. Beware queries that get too large!

- If the same query is being generated more than once per view, 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 https://docs.djangoproject.com/en/1.5/topics/testing/overview/#django.test.TestCase.assertNumQueries
- Read up on Django ORM optimization: https://docs.djangoproject.com/en/1.5/topics/db/optimization/

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/

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.

THIRD-PARTY PACKAGES: We use this to aid in performance analysis

- django-debug-toolbar - This critical development tool can also be an invaluable aid in cache analysis. We 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.
- newrelic is a third-party library provided by 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.

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 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 time. 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.

Compression and Minification of HTML, CSS, and Javascript

When a browser renders a web page, it usually has to load HTML, CSS, Javascript, and images files. Each of these files consumes the bandwidth of the user, slowing down page loads. One way to reduce bandwidth consumption is via compression and minification, and 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 takes 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 building 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 are **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

- https://docs.djangoproject.com/en/1.5/ref/templates/builtins/#spaceless
- https://docs.djangoproject.com/en/1.4/ref/middleware/#module-django.middleware.gzip
- www.djangopackages.com/grids/g/asset-managers/

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 www.varnish-cache.org/.

Content Delivery Networks like Akamai and Amazon Cloudfront serve static media such as images, video, CSS, and Javascript files so your webserver can focus on delivering content.

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:

- 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

Summary

In this chapter we explored a number of bottleneck reductions strategies including:

- Should you even care?
- Getting the most out of databases
- Using django-debug-toolbar
- 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 project.

Security Best Practices

21

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 core developer team 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.

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.

Use django-secure

One of the core Django developers, Carl Meyer, wrote an application called django-secure which checks to make sure a number of absolutely critical settings are set appropriately.

Follow the instructions at https://github.com/carljm/django-secure/.

Use SSL/HTTPS in Production

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.

> **TIP: Jacob Kaplan-Moss on HTTPS vs HTTP**
>
> One of the BDFL of Django told us, "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."

Setting up SSL/HTTPS for production sites is outside the scope of this book.

Additional reading: https://docs.djangoproject.com/en/1.5/topics/security/#ssl-https

Always Use CSRF Protection With Forms That Modify Data

Django comes with Cross-Site-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 cases 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. Which means they don't trigger Django's CSRF protection.

Prevent Against Cross Site Scripting (XSS) Attacks

XSS attacks occurs when users enter malignant Javascript that is then rendered into a template directly. 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 it'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.org/django-docs-on-html-scraping
- http://en.wikipedia.org/wiki/Cross-site_scripting

Don't Run Arbitrary Python Code

Beware of the `eval()`, `exec()`, and `execfile()` built-ins.

There was once a project where the Django requests coming into the site were 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.

Which meant that arbitrary Python code could be run on the site at any time.

Needless to say, upon discovery the alarm was raised and the critical security flaw was quickly removed. This just goes to show that no matter how secure Django might be, we always need to be aware that certain shortcuts are incredibly dangerous.

Don't use ModelForms.Meta.excludes

When using ModelForms, always use `Meta.fields`. Never use use of `Meta.excludes`. The use of Meta.excludes is considered a security risk. **We can't stress this strongly enough. Do not use Meta.excludes.**

One common reason we want to avoid the `Meta.excludes` attribute is that it's 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:

```
# 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:

```
# 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')
```

The **right way** to define ModelForm fields:

```
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 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:

```
# 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 hard, 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 Ice Cream Stores are more secure.

Beware of SQL Injection Attacks

Django's ORM will handle most, if not all, of the database queries necessary for your project. The ORM generates properly-escaped SQL which will protect your site from users attempting to execute malignant, arbitrary SQL code.

Django also lets you touch the database more directly through raw SQL. Use these features sparingly, and be especially careful to escape your SQL code properly.

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.

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.

Only Allow 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.

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.

Limit 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 PaaS might not give you fine-grain control over web server configuration.

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 BDFL Jacob Kaplan-Moss says (paraphrased), "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, I can tell which version of Django you're using, sometimes down to the point level, by examining the content of /admin/ URL on a project."

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.

Summary

Please use this chapter as a starting point for Django security, not the ultimate reference guide. Django comes with a good security record due to the diligence of its community and attention to detail. This is one of those areas where it's a very good idea to ask for help.

Logging: Tips and Tools

22

Logging is one of those areas of Django which really benefits from being part of the Python programming language. Because Django shares the same logging tool as non-Django related tools frequently used in web projects, tracking what is going on across your project at all levels is possible.

Don't Use Print Statements

It's tempting to put print statements all over the place while debugging, but don't get into this habit. Here's why:

- 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.

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.

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

Useful Third-Party Tools

- Sentry (https://www.getsentry.com/) aggregates errors for you.
- loggly.com simplifies log management and provides excellent query tools.

Summary

Logging records the happenings of your project as they occur - but only if you take the time and effort to add logging to your project.

In the next chapter we'll discuss *Signals*, which become much easier to write and debug when using Logging.

Signals: Use Cases and Avoidance Techniques

23

The Short Answer: Use signals as a last resort.

The Long Answer: Often when new Djangonauts first discover signals, they get signal-happy. They start sprinkling signals everywhere they can and feeling like real experts at Django. They claim mastery of the Observer Pattern, Aspect Oriented Programming, and a bunch of other powerful buzzwords.

After coding this way for awhile, 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 are 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.

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, especially `save()`.

- 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.

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.

Validate Your Model Elsewhere

If you're using a `pre_save` signal to trigger input cleanup for a specific model, try write a custom validator for your field(s) instead.

If validating through a ModelForm, try overriding your model's `clean()` method instead.

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.

What About Those Random Utilities? 24

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:

```
core/
    __init__.py
    managers.py  # contains the custom model manager
    models.py
    views.py  # Contains the custom view mixin
```

TIP: Django Apps 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 at least one of the following:

- Have non-abstract models in core.

- Need admin auto-discovery to work 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:

```
from core.managers import PublishedManager
from core.views import IceCreamMixin
```

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 likes 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:

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.

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.

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.

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.

django.utils.translation

Much of the non-English speaking world appreciates use of this tool, as it provides Django's i18n support.

Summary

We learned the practice of putting often reused files into utility packages while writing code in C++ and Java. We carried this practice into Python and 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.

Deploying Django Projects

25

Deployment of Django projects an in-depth topic that could fill an entire book on its own. Here, we touch upon deployment at a high-level.

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 1) Gunicorn behind a Nginx proxy, and 2) 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.
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.

WARNING: Do not use mod_python

As of June 16th, 2010 the mod_python project is officially dead. The previous maintainer of mod_python and the official Django documentation explicitly warns against using mod_apache 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.

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.

Which means, 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 (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 (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 (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.

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.

Where and How to Ask Django Questions

26

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!

What to Do When You're Stuck

Follow these steps to increase your chances of success:

1. 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.

2. Read through the documentation in detail, to make sure you didn't miss something.

3. See if someone else has had the same issue. Check Google, mailing lists, and StackOverflow.

4. 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.

5. Still don't get an answer after a couple of days? Try asking on the django-users mailing list or in IRC.

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.

1. 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.

2. Paste a relevant code snippet and traceback into https://gist.github.com/ (or another pastebin).

3. 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!

4. 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.

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.

10 Easy Ways To Participate

1. Attend Python and Django user group meetings. Join all your local groups listed 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 #pyladies, 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/

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.

Closing Thoughts

27

[Illustration on a door closing]

While we've covered a lot of ground here, this is also just the tip of the iceberg. 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.

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.

Package	Reason	Link
Django	The web framework for perfectionists with deadlines	http://djangoproject.com
Pillow	Friendly installer for the Python Imaging Library	http://pypi.python.org/pypi/Pillow
South	Easy database migrations for Django	http://south.readthedocs.org
celery	Distributed Task Queue	http://www.celeryproject.org/
coverage	Checks how much of your code is covered with tests	
django-braces	Drop-in mixins that really empower Django's Class Based Views	
django-celery	Celery integration for Django	
django-crispy-forms	Rendering controls for Django forms	

Package	Reason	Link
django-pipeline	Compression of CSS and JS. Use with cssmin and jsmin packages	
django-debug-toolbar	Display panels used for debugging Django HTML views	
django-discover-runner	Test runner based off unittest2	
django-extensions	Provides shell_plus management command and a lot of other utilities.	
django-floppy-forms	Form field, widget, and layout that can work with django-crispy-forms.	
django-haystack	Full-text search that works with SOLR, Elasticsearch, and more.	
django-heroku-memcacheify	Easy Memcached settings configuration for Heroku.	
django-heroku-postgresify	Easy PostgreSQL settings configuration for Heroku.	
django-model-utils	Useful model utilities including a time stamped model	
django-skel	Django project template optimized for Heroku deployments	https://github.com/rdegges/django-skel
django-social-auth	Easy social authentication and registration for Twitter, Facebook, Google, and lots more.	
django-registration	Email and username registration made easy, but it lacks sample templates.	

Package	Reason	Link
django-secure	Helps you lock down and your site's security	
django-tastypie	Expose Model and non-Model resources as a RESTful API	
psycopg2	PostgreSQL database adapter	
requests	Easy-to-use HTTP library that replaces Python's urllib2 library	http://docs.python-requests.org
sentry	Exceptional error aggregation	http://getsentry.com
Sphinx	Documentation tool	
newrelic	Realtime logging and aggregation platform	http://newrelic.com

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:

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

If you're running Django 1.5, you should see the following output:

```
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 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:

```
$ virtualenv testenv
$ source testenv/bin/activate
```

If you're on Windows, verify that this works:

```
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:

```
(testenv) $
```

On Windows, this will look something like:

```
(testenv) >
```

Did you run into any problems? If so, study the Virtualenv documentation (www.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:

```
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:

```
(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://www.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/

About This Book

Acknowledgements

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 editors. 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, Jeff Triplett for keeping our stuff agnostic, and Preston Holmes for his contributions to the User model chapter.

Malcolm Tredinnick lives in Sydney, Australia and seems to travel internationally more than is probably healthy. A Python user for over 15 years and Django user since just after it was released to the public in mid-2005, he's been a Django core developer since 2006. A user of many programming languages, he still feels Django is one of the better web libraries/frameworks he has used professionally and is glad to see its incredibly broad adoption over the past years.

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*".

Preston Holmes 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.

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.

Alpha Reviewers

During the Alpha release 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

If your name is not on this list please send us an email so we can make corrections!