Django Unleashed
This page intentionally left blank
Django Unleashed

Andrew Pinkham
## Contents

Preface xiii  
Acknowledgments xvii  
About the Author xix  

### I  Django’s Core Features 1

1 Starting a New Django Project: Building a Startup Categorizer with Blog 3  
1.1 Introduction 3  
1.2 Website Basics 4  
1.3 Understanding Modern Websites 5  
1.4 Building Modern Websites: The Problems That Frameworks Solve and Their Caveats 6  
1.5 Django: Python Web Framework 8  
1.6 Defining the Project in Part I 11  
1.7 Creating a New Django Project and Django Apps 13  
1.8 Putting It All Together 21  

2 Hello World: Building a Basic Webpage in Django 23  
2.1 Introduction 23  
2.2 Creating and Integrating a New App 24  
2.3 Building Hello World 25  
2.4 Displaying Hello World 26  
2.5 Controller Limitations: The Advantages of Models and Views 27  
2.6 Removing Our Helloworld App from Our Project 27  
2.7 Putting It All Together 29  

3 Programming Django Models and Creating a SQLite Database 31  
3.1 Introduction 31  
3.2 Why Use a Database? 32  
3.3 Organizing Our Data 32  
3.4 Specifying and Organizing Data in Django Using Models 36
3.5 Using Django to Automatically Create a SQLite Database with manage.py 49
3.6 Manipulating Data in the Database: Managers and QuerySets 56
3.7 String Case Ordering 68
3.8 Putting It All Together 71

4 Rapidly Producing Flexible HTML with Django Templates 73
4.1 Introduction 73
4.2 Revisiting Hello World: The Advantages of Templates 74
4.3 Understanding Django Templates and Their Goals 77
4.4 Choosing a Format, an Engine, and a Location for Templates 77
4.5 Building a First Template: A Single Tag Object 78
4.6 Building the Rest of Our App Templates 90
4.7 Using Template Inheritance for Design Consistency 102
4.8 Using Templates in Python with the Template, Context, and loader Classes 112
4.9 Putting It All Together 118

5 Creating Webpages with Controllers in Django: Views and URL Configurations 121
5.1 Introduction 121
5.2 The Purpose of Views and URL Configurations 122
5.3 Step-by-Step Examination of Django’s Use of Views and URL Configurations 126
5.4 Building Tag Detail Webpage 128
5.5 Generating 404 Errors for Invalid Queries 132
5.6 Shortening the Development Process with Django View Shortcuts 135
5.7 URL Configuration Internals: Adhering to App Encapsulation 143
5.8 Implementing the Views and URL Configurations to the Rest of the Site 148
## Contents

5.9 Class-Based Views 155  
5.10 Redirecting the Homepage 163  
5.11 Putting It All Together 166  

6 Integrating Models, Templates, Views, and URL Configurations to Create Links between Webpages 169  
6.1 Introduction 169  
6.2 Generating URLs in Python and Django Templates 170  
6.3 Using the `url` Template Tag to Build a Navigation Menu 175  
6.4 Linking List Pages to Detail Pages 177  
6.5 Creating Links on the Object Detail Pages 184  
6.6 Revisiting Homepage Redirection 186  
6.7 Putting It All Together 187  

7 Allowing User Input with Forms 189  
7.1 Introduction 189  
7.2 Django Forms as State Machines 190  
7.3 Creating `TagForm`, a Form for Tag Objects 190  
7.4 Building the Forms for `Startup`, `Newslink`, and `Post` Models 206  
7.5 Putting It All Together 210  

8 Displaying Forms in Templates 211  
8.1 Introduction 211  
8.2 Creating a New Template to Create Tag Objects 211  
8.3 Creating a New Template to Update Tag Objects 224  
8.4 Creating a New Template to Delete Tag Objects 226  
8.5 Creating Templates for `StartupForm`, `NewsLinkForm`, and `PostForm` 227  
8.6 Reconsidering Template Inheritance 229  
8.7 Putting It All Together 231  

9 Controlling Forms in Views 233  
9.1 Introduction 233  
9.2 Webpages for Creating Objects 233
9.3 Webpages for Updating Objects 256
9.4 Webpages for Deleting Objects 268
9.5 Putting It All Together 276

10 Revisiting Migrations 279
10.1 Introduction 279
10.2 Last Week’s Episode (Reviewing Chapter 3) 279
10.3 Data Migrations 280
10.4 Schema Migrations 288
10.5 Putting It All Together 296

11 Bending the Rules: The Contact Us Webpage 299
11.1 Introduction 299
11.2 Creating a contact App 300
11.3 Creating the Contact Webpage 301
11.4 Splitting Organizer urls.py 308
11.5 Putting It All Together 310

12 The Big Picture 313
12.1 Introduction 313
12.2 Django’s Core 313
12.3 Webpages with Views and URL Configurations 316
12.4 Generating Webpages Thanks to Models and Templates 317
12.5 Interacting with Data via Forms 318
12.6 Intervening in Control Flow 319
12.7 Moving Forward 319

II Djangos Contributed Libraries 321
13 Django’s Contributed Library 323
13.1 Introduction 323
13.2 Django’s Source Code (and Versioning) 323
13.3 Django’s contrib Code 325
13.4 Content (Not) Covered 327
13.5 Translation 328
13.6 Putting It All Together 329
14 Pagination: A Tool for Navigation 331
  14.1 Introduction 331
  14.2 A Word about URLs: Query versus Path 332
  14.3 Discovering Django Pagination in the Shell 333
  14.4 Paginating the Startup List Webpage 337
  14.5 Pagination of Tag List Webpage Using the URL Path 345
  14.6 Putting It All Together 351

15 Creating Webpages with Django Flatpages 353
  15.1 Introduction 353
  15.2 Enabling Flatpages 353
  15.3 Anatomy of the App 355
  15.4 Building an About Webpage 355
  15.5 Linking to FlatPage Objects 363
  15.6 Security Implications of FlatPages 363
  15.7 Migrations for Sites and Flatpages 365
  15.8 Putting It All Together 371

16 Serving Static Content with Django 373
  16.1 Introduction 373
  16.2 Adding Static Content for Apps 374
  16.3 Adding Static Content for the Project 376
  16.4 Integrating Real CSS Content 377
  16.5 Putting It All Together 381

17 Understanding Generic Class-Based Views 383
  17.1 Introduction 383
  17.2 Building Generic Object Detail Pages 384
  17.3 Why Use Classes for Generic Views? 393
  17.4 Building Generic Object Create Pages 394
  17.5 Replacing CBVs with GCBVs 395
  17.6 Forgoing GCBVs 400
  17.7 Adding Behavior with GCBV 401
  17.8 Putting It All Together 416

18 Advanced Generic Class-Based View Usage 417
  18.1 Introduction 417
  18.2 Rapid Review of GCBV 418
18.3 Globally Setting Template Suffix for Update Views 419
18.4 Generating Pagination Links 419
18.5 Re-creating PostDetail with DateDetailView 426
18.6 Switching to GCBVs with PostGetMixin in Post Views 429
18.7 Making PostGetMixin Generic 432
18.8 Fixing NewsLink URL Patterns and Form Behavior 438
18.9 Putting It All Together 449

19 Basic Authentication 451
19.1 Introduction 451
19.2 Configuring Logging 452
19.3 Sessions and Cookies 456
19.4 auth App Anatomy: The Basics 457
19.5 Adding Login and Logout Features 458
19.6 Putting It All Together 472

20 Integrating Permissions 473
20.1 Introduction 473
20.2 Understanding contenttypes and Generic Relations 473
20.3 auth App Anatomy: Permission and Group Models 476
20.4 Protecting Views with Permissions 483
20.5 Conditionally Displaying Template Links 496
20.6 Displaying Future Posts in the Template 497
20.7 Putting It All Together 500

21 Extending Authentication 501
21.1 Introduction 501
21.2 auth App Anatomy: Password Views 501
21.3 Changing Passwords 503
21.4 Resetting Passwords 506
21.5 Disabling Accounts 513
21.6 Creating Accounts 517
21.7 URL Cleanup 544
21.8 Anatomy of the App: Full Dissection  545
21.9 Putting It All Together  547

22 Overriding Django’s Authentication with a Custom User  549
  22.1 Introduction  549
  22.2 Creating a User Profile  550
  22.3 Custom User  558
  22.4 Data Migrations  568
  22.5 Adding an Author to Blog Posts  572
  22.6 Putting It All Together  576

23 The Admin Library  577
  23.1 Introduction  577
  23.2 A First Look  577
  23.3 Modifying the Admin Controls for Blog Posts  581
  23.4 Configuring the Admin for the User Model  593
  23.5 Creating Admin Actions  616
  23.6 Putting It All Together  618

III Advanced Core Features  619

24 Creating Custom Managers and Querysets  621
  24.1 Introduction to Part III  621
  24.2 Introduction to Chapter 24  621
  24.3 Custom Managers and Querysets  622
  24.4 Fixtures  624
  24.5 Management Commands  627
  24.6 Putting It All Together  648

25 Handling Behavior with Signals  649
  25.1 Introduction  649
  25.2 Apps and AppConfig  650
  25.3 Signals  652
  25.4 Putting It All Together  660

26 Optimizing Our Site for Speed  661
  26.1 Introduction  661
  26.2 Profiling  662
  26.3 Limiting Database Queries  663
  26.4 Changing Database Behavior Internally  679
26 Changing Performance Globally 681
26.6 Putting It All Together 685

27 Building Custom Template Tags 687
27.1 Introduction 687
27.2 Custom Template Filters 688
27.3 Custom Template Tags 690
27.4 Putting It All Together 706

28 Adding RSS and Atom Feeds and a Sitemap 707
28.1 Introduction 707
28.2 RSS and Atom Feeds 707
28.3 Sitemaps 715
28.4 Putting It All Together 724

29 Deploy! 725
29.1 Introduction: Understanding Modern Deployments 725
29.2 Preparing for Deployment 726
29.3 Deploying to Heroku 738
29.4 Adding Backing Services 741
29.5 Putting It All Together 748

30 Starting a New Project Correctly 749
30.1 Introduction 749
30.2 Preparing a Project 749
30.3 Building the Project 752
30.4 The Road Ahead 754

IV Appendixes 755

A HTTP 757
B Python Primer 761
C Relational Database Basics 765
D Security Basics 769
E Regular Expressions 771
F Compilation Basics 773
G Installing Python, Django, and Your Tools 775

Index 779
Preface

In early 2013, a startup in Austin, Texas, approached me to work on a banking application using Django. My experience with Django was limited: I had tried to use the tool in 2009 but felt that the learning curve was steep. I wanted to give Django a try but did not have enough time to learn how to use it given the project’s time constraints (which was fine: we were forced to use PHP anyway). When I looked at Django again in 2013, I discovered that it had become far more accessible. For certain, those four years had seen Django improve by leaps and bounds. However, I had also gained key knowledge working with web frameworks.

At the end of the project in 2013, I was asked by a different group to take what I had learned and teach its engineers how to program Django. I liked the work enough that I started creating a series of videos based on the material. During a test showing of the videos, one of my reviewers casually commented that the material would be more suitable and more approachable as a book. I still have a hard time believing that such an innocent comment resulted in a year and a half of such intense work, but that is the origin of this book: an off-hand comment.

This book is the book I wish I’d had in 2009 and in 2013. It is a how-to book that teaches you how to build a webpage from scratch using Django. The first part of the book (the first 12 chapters) are for my 2009 self. It answers the basic questions that I had when I started learning Django, and it explains the basics of web frameworks and websites. I think of the remaining chapters as a response to my 2013 self. They address the needs of more experienced users. Related materials are available at https://django-unleashed.com. I hope you find this book useful.

Is This Book for Me?
This book is meant for two types of people:

1. Programmers who have never built a website before and do not know how web frameworks operate
2. Programmers who have dabbled or used the basics of Django, and who would like to hone their skills and take advantage of Django’s intermediate features

The book thus caters to both beginners and intermediate users. The only knowledge assumed is basic programming knowledge and Python.
What This Book Contains
This book is a hands-on, single example: we build and deploy a fully functional website over the course of the 30 chapters. Each chapter covers a single part of Django and is the logical next step to building our website while learning how to use Django.

Part I, Django’s Core Features, is an introduction to websites, web frameworks, and Django. We assume knowledge of programming and Python, but absolutely no knowledge of the internals of back-end web programming. In these first 12 chapters, we use the core parts of Django—the parts used in (almost) every website—to create the basics of our website. This includes interacting with a database, sending HTML to visitors, and accepting user input in a safe manner.

Part II, Django’s Contributed Library, examines the tools provided by Django that are helpful when building a website but that are not necessary to every site. Effectively, we will be adding features to our website to modernize the site and make it full-featured. From Chapter 13 through Chapter 23, we will see how to integrate CSS into our website, shorten our code through generic behavior, and add user authentication to our website.

Part III, Advanced Core Features, expands on Django’s basics, detailing how to improve their use. We see how to take full control of our site, shortening code, optimizing our site for speed, and expanding behavior. We then deploy our website to the Internet, hosting the website on Heroku’s managed cloud. Finally, in Chapter 30, we consider what we would have done differently in our project had we known at the beginning what we now know at the end of the book.

Conventions Used in This Book
This book is written with a bottom-to-top approach, meaning we start with a lower level of abstraction (more details) and gradually move up the abstraction ladder (shorter but more opaque code). If you would prefer to learn with a top-to-bottom approach, I recommend reading Chapter 12 after Chapter 1, and starting each chapter with the last section of the chapter, titled “Putting It All Together” throughout the book.

This book features quite a few asides (sometimes called admonitions) meant to help you understand Django or else to add tidbits of information to your programming toolkit.

Info  An aside with basic information that extends or adds to the current content.

Warning!  Gotchas, errors, and things to watch out for: these warnings are here to make your life easier by helping you avoid common mistakes.

Documentation  Links to documentation from Django, Python, and other resources, which enable you to continue to learn material on the subject at hand.

Code Repository  This book is heavily tied to the website found at https://django-unleashed.com, as is the project code found throughout the book and provided in full on github. Each example from the project has the git commit hash printed with it (and is a link to the digital version), allowing you to access each commit by adding https://dju.link/ before the commit hash (this may in turn be followed by a file path). Even so,
every so often a particular commit is worth noting, and these asides will point you toward the code in the repository.

**Ghosts of Django Past and Future**  The project in this book uses Django 1.8, the latest version, to create a website. However, it is not uncommon to find earlier versions of Django in the wild or at your new workplace. These asides aim to give you knowledge of changes between Django 1.4 and Django 1.8 so you can more easily navigate the various versions of Django if need be (that said, any new project should strive to use the latest version of Python and Django).
Acknowledgments

I have been blessed with an incredible family. I could not have done this without them. A huge thank you to Amber Gode and Anna Ossowski, both of whom read and reviewed large portions of this book and without whom this would be a very different product. Thanks to Wendell Smith, James Oakley, Dave Liechty, Jacinda Shelly, and Andrew Farrell for all of their hard work and feedback on both the code and the writing. Special thanks to Amy Bekkerman for always knowing the right question to ask. Thanks to Harry Percival for catching problems with the code. Thanks to Sasha Méndez for her feedback and particularly to Debra Williams Cauley of Pearson. She shared my enthusiasm for this project and enabled me to get this book going. Thank you to Sarah Abraham, Matt Kaemmerer, and Blake West, who were my very first guinea pigs (in the class that eventually gave rise to this book). Thanks to Paul Phillips for always grabbing a beer and listening to me complain about the sometimes frustrating, blinding work of coding and writing.

Finally, I want to acknowledge the Django and Python Communities. They are an amazing group of individuals, and I would not have written such an extensive book without their openness and support.
This page intentionally left blank
Andrew Pinkham is a software engineer who grew up in Paris and currently resides in Austin, Texas. Andrew runs a consulting business called JamBon Software, which specializes in web and mobile products and also offers Python and Django training. He prides himself on being an engineer who can communicate complex ideas in simple ways and is passionate about security and distributed systems. In his free time, Andrew writes fiction and swims. He is a 2009 graduate of Dartmouth College and can be found online at andrewsforge.com, or afrg.co for short.
We Want to Hear from You!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we’re doing right, what we could do better, what areas you’d like to see us publish in, and any other words of wisdom you’re willing to pass our way.

We welcome your comments. You can email or write to let us know what you did or didn’t like about this book—as well as what we can do to make our books better.

*Please note that we cannot help you with technical problems related to the topic of this book.*

When you write, please be sure to include this book’s title and author as well as your name and email address. We will carefully review your comments and share them with the author and editors who worked on the book.

Email: errata@informit.com

Mail:
  
  Sams Publishing  
  ATTN: Reader Feedback  
  330 Hudson Street  
  7th Floor  
  New York, New York 10013

Reader Services

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.
This page intentionally left blank
Starting a New Django Project: Building a Startup Categorizer with Blog

In This Chapter
- The difference between static and dynamic websites
- The difference between the front end and back end of websites
- The HTTP request/response cycle
- The nature of a framework and how it differs from a library
- What it means to be a Python web framework (e.g., Django)
- The outline of the project we will build in Parts I, II, and III

1.1 Introduction
We have a lot to do and a lot to learn, but instead of jumping right in, let’s take a moment to understand what we’re doing.

Part I is an example meant to demonstrate the core features of Django. Part I is intended to be read linearly. Jump between chapters at your own peril!

This first chapter is a general introduction to the modern world of building dynamic websites. We start by introducing web technologies and jargon before taking a look at Django. Our introduction to Django focuses on what Django is and appropriate ways to use it. We then outline the project we’ll build, scoping out the content for not only Part I but also Parts II and III. This overview gives us the opportunity to use Django to generate a basic project that we’ll use throughout the book.

Warning!
This book assumes knowledge of Python (but not web technologies)! While the appendix supplies a very short review of Python, this book will not teach you to code in Python.
Chapter 1  Starting a New Django Project

Info
This book is heavily tied to a git repository, which contains all of the project code and much of the example code found in this book:

https://github.com/jambonrose/DjangoUnleashed-1.8/

If you are reading the digital version of this book, the file paths and commit hashes in the project examples of this book are actually links that will take you directly to relevant commit on Github.

If you are reading a physical copy of this book, I have provided the dju.link shortlink domain. The link http://dju.link/9937ef66c0 will redirect you to the Github commit diff for the project, just as http://dju.link/9937ef66c0/helloworld/views.py will redirect you to the views.py file as it exists in the 9937ef66c0 hash.

Additional content may be found on the book’s website:

http://django-unleashed.com/

1.2 Website Basics

Before talking about how we build websites, it's important to understand what a website is and how it operates.

When we open our browser and enter a URL such as http://google.com, our computer uses HTTP (the scheme in the URL) to talk to the computer (or set of computers) found at the google.com domain. The goal of this computer is to give us information that we are asking for.

A website is a resource stored on a server. A server is simply a computer whose job is to provide a resource (a website in this case) or service and serve it to you. A website comprises one or more webpages. A webpage is a discrete entity that contains data. The core functionality of a website is to send these webpages to people who ask for them. To do this, we use a protocol (a means of communication) called Hyper Text Transfer Protocol (HTTP). Formally, a user's browser sends an HTTP request to a website. The website then sends an HTTP response containing a webpage. The process is illustrated in Figure 1.1.

Each webpage is uniquely identifiable, usually by using a Uniform Resource Locator (URL). A URL is a string with specific information, split according to the following (specified in RFC 3986):

scheme://network_location/path?query#fragments.

For example, Figure 1.2 shows the breakdown for a real URL.

The network location, or authority, is typically either an IP address (such as 127.0.0.1) or a domain name, as shown in Figure 1.2. The scheme tells the browser not only what to

---

1. https://dju.link/rfc3986
1.3 Understanding Modern Websites

HTTP is a stateless protocol: it doesn’t know who you are or where you’ve been. It knows only what you’ve just asked it for. In the early days of the Internet, each webpage on a site was a file, such as a text file or a PDF. Websites were **static**.

Today, many websites are **dynamic**. We now interact with websites: instead of just asking the server to send us a file, we write comments on videos, blog about the best web framework ever, and tweet cat pictures to our friends. To enable these activities, webpages must be generated (computed) for each user based on new and changing data. We’ve had to add a number of technologies on top of HTTP to determine state (such as sessions, which we’ll see in Chapter 19: Basic Authentication), and we now have entire languages and systems (like Django!) to make the dynamic generation of webpages as easy as possible. Our original HTTP loop now has an extra step between the request and response, as shown in Figure 1.3.

This dynamic generation of webpages is referred to as **back-end** programming, as opposed to **front-end** programming. Front-end programming involves creating the
behavior of the webpage once it has already been generated by the back end. We can think
of the combined experience in four steps:

1. A user’s browser issues a request for a page.
2. The server (or back end) generates a markup file (typically HTML) based on recorded
   information and information provided by the user; this file in turn points the user to
   download associated content such as JavaScript, which defines behavior, and
   Cascading Style Sheets (CSS), which define style such as color and fonts. The entire
   set of items defines the webpage.
3. The server responds to the user’s browser with this markup file (typically causing the
   browser to then ask for the other content such as CSS and JavaScript).
4. The user’s browser uses the information to display the webpage. The combination of
   HTML (content and structure), CSS (style), and JavaScript (behavior) provides the
   front-end part of the website.

While front-end programming certainly provides for a dynamic experience, the words
dynamic webpage typically refer to a webpage that is computed on the back end. It can be
difficult to distinguish the difference between front-end and back-end programming because
modern websites strive to blur the difference to create a more seamless user experience. In
particular, websites known as single-page applications blur this line to the point that
step 2 is seriously mangled. However, the distinction is still important, as the HTTP
protocol remains between the user and the server and the tools for front-end and back-end
programming are typically quite different.

Furthermore, this book does not cover front-end programming. We will see how to
serve static content such as CSS and JavaScript in Chapter 16: Serving Static Content
with Django, but we will not write a single line of either. This book is dedicated entirely to
back-end programming and generating dynamic webpages with Django.

1.4 Building Modern Websites: The Problems That Frameworks Solve and Their Caveats

Very few people program dynamic websites from scratch anymore (i.e., without relying on
other people’s code). It is a difficult, tedious process, and it is typically not a good use of
time. Instead, most developers rely on frameworks.
A framework is a large codebase, or collection of code, meant to provide universal, reusable behavior for a targeted project. For example, a mobile framework, such as those provided by Apple and Google for their mobile phones or smartphones, provides key functionality for building mobile apps. Consider the many touch actions on the iPhone: a user can tap his or her screen or hold, slide, turn with two fingers, and more. Developers do not need to worry about figuring out what touch action the user has performed: Apple’s framework handles that task for developers.

Using frameworks offers enormous advantages. The most obvious is the removal of tedious and repetitive tasks: if iPhone apps require specific behavior, then the framework will provide it. This saves time for developers not only because of the provided functionality, which allows developers to avoid coding entirely, but also because the code provided is tested by many other developers on a wide variety of projects. This widespread testing is particularly important when it comes to security—a group of developers working on a framework are more likely to get sensitive components right than is any single developer.

Frameworks are different from other external codebases, such as libraries, because they feature **inversion of control**. Understanding inversion of control is key to properly using frameworks. Without a framework, the developer controls the flow of a program: he or she creates behavior or pulls behavior into the code project by calling functions from a library or toolkit. By contrast, when using a framework, the developer adds or extends code in specific locations to customize the framework to the program’s requirements. The framework, which is essentially the base of the program, then calls those functions implemented by the developer. In this way, the framework, not the developer, dictates control flow. This is sometimes referred to as the Hollywood principle: “Don’t call us, we’ll call you.” We can easily demonstrate the difference in pseudocode (Example 1.1).

**Example 1.1: Python Code**

```python
# Using a Library
def my_function(*args):
    ...
    library.library_function(*args)
    ...

# Using a framework
def my_function(*args):
    ...

framework.run(my_function)
```

Inversion of control may seem counterintuitive or even impossible. How may the robot choose its behavior? Remember: the framework is built by other developers, and they are the ones who specify the behavior followed by the framework. As a developer using a framework, you are simply adding to or directing the behavior provided by other developers.

Using a framework has a few caveats. A framework may offer significant time savings, reusability, and security and may encourage a more maintainable and accessible codebase,
but only if the developer is knowledgeable about the framework. A developer cannot fill in all the gaps (by adding or extending code) expected by the framework until he or she understands where all the gaps are. Learning a framework can be tricky: because a framework is an interdependent system, using a part of the framework may require understanding another part of the system (we’ll see this in Chapter 6: Integrating Models, Templates, Views and URL Configurations to Create Links between Webpages), which requires knowledge and tools from the three chapters preceding it. For this reason, using a framework requires investing significant overhead in learning the framework. In fact, it will take all of Part I of this book to explain the core inner workings of Django and to gain a holistic understanding of the framework. But once there, we’ll be off to the races.

Despite this overhead, it is in your interest to use a framework and to spend the time to learn how to use it properly. Colloquially, developers are told, “Don’t fight the framework.”

1.5 Django: Python Web Framework

As outlined in Section 1.3, a website must always

1. Receive an HTTP request (the user asks for a webpage)
2. Process the request
3. Return the requested information as an HTTP response (the user sees the webpage)

Django is a free and open-source Python back-end web framework that removes the tedium of building websites by providing most of the required behavior. Django handles the majority of the HTTP request and response cycle (the rest is handled by the server Django runs on top of). Developers need only focus on processing the HTTP request, and Django provides tools to make even that easy.

All Django projects are organized in the same way, largely because of the framework’s inversion of control but also because it makes navigating existing Django projects much easier for developers who, for instance, maintain the code or step into a job mid-project.

Django’s project structure is most often described according to the Model-View-Controller (MVC) architecture because it makes the framework easier to learn. Originally, MVC was a very specific architecture, but it has become an umbrella term for libraries that are patterned after the following idea (illustrated in Figure 1.4):

- The Model controls the organization and storage of data and may also define data-specific behavior.
- The View controls how data is displayed and generates the output to be presented to the user.
- The Controller is the glue (or middleman) between the Model and View (and the User); the Controller will always determine what the user wants and return data to the user, but it may also optionally select the data to display from the Model or use the View to format the data.
Most often, literature will state that different pieces of Django map to different pieces of MVC. Specifically,

- **Django models** are an implementation of MVC Models (Chapter 3: Programming Django Models and Creating a SQLite Database).
- **Django templates** map to MVC Views (Chapter 4: Rapidly Producing Flexible HTML with Django Templates).
- **Django views and URL configuration** are the two pieces that act as the MVC Controller (Chapter 5: Creating Webpages with Controllers in Django).

**Warning!**

Django and MVC use the word *view* to mean different things.

- The View portion of MVC determines how data is displayed.
- In Django, a view refers to something that builds a webpage and is part of the implementation of MVC Controllers.

Django views and MVC Views are unrelated. Do not confuse them.

The truth is a little bit more complicated. Django projects aren’t truly MVC, especially if we abide by the original definition. We will discuss this topic in much more depth in Chapter 12: The Big Picture, once we have a better grasp of all of the moving pieces. For the moment, because it can help beginners organize the framework, we continue to use the (more modern and vague version of) MVC architecture to make sense of the framework.

If we combine our diagrams of the HTTP request/response loop and MVC architecture as in Figure 1.5, we get a much better picture of how Django works.

The Controller, the subject of Chapter 5, represents the heart of Django and is the only part of the MVC architecture that is necessary to generate a webpage. However, most
browsers expect data to be returned by the server in specific formats, such as XML, HTML, or HTML5. The View encapsulates the tools Django supplies for easily outputting such data and is the subject of Chapter 4. Finally, we typically need to use persistent data when generating content in the Controller. The Model section represents the tools for structuring and storing data and is the subject of Chapter 3.

You may have noticed that the popular format JSON is missing from the list of formats that the View section of Django outputs (XML, HTML, HTML5). Django doesn’t need to supply a tool for outputting JSON because Python, which Django is built in, provides a JSON serializer.

You’ll note that the Model section is connected to a database. The Model itself does not store data but instead provides tools for communicating with databases. We discuss the merits of databases in more depth in Chapter 3. For the moment, just note that Django provides the tools to communicate with several different databases, including SQLite, MySQL, PostgreSQL, and Oracle, which is yet another huge time-saver for us.

Django provides many more tools to make building websites easy. For instance, database schema migrations (Chapter 3 and Chapter 10: Revisiting Migrations), which help with managing models, and an authentication system (Chapter 19 and Chapter 22: Overriding Django’s Authentication with a Custom User) are built in. What’s more, Django is Python code, allowing developers to use any standard or third-party Python library. Python libraries afford developers an enormous amount of power and flexibility.
Django prides itself on being the “web framework for perfectionists with deadlines.” Django provides the functionality needed for every website. The framework also comes with tools to make common website features easy to implement. This “batteries included” approach is why tens of thousands of developers use Django. Released into the wild in 2005, Django powers many websites, including Instagram, Pinterest, Disqus, and even The Onion. The core team of Django developers rigorously and regularly test Django, making it both fast and safe.

Django follows the Don’t Repeat Yourself (DRY) principle. You will never need to repeat your code if you don’t want to (of course, Django won’t stop you if you do). Additionally, Django adheres to the Python philosophy that explicit is better than implicit. Django will never assume what you want and will never hide anything from you. If there is a problem, Django will tell you.

As mentioned in Section 1.3, despite all of the things Django will do for you, it will not build or help build front-end behavior for you (this is the purview of JavaScript apps and the browser). Django is a back-end framework, only one half of the equation for building a modern website. It allows you to dynamically create HTML for the front end (Chapter 4) and to intelligently provide the content necessary for a modern front end (Chapter 16), but it does not provide the tools to build dynamic browser behavior for the user. However, before you toss this book in a corner and walk away from Django forever, note that a back end is necessary before a front end can exist. A back end may be only half of the equation, but it is the first half: without the request/response loop, there is no website.

1.6 Defining the Project in Part I

The purpose of *Django Unleashed* is to teach Django by example. The goal of Part I of this book is to teach you the core fundamentals of Django, the parts of the system required by every website, and how MVC (mostly) applies to that system. To accomplish these tasks, we begin building a website that is self-contained to Django: we purposefully avoid any external libraries built for Django in order to better focus on the framework itself. At each step of the building process, you are introduced to a new Django feature, providing insight into the framework. By the end of Part I, these insights will allow you to see exactly how Django operates and adheres to MVC. Note that the book builds on this project all the way through Part III. Even then, the goal is not to build a production-quality website but rather to teach you via example. We nonetheless discuss how to begin and build a production website in Chapter 30.

1.6.1 Selecting Django and Python Versions

Django 1.8 is the latest and greatest Django version and is what every new project should use. Although this book includes informative notes about older versions, please do not use deprecated versions for new projects because these versions do not receive security updates. Django 1.8 supports Python 2.7 and Python 3.2+ (Python 3.2, Python 3.3, and Python 3.4). When starting a new project, developers are left with the choice of which Python version to use for their project. The choice, unfortunately, is not as simple as picking the latest version.
Python 3 is the future, as Python 2.7 is officially the last Python 2 version. For a website to work for as long as possible, it becomes desirable to create Django websites in Python 3. However, Python 2 is still commonly used, as Django has only officially supported Python 3 since version 1.6, released in November 2013. What’s more, enterprise Linux systems still ship with Python 2 as the default, and tools and libraries built for Django may still require Python 2 (as our site in Parts I, II, III is self-contained to Django, we do not need to worry about this decision yet, but we return to the issue in Chapter 30).

When creating reusable tools for a Django project, the gold standard is thus to write code that works in both Python 3 and Python 2. The easiest way to do this is to write code intended for Python 3 and then make it backward compatible with Python 2.7.

Our project is a simple website not aimed at being reused. In light of this and the many guides written about writing Python code that runs in both 2 and 3, our project will be built to run only in Python 3. Specifically, we use Python 3.4 (there is no technological reason to choose 3.2 or 3.3 over 3.4). This will further allow us to focus on Django itself and not get distracted by compatibility issues.

### 1.6.2 Project Specifications

Website tutorials have gone through phases. Tutorials started by teaching developers how to build blogs. Some disparaged these yet-another-blog tutorials as being passé. Writers switched first to building forums, then polls, and finally to-do lists.

In the real world, if you needed any of these applications, you would download an existing project such as WordPress or sign up for a service such as Medium. Rather than weeks of development, you would have a website by the end of an afternoon. It might not be as you envisioned your perfect site, but it would be good enough.

One of Django’s major strengths is its precision. Django allows for the rapid creation of unusual websites that work exactly as the developer desires. It is in your interest for this book to build a website that is not available on the Internet already. The difficulty with building an unusual website is that the material tends to be less accessible.

Given the approachable nature of a blog, we will build a blog with special features. A blog is a list of articles, or blog posts, published on a single site and organized by date. Blog authors may choose to write about anything in each post, but they usually stick to a general theme throughout the entire blog. Our blog focuses on news relating to technology startup businesses. The goal is to help publicize startups to blog readers.

The problem with most blogs is that their topics are not well organized. Blogging platforms typically label blog posts with tags, leading writers to create tags for each item they blog about. A blog about startups would likely have a tag for each startup written about. We use Django to improve our blog’s topic organization.

In our website, we expand blog functionality by codifying the creation of startups. Each startup will be its own object, not a tag. The advantage of making startups their own objects is that it allows us to add special information about them. We can now display information related to the business. We can list a description and a date, and we can even link to external articles written about the startup. These capabilities would not be possible if the startup were simply a tag.
Furthermore, we may organize startups with the same tags we use to label the blog posts. For example, we may label Startup A with the Mobile and Video Games tags. We could then tag Startup B with Mobile and Enterprise. These categories make organizing data simple but flexible. If we browse to the Mobile tag, the website uses that tag to list both Startup A and Startup B as well as any blog posts with the tag. For our website, we also enable blog posts to be directly connected to startup objects. Blog posts will thus exist for news about the site itself or to announce news about startups in our system. Our website makes startups far more discoverable than a regular blog website would.

In Part I, we focus on the most basic features. We create the blog, startup, and tagging system in Django. The goal is to make Django’s core, features necessary to every website, as evident as possible.

In Part II, we allow authenticated users to log in. The public will be able to read any of the content of the website. Authenticated users will be able to submit articles, startups, and tags. These content suggestions will be reviewable by you, the site administrator.

In Part III, we allow for tag inheritance. If we write a blog post about Startup A, the tags labeling the startup will now also label the blog post.

It benefits us to list the webpages we will build in Part I:

1. A page to list tags
2. A page to list startups
3. A page to list blog posts
4. A page for each tag
5. A page for each startup
6. A page for each blog post (which also lists news articles)
7. A page to add a new tag
8. A page to add a new startup
9. A page to add a new blog post
10. A page to add and connect news articles to blog posts

1.7 Creating a New Django Project and Django Apps

In the following section, we create a new Django project in preparation for the website laid out in the last section. We then create Django apps, which are like small libraries within our project (we go over them in detail when we create them). By the end, we will be ready to start coding our website.

We do not cover how to install Django here. The official website has an excellent and updated guide\(^2\) to do this. Just in case, however, I have supplied my own writing on the subject in Appendix G.

\(^2\) https://djui.link/18/install
1.7.1 Generating the Project Structure

Inversion of control means that Django already provides most of the code required to run a website. Developers are expected to supplement or extend the existing code so that the framework may then call this code; by placing code in key places, developers instruct the framework how to behave according to the developers’ desires. Think of it as creating a building: even though many of the tools and contractors are supplied, the developer must still give these contractors orders, and the process requires a very specific scaffolding. Originally, building the scaffolding was a real pain, as developers had to manually account for framework conventions. Luckily, modern frameworks supply tools that generate the correct scaffolding for us. Once this scaffolding is in place, we can instruct the various contractors to behave in specific ways.

With Django correctly installed (please see Appendix G), developers have access to the `django-admin` command-line tool. This command, an alias to the `django-admin.py` script, provides subcommands to automate Django behavior.

Ghosts of Django Past

If you are using a version of Django prior to 1.7, then the alias `django-admin` will be unavailable. You will instead have to invoke the actual script, `django-admin.py`.

Our immediate interest with `django-admin` is the `startproject` subcommand, which automatically generates correct project scaffolding with many, but not all, of the expected Django conventions. To create a project named *suorganizer* (start up organizer), you can invoke the command shown in Example 1.2.

**Example 1.2: Shell Code**

```
$ django-admin startproject suorganizer
```

Inside the new folder by the name of our new project, you will find the folder structure shown in Example 1.3.

**Example 1.3: Shell Code**

```
$ tree .
.
  manage.py
  suorganizer
   __init__.py
  settings.py
  urls.py
  wsgi.py

1 directory, 5 files
```
Please note the existence of two directories titled suorganizer. To avoid confusion between the two directories, I distinguish the top one as root, or /, throughout the rest of the book. As such, instead of writing suorganizer/manage.py, I will refer to that file by writing /manage.py. Importantly, this means /suorganizer/settings.py refers to /suorganizer/suorganizer/settings.py. What’s more, all commands executed from the command line will henceforth be run from the root project directory, shown in Example 1.4.

Example 1.4: Shell Code

$ ls
manage.py suorganizer

Let’s take a look at what each file or directory does.

- / houses the entire Django project.
- /manage.py is a script much like django-admin.py: it provides utility functions. We will use it in a moment. Note that it is possible to extend manage.py to perform customized tasks, as we will see in Part II.
- /suorganizer/ contains project-wide settings and configuration files.
- /suorganizer/__init__.py is a Python convention: it tells Python to treat the contents of this directory (/suorganizer/) as a package.
- /suorganizer/settings.py contains all of your site settings, including but not limited to
  - timezone
  - database configuration
  - key for cryptographic hashing
  - locations of various files (templates, media, static files, etc)
- /suorganizer/urls.py contains a list of valid URLs for the site, which tells your site how to handle each one. We will see these in detail in Chapter 5.
- /suorganizer/wsgi.py stands for Web Server Gateway Interface and contains Django’s development server, which we see next.

### 1.7.2 Checking Our Installation by Invoking Django’s runserver via manage.py

While Django has only created a skeleton project, it has created a working skeleton project, which we can view using Django’s testing server (the one referenced in /suorganizer/wsgi.py). Django’s /manage.py script, provided to every project, allows us to quickly get up to speed.
Django requires a database before it can run. We can create a database with the (somewhat cryptic) command `migrate` (Example 1.5).

**Example 1.5: Shell Code**

```
$ ./manage.py migrate
```

You should be greeted with the output (or similar output) shown in Example 1.6.

**Example 1.6: Shell Code**

```
Operations to perform:
 Synchronize unmigrated apps: staticfiles, messages
 Apply all migrations: contenttypes, auth, admin, sessions
Synchronizing apps without migrations:
 Creating tables...
 Running deferred SQL...
 Installing custom SQL...
Running migrations:
Rendering model states... DONE
Applying contenttypes.0001_initial... OK
Applying auth.0001_initial... OK
Applying admin.0001_initial... OK
Applying contenttypes.0002_remove_content_type_name... OK
Applying auth.0002_alter_permission_name_max_length... OK
Applying auth.0003_alter_user_email_max_length... OK
Applying auth.0004_alter_user_username_opts... OK
Applying auth.0005_alter_user_last_login_null... OK
Applying auth.0006_require_contenttypes_0002... OK
Applying sessions.0001_initial... OK
```

We’ll see exactly what’s going on here starting in Chapter 3 and in detail in Chapter 10. For the moment, let’s just get the server running by invoking the `runserver` command shown in Example 1.7.

**Example 1.7: Shell Code**

```
$ ./manage.py runserver
Performing system checks...
System check identified no issues (0 silenced).
May 2, 2015 - 16:15:59
Django version 1.8.1, using settings 'suorganizer.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```
Ghosts of Django Past

In versions prior to Django 1.7, the command above will not work, as manage.py does not have execute permissions. Run chmod +x manage.py to give manage.py the needed permission, or else execute it by invoking it through Python.

For example: python manage.py runserver

If you navigate your browser to http://127.0.0.1:8000/, you should be greeted with the screen printed in Figure 1.6.

Django is running a test server on our new project. As the project has nothing in it, Django informs us we need to create an app using /manage.py.

To quit the server, type Control-C in the terminal.

1.7.3 Creating New Django Apps with manage.py

In Django nomenclature, a project is made of any number of apps. More expressly, a project is a website, while an app is a feature, a piece of website functionality. An app may be a blog, comments, or even just a contact form. All of these are encapsulated by a project, however, which is the site in its totality. An app may also be thought of as a library within the project. From Python’s perspective, an app is simply a package (Python files can be modules, and a directory of modules is a package).

We have two features in our site: (1) a structured organization of startups according to tags and (2) a blog. We will create an app for each feature.

As with a project, Django supplies a way to easily create the scaffolding necessary to build an app. This time, we invoke /manage.py to do the work for us, although we could just as easily have used django-admin. Let’s start with the central focus of our site, our startup organizer, and create an app called organizer, as shown in Example 1.8.

It worked!

Congratulations on your first Django-powered page.

Of course, you haven’t actually done any work yet. Next, start your first app by running python manage.py startapp [appname].

You’re seeing this message because you have DEBUG = True in your Django settings file and you haven’t configured any URLs. Get to work!

Figure 1.6: Runserver Congratulations Screenshot
Example 1.8: Shell Code

$ ./manage.py startapp organizer

The directory structure of the project should now be as shown in Example 1.9.

Example 1.9: Shell Code

$ tree .
.
  manage.py
  organizer
    __init__.py
    admin.py
    migrations
      __init__.py
    models.py
    tests.py
    views.py
  suorganizer
    __init__.py
    settings.py
    urls.py
    wsgi.py

3 directories, 11 files

Ghosts of Django Past

Prior to version 1.7, Django did not supply a migration system, and thus the migrations directory in projects older than that version will not appear, or will actually belong to a tool called South. Be careful about this when using projects built in early Django versions!

Info

You will likely also find files ending in .pyc. These are compiled Python files and can be safely ignored. However, if you find them as distracting as I do, you can use the following shell command to remove them: find . -name '*.pyc' -delete. Python will re-create them the next time you run your site.

Let’s take a look at the new items.

- /organizer/ contains all the files related to our new organizer app. Any file necessary to running our blog will be in this directory.
- /organizer/__init__.py is a Python convention: just as for /suorganizer/__init__.py, this file tells Python to treat the contents of this directory (/organizer/) as a package.
- /organizer/admin.py contains the configuration necessary to connect our app to the Admin library supplied by Django. While Admin is a major Django feature, it is
not part of Django’s core functionality, and we will wait until Part II to examine it, along with the rest of the Django Contributed Library (apps included with Django’s default install). If you are very impatient, you should be able to jump to Chapter 23: The Admin Library as soon as you’ve finished reading Chapter 5.

- `/organizer/migrations/` is a directory that contains data pertaining to the database tables for our app. It enables Django to keep track of any structural changes the developer makes to the database as the project changes, allowing for multiple developers to easily change the database in unison. We will see basic use of this database table in Chapter 3 and revisit the topic in Chapter 10.
- `/organizer/migrations/__init__.py` marks the migration directory as a Python package.
- `/organizer/models.py` tells Django how to organize data for this app. We do see how this is done in the next chapter.
- `/organizer/tests.py` contains functions to unit test our app. Testing is a book unto itself (written by Harry Percival), and we do not cover that material.
- `/organizer/views.py` contains all of the functions that Django will use to process data and to select data for display. We make use of views starting in Chapter 2 but won’t fully understand them until Chapter 5.

Django encapsulates data and behavior by app. The files above are where Django will look for data structure, website behavior, and even testing. This may not make sense yet, but it means that when building a site with Django, it is important to consider how behavior is organized across apps. Planning how your apps interact and which apps you need, as we did earlier in this chapter, is a crucial step to building a Django site.

We can create our `blog` app in exactly the same way as the `organizer` app, as shown in Example 1.10.

**Example 1.10: Shell Code**

```bash
$ ./manage.py startapp blog
```

Note that the directory structure and all the files generated are exactly the same as for our `organizer` app.

### 1.7.4 Connecting Our New Django Apps to Our Django Project in `settings.py`

Consider for a moment the difference between `/organizer/` (or `/blog/`) and `/suorganizer/`. Both encapsulate data, the former for our `organizer` (or `blog`) app and the second for our project-wide settings, a phrase that should mean more now that we know the difference between an app and a project (reminder: a project is made up of one or more apps).

We must now connect our new apps to our project; we must inform our project of the existence of `organizer` and `blog`. On line 33 of `/suorganizer/settings.py`, you will find a list of items titled `INSTALLED_APPS`. Currently enabled in our project are a list of...
Django contributed apps (you can tell because these items all start with `django.contrib`), some of which we examine in Part II. We append the list with our new apps, as shown in Example 1.11.

**Example 1.11: Project Code**

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

**Info**

While the order of `INSTALLED_APPS` typically does not matter, there are instances in which apps listed prior to others will be given precedence. We see an instance of this in Chapter 24.

Let's run our test server again (Example 1.12).

**Example 1.12: Shell Code**

```
$ ./manage.py runserver 7777
Performing system checks...
System check identified no issues (0 silenced).
February 10, 2015 - 19:09:25
Django version 1.8.3, using settings 'suorganizer.settings'
Starting development server at http://127.0.0.1:7777/
Quit the server with CONTROL-C.
```

**Info**

Note that this time, I've run the server with an extra parameter that specifies which port I want to run on. Instead of the default port 8000, the server may now be accessed on port 7777 via URL `http://127.0.0.1:7777/`. By convention, `http://127.0.0.1` will always point to your own computer. Any port may be specified, but a port number below 1024 may require superuser privileges (which are typically attained via `sudo`). The ability to specify a port is useful when the 8000 port...
is already taken. For instance, you may be testing another Django website at the same
time or have another program that defaults to 8000. To make the server publicly
available on port 80 (the standard port for HTTP; a very dangerous thing to do), you
could use the command `sudo ./manage.py runserver 0.0.0.0:80`.

Navigating to the page in your browser, you should be greeted by exactly the same page
in your browser, telling you once again to

1. Create a new App
2. Configure our site URLs

We have successfully done item 1 and will demonstrate item 2 in our Hello World
example in the next chapter.

We will return to our main project in Chapter 3, where we organize our data and create
a database. In Chapter 4, we create templates to display data. In Chapter 5, we build our
URL configuration (expanding on item 2 above) and the rest of the MVC Controller.
These activities will effectively reveal how Model-View-Controller theory maps to Django.

## 1.8 Putting It All Together

The chapter outlined the project to be built in Parts I, II, and III of the book and
introduced Django.

Django is a Python web framework based on MVC architecture, which signifies that
Django removes the tedium of building websites by supplying a universal, reusable codebase.
This approach saves developers time in the long run but creates an overhead cost of having
to learn the interdependent system. Like any framework, Django works on the principle of
inversion of control, sometimes called the Hollywood principle ("Don’t call us, we’ll call
you"), which explains why we write code in locations dictated by Django convention.
Specifically, in keeping with MVC architecture, we know that we need only worry about
the Models, Views, and Controllers and that Django will glue them together and handle
everything else.

In this chapter, we used `django-admin` to generate the project scaffolding necessary for
Django. This scaffolding allows us to add code in specific locations, according to inversion
of control.

We not only generated a Django project but also created the apps necessary for any
project: a project is a website, whereas an app is a feature, a piece of website functionality.
The site we’ve set out to build is a startup categorization system paired with a blog. Given
the two features, we created two apps using Django’s `manage.py` tool. We then used this
tool to run a test server, checking our work. The test server informed us that, now that we
have our apps created and connected to our project via `settings.py`, we should
configure our site URLs. We take a quick look at this in the next chapter, but we wait until
Chapter 5 before we really get there.

This book seeks to teach Django by example. Part I teaches Django’s core, or the pieces
of the framework that are typically required for every project. Django organizes project data
according to MVC theory. Chapters 3, 4, and 5 each demonstrate a core Django feature,
each an aspect of MVC. In Chapter 3, we organize our data and create a database.
In Chapter 4, we create the display output for our data. In Chapter 5, we connect our data to our display, creating webpages by programming Django views, pointed to by URL configurations.

Before we jump into MVC, however, Chapter 2: Hello World: Building a Basic Webpage in Django illustrates a basic Django site, which sheds light on the power of MVC.
5

Creating Webpages with Controllers in Django: Views and URL Configurations

In This Chapter
- Build webpages with views and URL patterns
- Learn the purpose behind views and URL configurations
- Build views using both Python functions and objects
- Learn the differences between function and class-based views
- Make programming quicker with Django shortcuts
- Connect URL configurations to encapsulate behavior by app
- Preview webpage redirection

5.1 Introduction

In Chapter 2: Hello World: Building a Basic Webpage in Django, we saw that the Controller is the only part of Django actually required to make a webpage (the relevant diagram is reprinted in Figure 5.1). However, we immediately ran into problems: we had no way to easily fetch and format data. Because the main function of websites revolves around data, the Controller is often described as the glue between Model and View despite the Controller’s independence.

In this chapter, we return to the Controller, seen earlier in Chapter 2 and Chapter 4: Rapidly Producing Flexible HTML with Django Templates. We first re-examine how the two parts of the Controller, URL configurations and views, interact. We then use the cumulative knowledge we have gained to build dynamic webpages.

The Controller is central to Django and comes with a number of options. Once we have the basics, we look at how to handle problems that occur in views. We then look at ways to more rapidly code views (at the cost of developer control). Coding views enable us to very quickly build all the webpages for our site.
Before we finish the chapter, we also examine two special methods for creating Controllers, which become important later in the book. This chapter assumes knowledge of HTTP and regular expressions. Primers on both are provided in Appendix A and Appendix E, respectively.

5.2 The Purpose of Views and URL Configurations

A webpage consists of (1) the data contained in the webpage and (2) the URL (location) of the webpage. Django follows this abstraction by splitting the Controller into two parts. Django views give Django the data of the webpage. The URL associated with each view is listed in the URL configuration.

**Warning!**
For many beginners, the name of the Controller causes confusion: Django views are unrelated to MVC architecture’s View. Django views are one half of the Controller. Django templates map to MVC’s Views. To differentiate between the two, I capitalize View when referring to MVC and use lowercase view when referring to Django.

In the rest of this section, we expand on the nature and purpose of the URL configuration and views. To make the material more tangible, we then step through what happens when Django receives a request, detailing the actions the Controller takes.
5.2 The Purpose of Views and URL Configurations

5.2.1 Django URL Configurations

As discussed in Chapter 1, Section 1.2, webpages were originally quite basic. The webpage’s data were contained in a flat file (a text file, an HTML file, or a PDF file, for instance). The URL was literally the location of the file on the server. If a user directed his or her browser to \texttt{http://awebsite.com/project1/important.pdf}, the \texttt{awebsite.com} server would go to the \texttt{project1} directory and fetch the \texttt{important.pdf} file to give to the user’s browser.

Because modern web frameworks generate webpages dynamically, URLs have ceased to be the actual path to the data. A URL is now an abstraction, and it represents the logical path to data. For instance, the path \texttt{/startup/jambon-software} obviously requests information about the JamBon Software startup, whereas the path \texttt{/blog/2013/1/django-training/} is clearly a request for a blog post about Django classes published in January 2013.

The name \textit{Uniform Resource Locator} is thus not quite right anymore, as we are not actually requesting the location of the data. Instead, we are simply identifying it. Appropriately, URLs are a direct subset of Uniform Resource Identifiers (URIs), as illustrated in Figure 5.2.

While there is some confusion surrounding the difference between URLs and URIs, RFC 3986\(^1\) is quite clear on the topic (effectively superseding RFC 3305)\(^2\):

A URI can be further classified as a locator, a name, or both. The term “Uniform Resource Locator” (URL) refers to the subset of URIs that, in addition to identifying a resource, provides a means of locating the resource by describing its primary access mechanism (e.g., its network “location”).

---

\(^1\) https://dju.link/rfc3986  
\(^2\) https://dju.link/rfc3305
Every URL is thus a URI. However, a URL must specify a scheme to access the data, such as http or https, while a URI does not have to. According to this definition, the string /blog/2013/1/django-training/ is a URI, but the string http://site.django-unleashed.com/blog/2013/1/django-training/ is a URL despite the fact that the URL path is not an actual location. For this reason, Django continues to refer to URLs instead of URIs.

Because of the Hollywood principle (inversion of control), the URL configuration acts as a way to direct both users and Django to data. The URL configuration connects URLs to views: Django uses the URL configuration to find views. Django does not know the existence of any view without the URL configuration.

The URL configuration is a list of URL patterns. The URL pattern represents the two parts of a webpage: it maps a URI (the route/location/identifier) to a view (the data). Formally, the URI is a regular expression pattern, whereas the view is a Python callable. A URL configuration can also point to another URL configuration instead of a view, as we discuss in more depth in Section 5.7.1.

In Figure 5.3, each arrow is a URL pattern. Multiple URIs may point to a single view, but a single URI may not be defined more than once. The regular expression pattern in each URL pattern is how Django performs its matching. When Django receives an HTTP request, it tries to match the URL of the request to each and every regular expression pattern in each and every URL pattern. Upon finding a match, Django calls the view that the regular expression pattern maps to. Django uses the first match, meaning that the order of the list of URL patterns matters if there are several potential matches. If Django does not find a match, it returns an HTTP 404 error.

In the example provided by Figure 5.3, if a user requested the URI /startup/, perhaps in a URL such as http://site.django-unleashed.com/startup/, then Django

![Figure 5.3: URL Configuration](image-url)
would call the `startup_list()` function view. Django automatically strips the root slash of the URL path (to Django, `/startup/` becomes `startup/`).

We first coded a URL pattern in Chapter 2 and then again in Chapter 4. This last one, shown in Example 5.1, should still exist in `suorganizer/urls.py`.

```python
Example 5.1: Project Code
suorganizer/urls.py in 95b20c151b

23url(r'^$\$', homepage),
```

Requesting the root path of our website causes Django to call `homepage()`, coded in `organizer/views.py`. We walk through exactly how Django does this shortly.

**Info**

A word of caution: there are *URL patterns*, and there are *regular expression patterns*. Simply referring to a pattern is ambiguous and should be avoided.

### 5.2.2 Django Views

The view is where webpage data is generated. The developer uses the view to interact with the database, load and render the template, and perform any other logic necessary to displaying a webpage.

A Django view is any Python callable (function, class, or object) that meets the following two requirements:

- Accepts an `HttpRequest` object as argument
- Returns an `HttpResponse` object

An `HttpRequest` object contains all of the information about the page requested, any data the user is passing to the website, and any data the browser is sending about the user. The `HttpResponse` returns an HTTP code (please see Appendix A for information about HTTP codes) as well as any data the developer chooses to return to the user.

Because the nature of a view depends solely on its input and output, any Python callable can be a view. Typically, however, you will be using either functions or Django’s supplied classes to create views. For the moment, we build views using functions and wait until the end of the chapter to look at Django’s class-based views.

Developers often refer to Django views as **view functions**. This is rather confusing, as views are not limited to being functions (this was not the case historically, which is where the vocabulary originates). In this book, I refer to any callable that builds a webpage as a **view**. Any view that is built using a function is called a **function view**, and any view that is an object is called a **class-based view** (following the documentation’s nomenclature).

We currently have a function view coded in `organizer/views.py`, shown in Example 5.2.
We can see how the function in Example 5.2 adheres to view requirements: it accepts an HttpRequest object as the request argument and returns an HttpResponse object with the output of a rendered template. It is also clearly dynamic, generating content based on data in the database.

In this chapter, we focus on using the database to generate dynamic pages. In Chapter 9: Controlling Forms in Views, we generate dynamic pages based on not only the database but also the contents of the HttpRequest object. In Chapter 15: Creating Webpages with Django Flatpages, we also discuss the ability to make static/flat pages with views.

### 5.3 Step-by-Step Examination of Django’s Use of Views and URL Configurations

Nothing clarifies programming quite like walking through each step the code takes. Let’s find out what happens when we run our web server and navigate to http://127.0.0.1:8000/.

Before we can go to the webpage, we have to start Django. We do so with Example 5.3.

---

**Example 5.3: Shell Code**

```
$ ./manage.py runserver
```

Django loads the settings in suorganizer/settings.py, configuring itself. It then loads all of the URL patterns in the URL configuration into memory, which allows Django to match URLs quickly. Once set up, we can type http://127.0.0.1:8000/ into our browser.

Our browser begins by finding the server with the network location 127.0.0.1. That’s easy: that IP address always refers to the machine you’re using. Once it knows that, it looks at the scheme and path of the URL and sends an HTTP request for the path / to itself on port 8000. Django receives this request.

---

**Info**

If we had requested just http://127.0.0.1:8000 (without the last slash), the browser would still request the path /. It is implicit in this case.
Django first translates the actual HTTP request (raw data) into an `HttpRequest` object (Python). Having this object in Python makes it easy for us and the framework to manipulate any information the browser is passing our site, as we shall discover in Chapter 9. Django takes the path in the `HttpRequest` object—currently `/`—and strips it of the first `/`. In this case, we are left with the empty string. Our new path is the empty string `"`.

Django’s next goal is to select a URL pattern. Django has the list of URL patterns in the URL configuration it loaded into memory when it first started up. Each URL pattern consists of at least two things: a regular expression pattern and a view. To select a URL pattern, Django tries to match the requested path—the empty string in this case—to each regular expression pattern of each URL pattern. Given our URL configuration, Django currently has only two options, shown in Example 5.4.

### Example 5.4: Project Code

```
suorganizer/urls.py in 95b20c151b

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

    from organizer.views import homepage

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

Each call to `url()` in Example 5.4 is a URL pattern. Django tries to match the empty string, derived from the URL path, to each of the regular expression patterns in the URL patterns above. The empty string very clearly does not match the text `admin/`. However, Django will select the second URL pattern because the regular expression `r'^$'` matches the empty string:

- The `r` informs Python the string is raw, meaning it does not escape any of the characters in the string.
- The `^` matches the beginning of a string.
- The `$` matches the end of a string.

With the URL pattern `url(r'^$'`, `homepage`) selected, Django calls the Python function the URL pattern points to. In this case, the URL pattern points to the `homepage()` Python function, imported via the call `from organizer.views import homepage` on line 19. When Django calls the view, it passes the `HttpRequest` object to the view.

We coded the view such that it loads tag data from the database, loads the tag list template, and renders the template with the `Tag` object data. We then pass this output to an `HttpResponse` object and return it to Django. Django translates this object into a real HTTP response and sends it back to our browser. Our browser then displays the webpage to us.
To clarify, the regular expression `r'^a$'` would match a request to `http://127.0.0.1:8000/a`. If we were to change the URL pattern from `url(r'^$', homepage)` to `url(r'^home/$', homepage)`, we would now need to navigate to `http://127.0.0.1:8000/home/` to run the `homepage()` function and display a list of tags.

Inversion of control should be apparent. We are not controlling Django. It translates HTTP requests and responses for us and handles the entire URL matching process. We are simply providing it with the data to use in these matches and telling it what to use to build the webpage (the view). And even then, we are relying heavily on the tools Django provides.

If we were to ask Django for a webpage that did not exist, such as `http://127.0.0.1:8000/nonexistent/`, Django would try to match `nonexistent/` to the regular expression patterns in our URL configuration. When it did not find one, it would error. In production, Django would send back an HTTP 404 response. However, because we have `DEBUG=TRUE` in our `suorganizer/settings.py` file, Django instead tries to warn us of the problem and shows us a list of valid URL paths.

### 5.4 Building Tag Detail Webpage

To reinforce what we already know and expand our knowledge of URL patterns, we now create a second webpage. Our webpage will display the information for a single Tag object. We call our function view `tag_detail()`. Let’s begin by adding a URL pattern.

In Chapter 3, we specifically added `SlugField` to our `Tag` model to allow for the simple creation of unique URLs. We intend to use it now for our URL pattern. We want the request for `/tag/django/` to show the webpage for the `django` Tag and the request for `/tag/web/` to show the webpage for the `web` Tag.

This is the first gap in our knowledge. How can we get a single URL pattern to recognize both `/tag/django/` and `/tag/web/`? The second gap in our knowledge is that we have no easy way to use the information in the URL pattern. Once we’ve isolated `django` and `web`, how can we pass this information to the view so that it may request the data from the database?

To make the problem more concrete, let’s start with the `tag_detail()` view.

#### 5.4.1 Coding the `tag_detail()` Function View

Open `/organizer/views.py` and program the bare minimum functionality of a view (accept an `HttpRequest` object, return an `HttpResponse` object), as shown in Example 5.5.

#### Example 5.5: Project Code

```
organizer/views.py in f0d1985791

16  def tag_detail(request):
17       return HttpResponse()
```
Our first task is to select the data for the Tag object that the user has selected. For the moment, we will assume that we have somehow been passed the unique slug value of the Tag as the variable `slug`, and we use it in our code (but Python will yell at you if you try to run this). We use the `get()` method of our Tag model manager, which returns a single object. We want our search for the `slug` field to be case insensitive, so we use the `iexact` field lookup scheme. Our lookup is thus `Tag.objects.get(slug__iexact=slug)`, as shown in Example 5.6.

**Example 5.6: Project Code**

```python
organizer/views.py in ba4f692e00

16   def tag_detail(request):
17     # slug = ?
18     tag = Tag.objects.get(slug__iexact=slug)
19     return HttpResponse()
```

We may now load the template we wish to render, `organizer/tag_detail.html`. When we wrote the template, we wrote it to use a variable named `tag`. We thus create a `Context` object to pass the value of our Python variable named `tag` to our template variable `tag`. Recall that the syntax is `Context({'template_variable_name': Python_variable_name})`. We thus extend our view code as shown in Example 5.7.

**Example 5.7: Project Code**

```python
organizer/views.py in 2fdb78366f

16   def tag_detail(request):
17     # slug = ?
18     tag = Tag.objects.get(slug__iexact=slug)
19     template = loader.get_template('organizer/tag_detail.html')
20     context = Context({'tag': tag})
21     return HttpResponse(template.render(context))
```

We have what would be a fully working function view if not for the problem we are now forced to confront: the `slug` variable is never set. The value of the slug will be in the URL path. If Django receives the request for `/tag/django/`, we want the value of our `slug` variable to be set to `'django'`. Django provides two ways to get it.

The first way is terrible and inadvisable: we can parse the URL path ourselves. The `request` variable, an `HttpRequest` object, contains all the information provided by the user and Django, and we could access `request.path_info` to get the full path. In our example above, `request.path_info` would return `/tag/django/`. However, to get the slug from our URL path, we would need to parse the value of `request.path_info`, and doing so in each and every view would be tedious and repetitive, in direct violation of the Don’t Repeat Yourself (DRY) principle.
The second method, the recommended and easy solution, is to get Django to send it to us via the URL configuration, as we shall discover in the next section. To accommodate this solution, we simply add `slug` as a parameter to the function view.

Our final view is shown in Example 5.8.

Example 5.8: Project Code

organizer/views.py in 84eb438c96

```python
16   def tag_detail(request, slug):
17       tag = Tag.objects.get(slug__iexact=slug)
18       template = loader.get_template('organizer/tag_detail.html')
19       context = Context({'tag': tag})
20       return HttpResponse(template.render(context))
```

5.4.2 Adding a URL Pattern for `tag_detail`

With our `tag_detail()` function view fully programmed, we now need to point Django to it by adding a URL pattern to the URL configuration. The pattern will be in the form of `url(<regular_expression>, tag_detail)`, where the value of `<regular_expression>` is currently unknown. In this section, we need to solve two problems:

1. We need to build a regular expression that allows for multiple inputs. For example, `/tag/django/` and `/tag/web/` must both be valid URL paths.
2. We must pass the value of the slug in the URL path to the detail view.

The answer to both of these problems is to use regular expressions groups. To solve the first case, we first begin by building a static regular expression. Remember that our regular expressions patterns should not start with a `/`. To match `/tag/django/` we can use the regular expression `r'^tag/django/$'`. Similarly, `r'^tag/web/$'` will match `/tag/web/`. The goal is to build a regular expression that will match all slugs. As mentioned in Chapter 3, a `SlugField` accepts a string with a limited character set: alphanumeric characters, the underscore, and the dash. We first define a regular expression character set by replacing `django` and `web` with two brackets: `r'^tag/\[/\]/$'`. Any character or character set inside the brackets is a valid character for the string. We want multiple characters, so we add the `+` character to match at least one character: `r'^tag/\[\w-]+/$'`. In Python, `\w` will match alphanumeric characters and the underscore. We can thus add `\w` and `-` (the dash character) to the character set to match a valid slug: `r'^tag/\[\w-\]+/$'`. This regular expression will successfully match `/tag/django/`, `/tag/web/`, and even `/tag/video-games/` and `/tag/video games/`.

Info

In the code above we opted to specify `\[\w-\]+` for the slug match, instead of `\[\w-\]` or `\[-\w\]+`. Python will accept and work correctly with `\[\w-\]+` or `\[-\w\]+`, but the
character set is imprecise. The - character is reserved for specifying ranges, such as [A-Z]+, which will match any capital alphabet character. To specify the - character, we have to escape it with a slash: the \[A-Za-z]+ pattern will match a string of any length that contains only the letters A, Z, or -. However, as you may have guessed, if the dash is specified at the beginning or end of a character set, Python is smart enough to realize that you mean the character rather than a range. Even so, this can be ambiguous to other programmers, and it’s best to always escape the dash when you want to match the - character.

This regular expression matches all of the URLs we actually want, but it will not pass the value of the slug to the tag_detail() function view. To do so, we can use a named group. Python regular expressions identify named groups with the text (?P<name>pattern), where name is the name of the group and pattern is the actual regular expression pattern. In a URL pattern, Django takes any named group and passes its value to the view the URL pattern points to. In our case, we want our named group to use the pattern we just built—\[\w\-]+—and to be called slug. We thus have \(\?P<slug>\[\w\-]\)+. This regular expression will match a slug and pass its value to the view the URL pattern points to. We can now build our URL pattern.

We are building a URL pattern for our tag_detail() view, which exists in the views.py file in our organizer app. We first import the view via a Python import and then create a URL pattern by calling url() and passing the regular expression and the view. Example 5.9 shows the resulting URL configuration in suorganizer/urls.py.

Example 5.9: Project Code

```python
16 from django.conf.urls import include, url
19 from organizer.views import homepage, tag_detail
24 url(r'^tag/(?P<slug>\[\w\-]+)/$', tag_detail),
```

If we request http://127.0.0.1:8000/tag/django or the Django runserver, Django will select our new URL pattern and call tag_detail(request, slug='django').

The regular expression pattern and view pointer are not the only parameters we can pass to url(). It is possible, and highly recommended, to specify the keyword argument name for URL patterns. The utility of specifying name is the ability to refer to a URL pattern in Django, a practice we discuss in Chapter 6: Integrating Models, Templates, Views, and URL Configurations to Create Links between webpages. This practice not only is useful in Django but also allows me to refer to URL patterns in the book without ambiguity.
Chapter 5  Creating Webpages with Controllers in Django

It is possible to name a URL pattern whatever you wish. However, I strongly recommend you namespace your names, allowing for easy reference without conflict across your site. In this book, I use the name of the app, the name of the model being used, and the display type for the object type. We thus name the URL pattern organizer.tag.detail. Our final URL pattern is shown in Example 5.10.

Example 5.10: Project Code

```python
url(r'^tag/(?P<slug>\w\-]+)/$','tag_detail',
    name='organizer_tag_detail'),
```

Info

When I refer to namespaces, I mean it in an informal sense: it’s simply a string that we structure in a particular way. I am not referring to the actual URL namespace tool Django provides that we will use in Chapter 19: Basic Authentication.

Consider all the code we have avoided writing by writing our URL pattern intelligently. At the end of Section 5.4.1, we were considering parsing the raw URL path string (passed to the view via `request.path`) to find the slug value of our tag. Thanks to Django’s smart URL configuration, simply by providing a named group to our regular expression pattern, we can pass values in the URL directly to the view.

### 5.5 Generating 404 Errors for Invalid Queries

As things stand, we can use the command line to start our development server (Example 5.11) and see the fruits of our labor.

Example 5.11: Shell Code

```
$ ./manage.py runserver
```

If you navigate to the address of a valid Tag, you will be greeted by a simple HTML page built from our template. For example, `http://127.0.0.1:8000/tag/django/` will display a simple page about our Django tag. However, what happens if you browse to a URL built with an invalid tag slug, such as `http://127.0.0.1:8000/tag/nonexistent/?`

You’ll be greeted by a page of Django debug information, as shown in Figure 5.4.

Django is displaying a page informing you that Python has thrown an exception. The title of the page, “DoesNotExist at /tag/nonexistent/,” tells us that the URL we asked for does not exist. The subtitle, “Tag matching query does not exist” tells us that the database query for our Tag could not find a row in the database that matched what we desired (in this case, we queried `Tag.objects.get(slug__iexact='nonexistent')`). What's
more, below the initial readout presented in Figure 5.4, you'll find a Python traceback, shown in Figure 5.5, where Django informs us that the Python exception type being raised is DoesNotExist. Of the four functions in the traceback, three are in Django’s source code and therefore (most likely) are not the problem. The second item in the traceback, however, is in /organizer/views.py and reveals that the code throwing the exception is tag = Tag.objects.get(slug__iexact=slug), on line 17. This does not mean the code is wrong (it isn’t!), simply that the problem originates there. The problem, as stated in the top half of the page, is that there is no Tag object with slug “nonexistent” in the database.

This message is obviously not what we want users to be greeted with in the event of a malformed URL. The standard return for such in websites is an HTTP 404 error. Let us

![Figure 5.4: Django Error Message](image1)

![Figure 5.5: Django Error Traceback](image2)
return to our function view in /organizer/views.py and augment it so that it returns a proper HTTP error rather than throwing a Python exception.

Django supplies two ways to create an HTTP 404 error. The first is with the 
HttpResponseNotFound class, and the second is with the Http404 exception.

The HttpResponseNotFound class is a subclass of the HttpResponse class. Like its superclass, HttpResponseNotFound expects to be passed the HTML content it is asked to display. The key difference is that returning an HttpResponse object results in Django returning an HTTP 200 code (Resource Found), whereas returning a HttpResponseNotFound object results in an HTTP 404 code (Resource Not Found).

The Http404 is an exception and as such is meant to be raised rather than returned. In contrast to the HttpResponseNotFound class, it does not expect any data to be passed, relying instead on the default 404 HTML page, which we build in Chapter 29: Deploy! when we deploy our site.

Consider that our code is currently raising a DoesNotExist. We therefore have to catch this exception and then proceed with an HTTP 404 error. It is thus more appropriate and more Pythonic to use an exception, meaning our code in Example 5.12 will use the Http404 exception. Start by importing this in the file, by adding Http404 to the second import line (the one for HttpResponse). The import code should now read as shown in Example 5.12.

**Example 5.12: Project Code**

```
organizer/views.py in 294dabd8cc
1 from django.http.response import (  
2     Http404, HttpResponse)
```

To catch the DoesNotExist exception, we surround our model manager query with a Python try...except block. Should the query raise a DoesNotExist exception for a Tag object, we then raise the newly imported Http404. This leaves us with the code shown in Example 5.13.

**Example 5.13: Project Code**

```
organizer/views.py in 294dabd8cc
17  def tag_detail(request, slug):
18     try:
19         tag = Tag.objects.get(slug__iexact=slug)
20     except Tag.DoesNotExist:
21         raise Http404
22     template = loader.get_template('organizer/tag_detail.html')
23     context = Context({'tag': tag})
24     return HttpResponse(template.render(context))
```

Had we opted to use HttpResponseNotFound, we might have coded as in Example 5.14.
5.6 Shortening the Development Process with Django View Shortcuts

We now have a two-function view in /organizer/views.py, which currently reads as shown in Example 5.16.

```python
from django.http.response import (Http404, HttpResponse)
from django.template import Context, loader
```
from .models import Tag

def homepage(request):
    tag_list = Tag.objects.all()
    template = loader.get_template('organizer/tag_list.html')
    context = Context({'tag_list': tag_list})
    output = template.render(context)
    return HttpResponse(output)

def tag_detail(request, slug):
    try:
        tag = Tag.objects.get(slug__iexact=slug)
    except Tag.DoesNotExist:
        raise Http404
    template = loader.get_template('organizer/tag_detail.html')
    context = Context({'tag': tag})
    return HttpResponse(template.render(context))

That is a lot of code for two simple webpages. We also have a lot of duplicate code in each function, which is not in keeping with the DRY philosophy. Luckily for developers, Django provides shortcut functions to ease the development process and to significantly shorten code such as the preceding.

### 5.6.1 Shortening Code with **get_object_or_404()**

Our first shortcut, `get_object_or_404()`, is a complete replacement for the `try...except` block that currently exists in our `tag_detail()` function.

Let’s start by importing it into our `/organizer/views.py` file, as in Example 5.17.

**Example 5.17: Project Code**

organizer/views.py in 5705e49877

```python
from django.shortcuts import get_object_or_404
```

We can then delete the following lines, as in Example 5.18.

**Example 5.18: Project Code**

organizer/views.py in 294dabd8cc

```python
try:
    tag = Tag.objects.get(slug__iexact=slug)
except Tag.DoesNotExist:
    raise Http404
```
We replace the content in Example 5.18 with the code in Example 5.19.

**Example 5.19: Project Code**

```python
organizer/views.py in 5705e49877
```

```python
18 tag = get_object_or_404(
19     Tag, slug__iexact=slug)
```

The `get_object_or_404()` shortcut expects to have the model class and the desired query passed as arguments and will return the object if it finds one. If not, it raises `Http404`, just as we had programmed before. Because we are passing in the `Tag` object and using exactly the same query, the behavior of our shortened code is exactly the same as that of our original code.

Our `tag_detail()` thus reads as in Example 5.20.

**Example 5.20: Project Code**

```python
organizer/views.py in 5705e49877
```

```python
17 def tag_detail(request, slug):
18     tag = get_object_or_404(
19         Tag, slug__iexact=slug)
20     template = loader.get_template(
21         'organizer/tag_detail.html')
22     context = Context({'tag': tag})
23     return HttpResponse(template.render(context))
```

### 5.6.2 Shortening Code with `render_to_response()`

Most views must do the following:

1. Load a template file as a `Template` object.
2. Create a `Context` from a dictionary.
3. Render the `Template` with the `Context`.
4. Instantiate an `HttpResponse` object with the rendered result.

Django supplies not one but two shortcuts to perform this process for us. The first is the `render_to_response()` shortcut. The shortcut replaces the behavior that we currently have in our views, performing all four tasks listed above. Let’s start by importing it, adding it to the end of our pre-existing shortcut import, as shown in Example 5.21.

**Example 5.21: Project Code**

```python
organizer/views.py in 5ff3dee4fa
```

```python
1 from django.shortcuts import (
2     get_object_or_404, render_to_response)
```
We can now use `render_to_response()` to shorten our code. In our ```homepage()``` view, for instance, we can remove the code shown in Example 5.22.

**Example 5.22: Project Code**

```python
organizer/views.py in 5705e49877

7   def homepage(request):
8       tag_list = Tag.objects.all()
9       template = loader.get_template('organizer/tag_list.html')
10      context = Context({'tag_list': tag_list})
11      output = template.render(context)
```

The code in Example 5.22 is easily replaced with the code in Example 5.23.

**Example 5.23: Project Code**

```python
organizer/views.py in 5ff3dee4fa

7   def homepage(request):
8       return render_to_response('organizer/tag_list.html',
9          {'tag_list': Tag.objects.all()})
```

Observe how we pass in the same path to the template and a simple dictionary with the (identical) values to populate the template. The shortcut does the rest for us: the behaviors in the preceding two code examples are exactly the same.

The process to shorten ```tag_detail()``` is exactly the same. We start by removing the code in Example 5.24.

**Example 5.24: Project Code**

```python
organizer/views.py in 5705e49877

20      template = loader.get_template('organizer/tag_detail.html')
21      context = Context({'tag': tag})
22      return HttpResponse(template.render(context))
```

Then, in Example 5.25, we write a call to ```render_to_response()```, passing in the same values seen in the previous code: the same template path and the same dictionary passed to our ```Context``` instantiation.
Our entire file has been reduced to the code shown in Example 5.26.

```
Example 5.26: Project Code
organizer/views.py in 5ff3dee4fa

16    return render_to_response(
17        'organizer/tag_detail.html',
18        {'tag': tag})
```

The code in Example 5.26 was the original way to shorten code and, while still frequently seen on the Internet and in older projects, is no longer the best way to shorten a simple view. Instead, you’ll want to use `render()`.

### 5.6.3 Shortening Code with `render()`
Before introducing the `render_to_response()` shortcut, our `/organizer/views.py` read as shown in Example 5.27.

```
Example 5.27: Project Code
organizer/views.py in 4d36d603db

1      from django.shortcuts import get_object_or_404
2      from django.shortcuts import render_to_response
3      from django.template import Context, loader
```
from .models import Tag

def homepage(request):
    tag_list = Tag.objects.all()
    template = loader.get_template('organizer/tag_list.html')
    context = Context({'tag_list': tag_list})
    output = template.render(context)
    return HttpResponse(output)

def tag_detail(request, slug):
    tag = get_object_or_404(Tag, slug__iexact=slug)
    template = loader.get_template('organizer/tag_detail.html')
    context = Context({'tag': tag})
    return HttpResponse(template.render(context))

Example 5.27 is sufficient for the simple views we are currently building but will prove to be inadequate in the long run. Specifically, we are not using Django context processors.

At the moment, our views are rendering Template instances with Context instances and passing the result to an HttpResponse object. The problem with this approach is that sometimes Django needs to make changes to the values within the Context objects. To enable Django to make changes to data that render a Template, we must use a RequestContext instead of a Context object. When a Template renders with a RequestContext, Django uses the HttpRequest object to add data to the RequestContext, providing information not available to Context. To do so, Django calls the context processors, which are simply functions that are listed in the TEMPLATES options of /suorganizer/settings.py (Example 5.28).

Example 5.28: Project Code

| suorganizer/settings.py in 4d36d603db |

```python
TEMPLATES = [{
    'OPTIONS': {
        'context_processors': [
            'django.template.context_processors.debug',
            'django.template.context_processors.request',
            'django.contrib.auth.context_processors.auth',
            'django.contrib.messages.context_processors.messages',
        ],
    },
}],
```

At the moment, enabling context processors is of no use to us, but in Chapter 9, we build views and templates that rely on Django context processors. However, it behooves us to examine them now, as they provide insight into our new shortcut.
To make the change to using context processors, we need only change each use of Context to RequestContext. The only difference is that RequestContext needs the HttpRequest object, as it intends to pass it to all the context processors. We therefore pass request to RequestContext before the dictionary of values. Our code now reads as shown in Example 5.29.

Example 5.29: Project Code

```
organizer/views.py in c392ab707a

1  from django.http.response import HttpResponse
2  from django.shortcuts import get_object_or_404
3  from django.template import RequestContext, loader
4
5  from .models import Tag
6
7  def homepage(request):
8      tag_list = Tag.objects.all()
9      template = loader.get_template('organizer/tag_list.html')
10     context = RequestContext(request,
11                               {'tag_list': tag_list})
12     output = template.render(context)
13     return HttpResponse(output)
14
15  def tag_detail(request, slug):
16      tag = get_object_or_404(Tag, slug__iexact=slug)
17      template = loader.get_template('organizer/tag_detail.html')
18      context = RequestContext(request,
19                               {'tag': tag})
20     return HttpResponse(template.render(context))
```

Understanding and using RequestContext or Context has a direct effect on our choice of shortcuts. Prior to Django 1.3, developers would force the render_to_response() shortcut to use the RequestContext object by coding as shown in Example 5.30.

Example 5.30: Python Code

```
return render_to_response('path/to/template.html',
                          data_dictionary,
                          context_instance=RequestContext(request))
```
Many examples online and older projects continue to use this method. However, starting in Django 1.3 (released March 2011), developers should instead use the render() shortcut, which is identical to render_to_response() except that it uses a RequestContext object instead of a Context object and therefore takes the HttpRequest object as a third argument. Specifically, render() does the following:

1. Loads a template file as a Template object
2. Creates a RequestContext from a dictionary (with HttpRequest)
3. Calls all the context processors in the project, adding or modifying data to the RequestContext
4. Renders the Template with the RequestContext
5. Instantiates an HttpResponse object with the rendered result

The render() shortcut thus replaces the project code from Example 5.30, taking three arguments: request, the path to the template file, and the dictionary used to build the RequestContext object. We can follow the same replacement steps used for render_to_response() in the case of render(). Example 5.31 shows the resulting /organizer/views.py.

Example 5.31: Project Code
organizer/views.py in d2ecb7f70d

```python
from django.shortcuts import get_object_or_404, render

from .models import Tag

def homepage(request):
    return render(request,
                  'organizer/tag_list.html',
                  {'tag_list': Tag.objects.all()})

def tag_detail(request, slug):
    tag = get_object_or_404(Tag, slug__iexact=slug)
    return render(request,
                  'organizer/tag_detail.html',
                  {'tag': tag})
```

Using RequestContext is slower than using Context, and therefore render() is slower than render_to_response() (when without the context argument). Nonetheless, most developers now use render() out of the box, choosing to prioritize ease of programming over performance. Using Context or render_to_response(),
particularly in young projects with few users, could be considered a pre-optimization,
limiting functionality in favor of performance. In addition, context processors are not
typically the bottleneck on a website. By the same token, if a context processor is ever
needed on a view using `Context`, more work will be
required to get the context processor working, particularly if the developer is unclear as to
where the problem lies. It is therefore not a bad idea to start with `RequestContext` and
`render()` and replace them if necessary (and if possible!). We reinforce this notion in
Chapter 19 when we opt to use variables created by context processors on every webpage.

In keeping with this logic and with current trends, the rest of the book relies on
`render()` as the de facto view shortcut.

As we move forward, please keep in mind that while similar, `render_to_response()`
and `render()` have very different uses, and many of the examples online should be using
`render()` instead of `render_to_response()`, making this latter shortcut a common
pitfall for beginners when building forms (Chapter 9) or when using the contributed library.

## 5.7 URL Configuration Internals: Adhering to App Encapsulation

We currently have two function views, now masterfully shortened, and two URL patterns,
creating two webpages. However, our URL configuration is in direct violation of app
encapsulation in Django. The URL patterns that direct users to the two webpages generated
by the `organizer` app exist in a file that is for the project: the URLs are in a file under
`souorganizer/`, as opposed to a file within the `organizer/` directory.

The practical goal of this section is to refactor our URL configuration so that our
Django website adheres to the app encapsulation standard. However, to do so, we must learn
much more about the URL configuration. The instructional goal of this section is to teach
you exactly how URL patterns are used and built in Django.

### 5.7.1 Introspecting URL Patterns

We’ve discovered that a URL configuration is a list of URL patterns, stored by convention
in a variable named `urlpatterns`. What I (and others) casually refer to as a URL pattern is
actually a `RegexURLPattern` object. Each call to `url()` instantiates a
`RegexURLPattern`; a URL configuration is thus a list of `RegexURLPattern` objects
stored in a variable named `urlpatterns`.

Each `RegexURLPattern` is instantiated by a call to `url()` (see Example 5.32), which
takes as mandatory arguments (1) a regular expression pattern and (2) a reference to a view.
As an optional argument, it’s possible to pass (3) a Python dictionary, where each key value
is passed to the view as keyword arguments. We will see this in action before the end of the
chapter and then again in Chapter 19. Finally, `url()` will accept (4) a named argument
name, where we can specify the name of the `RegexURLPattern`. We’ve named our
second URL pattern `organizer.tag.detail`, but the utility of names won’t be clear
until Chapter 6.
Example 5.32: Python Code

```python
url(regular_expression,
    view,
    optional_dictionary_of_extra_values,
    name=a_name)
```

Ghosts of Django Past

Prior to Django 1.8, it was possible to point to a view using a string that acted as a Python namespace (similar to imports). For example, we could have used the line in Example 5.33.

Example 5.33: Python Code

```python
url(r'^$', 'organizer.views.homepage')
```

What’s more, while the `urlpatterns` variable was still a simple list, it was convention (but not necessary) to create and process the list using a call to the `patterns()` function, as in Example 5.34.

Example 5.34: Python Code

```python
urlpatterns = patterns('',
    url(regular_expression, view),
)
```

The first argument to `patterns` was the string prefix, which worked in tandem with namespace strings. For instance, the URL configurations in Example 5.35 and Example 5.36 are equivalent.

Example 5.35: Python Code

```python
urlpatterns = patterns('',
    url(regular_expression, 'organizer.views.homepage'),
)
```

Example 5.36: Python Code

```python
urlpatterns = patterns('organizer.views',
    url(regular_expression, 'homepage'),
)
```

The use of `patterns` and namespace strings in URL patterns are deprecated and should not be used. Use direct Python imports (what we are currently using) instead.
Django uses the ROOT_URLCONF setting in settings.py to find the URL configuration for the project. It does so as soon as the server starts (along with settings). This makes Django fast, as the entire regular expression pattern-matching scheme is stored in memory once, but it also means that if you change the URL configuration or any settings, you must restart the Django server (unless you’re running the development server, which anticipates changes).

Because a URL configuration is a list, the order of URL patterns matters, particularly when the URLs matched by regular expression patterns overlap. In Chapter 6, we will see an example of overlapping URLs and how order comes into play.

While we now understand the basics of URL patterns and configurations, we’re still missing a key concept: how to connect different URL configurations.

### 5.7.2 Using include to Create a Hierarchy of URL Configurations

The second argument passed to url() need not point at a view: it can point at another URL configuration, thanks to the include() function. This capability allows us to create a separate URL configuration in each Django app and have a URL pattern in the site-wide URL configuration point to each one. In effect, the full URL configuration is not a simple list but is actually a tree, where the leaves of the tree are webpages (see Figure 5.6).

When a URL pattern points to a URL configuration, the regular expression pattern acts as a URI prefix. For instance, if the path r'blog/' points to a URL configuration, then all of the URL patterns in that URL configuration will effectively have that URI prefixed to their own regular expression.

This functionality comes with an important pitfall: regular expression patterns in URL patterns that point to URL configurations must be treated as partial regular expression patterns: we cannot use the $ character to close the pattern, or it will prevent the use of the ensuing patterns. If a URL pattern with the regular expression pattern r'first/$' points
to a URL configuration with the regular expression pattern `r'first/second/$'`, Django will **effectively** (but not actually, as we'll discuss shortly) combine them for the result of `r'first/second/$'`. Instead of matching `/first/second/` as desired, Django will only match `/first/`. To properly build this URL pattern, the first regular expression must remove the `$`, reading `r'first/'`, so that the combination results in `r'first/second/$'`, as in Example 5.37.

**Example 5.37: Python Code**

```python
# app/urls.py
urlpatterns = patterns(
    url(r'second/$',
        a_view),
)

# project/urls.py
import app.urls as app_url_config

urlpatterns = patterns(
    url(r'first/', # there is no '$' here!
        include(app_url_config)),
)
```

Django is not actually combining regular expressions but rather truncating the URL path it receives. For this reason, the `^` can still be used in `r'first/second/$'`. When a user requests `/first/second/`, Django removes the first `/`, resulting in a request for `first/second/`. Django then uses regular expression pattern `r'first/'` to match `first/second/`. This explains why we cannot use the `$`: `r'first/'` will match `first/second/`, but `r'first/$'` will not. Once Django has selected this URL pattern, it uses the regular expression pattern `r'first/'` to truncate the path from `first/second/` to `second/`, allowing the regular expression pattern `r'second/$'` to match this new path.

Given Django's behavior, a second pitfall is the omission of slashes in intermediate paths. Django only removes the root slash of any URL path. If we use a regular expression pattern `r'first'` (no slash or `$`) to point a URL pattern to a URL configuration containing a URL pattern with a regular expression pattern `r'second/$'`, it will match not `/first/second/` but instead `/firstsecond/`, which is probably not desirable.

What's more, the behavior described above provides us with the reason to always use the `^` regular expression character at the beginning of every regular expression pattern. Without it, we stand to erroneously match URL paths. If we are now using `r'first/'` and `r'second/$'` (no `^`), it will validly match `/first/whoops/second/`, which is probably not what we want either.

We don't actually apply most of this information until we build our blog URL configuration. For our organizer app, we don't want to prefix our path with anything yet. (We will in Chapter 11: Bending the Rules: The Contact Us Webpage when we want the
path /tag/ and /startup/, not /organizer/tag/ or /organizer/startup/.) The prefix we use now is therefore empty.

Start by creating a new file, /organizer/urls.py. In it, we create a new URL configuration. We import the \url{} function to create RegexURLPattern objects. We then create a urlpatterns list to allow Django to find our URL configuration. We can then call \url{} with the same parameters as the ones currently in /suorganizer/urls.py. We end up with a /organizer/urls.py file which reads as in Example 5.38.

**Example 5.38: Project Code**

```python
from django.conf.urls import url
from .views import homepage, tag_detail
urlpatterns = [
    url(r'^$', homepage),
    url(r'^tag/(?P<slug>[\w\-]+)/$', tag_detail,
        name='organizer_tag_detail'),
]
```

To direct Django to this new URL configuration, we need to point our root URL configuration file to this new file using the \include{} function, already included in the Python imports. To start, we need to import the URLs from our organizer app. To avoid name-space clashes, we use the as keyword to rename the urls module organizer.urls. We can then simply point include() to this Python reference. We do this by using the ^ regular expression pattern character, shown in Example 5.39.

**Example 5.39: Project Code**

```python
from django.conf.urls import include, url
from django.contrib import admin
from organizer import urls as organizer_urls
urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^$', include(organizer_urls)),
]
```

If you are still running the development server, it will automatically detect the changes made and reload your URL configuration. If not, restart it by invoking runserver on the command line, as shown in Example 5.40.
Example 5.40: Shell Code

$ ./manage.py runserver

With the development server running, you can now browse to 127.0.0.1:8000 to see our homepage() view and 127.0.0.1:8000/tag/mobile/ to demonstrate our tag_detail() view. Consider that while our URL configuration has changed, the URLs we are able to use have not. We have refactored code, not added new behavior.

5.8 Implementing the Views and URL Configurations to the Rest of the Site

We now have a fundamental understanding of URL configurations and views and have two fully functional webpages using the best tools at our disposal. With these tools, we will now build the rest of the webpages in our site.

5.8.1 Restructuring Our homepage() View

Before we build out new views, it is in our best interest to change our homepage() view to give it a more sensible name and URL path.

Given that it is a list of Tag objects, we should replace the URL pattern so that it matches tag/ as the URL path and provide it with a name, organizer_tag_list, as demonstrated in Example 5.41 in /organizer/urls.py.

Example 5.41: Project Code

```python
organizer/urls.py in 1f86398a5e

1 from django.conf.urls import url
2 from .views import tag_detail, tag_list
3 
4 urlpatterns = [
5 url(r'^tag/$',
6     tag_list,
7     name='organizer_tag_list'),
8     url(r'^tag/(?P<slug>\[\w\-]+)/$',
9         tag_detail,
10     name='organizer_tag_detail'),
11 ]
```

Note that we use ^ and $ in the URL pattern starting on line 6 to carefully define the start and end of the URL path.

In our /organizer/views.py file, we thus need to rename our homepage() view to tag_list(), as in Example 5.42. We make no other changes.
5.8 Implementing the Views and URL Configurations to the Rest of the Site

Example 5.42: Project Code

organizer/views.py in 1f86398a5e

```python
16 def tag_list(request):
17     return render(
18     request,
19     'organizer/tag_list.html',
20     {'tag_list': Tag.objects.all()})
```

Given our changes, http://127.0.0.1:8000/ is no longer a valid URL. Django notes the result of our changes by displaying the list of valid URL patterns, indicating that we may browse to http://127.0.0.1:8000/tag/ or http://127.0.0.1:8000/tag/<slug>/, such as http://127.0.0.1:8000/tag/mobile/, to display valid pages.

5.8.2 Building a Startup List Page

In /organizer/urls.py, we begin by creating a URL pattern for a startup list page, as shown in Example 5.43. Our new URL pattern will direct requests for URL path startup/ to the function view startup_list().

Example 5.43: Project Code

organizer/urls.py in 69767312bf

```python
3 from .views import {
4     startup_list, tag_detail, tag_list
6     }...
7 urlpatterns = [
8     url(r'^startup/$',
9     startup_list,
10     name='organizer_startup_list'),
11     ...
12 ]
```

In /organizer/views.py, we may follow the example of our Tag object list view when building one for Startup objects. In Example 5.44, we load and render the template we built for this purpose and pass in all of the Startup objects in the database to the name of the template variable, which we earlier named startup_list.

Example 5.44: Project Code

organizer/views.py in 69767312bf

```python
4 from .models import Startup, Tag
    ...
7 def startup_list(request):
8     return render(
9     request,
10     'organizer/startup_list.html',
11     {'startup_list': Startup.objects.all()})
```

Remember to add the imports, as shown in Examples 5.43 and 5.44!
5.8.3 Building a Startup Detail Page

As we did for our tag_detail() view, we will now build a startup_detail() view. The function will show a single Startup object, directed to in the URL by the slug field of the model. Our function view thus must take not only a request argument but also a slug argument. In /organizer/views.py, enter the code shown in Example 5.45.

Example 5.45: Project Code
organizer/views.py in bb3aa7eb88

```python
7   def startup_detail(request, slug):
8       startup = get_object_or_404(
9           Startup, slug__iexact=slug)
10      return render(
11          request,
12          'organizer/startup_detail.html',
13          {'startup': startup})
```

As before, we use the slug value passed by the URL configuration to query the database via the Django-provided get_object_or_404, which will display an HTTP 404 page in the event the slug value passed does not match one in the database. We then use render() to load a template and pass the startup object yielded by our query to the template, to be rendered via the template variable of the same name.

In /organizer/urls.py, we direct Django to our new view by adding the URL pattern shown in Example 5.46.

Example 5.46: Project Code
organizer/urls.py in bb3aa7eb88

```python
3   from .views import (
4       startup_detail, startup_list, tag_detail,
5       tag_list)
6   ...
7   urlpatterns = [  
8       url(r'^startup/(?P<slug>[\w\-]+)\/$',
9           startup_detail,
10          name='organizer_startup_detail'),
11       ...
12   ]
```

Note again the ^ and $ characters that define the beginning and end of our URL path and how our use of regular expression named groups allows us to pass the slug portion of the URL directly to our view as a keyword argument. We make sure, as always, to name the URL pattern.
5.8.4 Connecting the URL Configuration to Our Blog App

We’ve created the four display webpages in our organizer app. We will now build two pages in our blog app. To maintain app encapsulation, we must first create an app-specific URL configuration file and then point a URL pattern in the site-wide URL configuration to it.

Start by creating /blog/urls.py and coding the very basic requirements for a URL configuration. This will yield the code shown in Example 5.47.

**Example 5.47: Project Code**

```
blog/urls.py in 02dabec093

1  urlpatterns = [
2  ]
```

In /suorganizer/urls.py we can direct Django to our blog app URL configuration thanks to include(), as shown in Example 5.48.

**Example 5.48: Project Code**

```
suorganizer/urls.py in 02dabec093

19  from blog import urls as blog_urls
20    ...  
21  urlpatterns = [
22    ...  
23    url(r'^blog/', include(blog_urls)),
24    ...  
25  ]
```

Remember that the full URL configuration is actually a tree. If a URL pattern points to another URL configuration, Django will pass the next URL configuration a truncated version of the URL path. We can thus continue to use the `^` regular expression character to match the beginning of strings, but we cannot use the `$` to match the end of a string. When the user requests the blog post webpage, he or she will request `/blog/2013/1/django-training/`. Django will remove the root slash and match the URL path in the request to the URL pattern above, as the regular expression `r'^blog/'` matches the path. Django will use the regular expression pattern `r'^blog/'` to truncate the path to `2013/1/django-training/`. This is the path it will forward to the blog URL configuration and is what we want our post detail view to match.

Before we create a blog post detail view, let us first program a list view for posts.

5.8.5 Building a Post List Page

With our blog app connected via URL configuration, we can now add URL patterns. Let’s start with a list of blog posts.
In /blog/views.py, our function view is straightforward, as you can see in Example 5.49.

Example 5.49: Project Code
blog/views.py in 928c982c03

```python
from django.shortcuts import render
from .models import Post

def post_list(request):
    return render(request, 'blog/post_list.html', {'post_list': Post.objects.all()})
```

We wish to list our blog posts at /blog/. However, this is already the URL path matched by our call to include in suorganizer/urls.py. When a user requests /blog/, Django will remove the root /, and match the URL pattern we just built. Django will then use r'^blog/' to truncate the path from blog/ to the empty string (i.e., nothing). We are thus seeking to display a list of blog posts when Django forwards our blog app the empty string. In Example 5.50, we match the empty string with the regular expression pattern r'^$'.

Example 5.50: Project Code
blog/urls.py in 928c982c03

```python
from django.conf.urls import url
from .views import post_list

urlpatterns = [
    url(r'^$', post_list, name='blog_post_list'),
]
```

5.8.6 Building a Post Detail Page
The final view left to program is our detail view of a single Post object. Programming the view and URL pattern for this view is a little bit trickier than our other views: the URL for each Post object is based not only on the slug but also on the date of the object, making the regular expression pattern and query to the database a little more complicated. Recall that we are enforcing this behavior in our Post model via the unique_for_month attribute on the slug.

Take http://site.djangoproject.com/blog/2013/1/django-training/ as an example. After include() in our root URL configuration truncates
5.8 Implementing the Views and URL Configurations to the Rest of the Site

blog/ from the URL path, our blog app URL configuration will receive 2013/1/
django-training/. Our regular expression pattern must match a year, month, and slug
and pass each one as a value to our view.

The year is four digits, and our named group is thus (?P<year>\d{4}). A month may
have one or two digits, so our named group is (?P<month>\d{1,2}). Finally, and as
before, our slug is any set of alphanumeric, underscore, or dash characters with length
greater than one, so we write our named group as (?P<slug>[\w\-]+). We separate
each part of the URL path with a / and wrap the string with ^ and $ to signify the
beginning and end of the URL path to match. The string containing our regular expression
is thus r'ˆ(?P<year>\d{4})/(?P<month>\d{1,2})/(?P<slug>[\w\-]+)/$'.

To direct Django to a view in /blog/views.py, we may write the call in Example
5.51 to url().

Example 5.51: Project Code
blog/urls.py in cb5dd59383

```python
3    from .views import post_detail, post_list
4         ...
5    urlpatterns = [
6         ...
9        url(r'ˆ(?P<year>\d{4})/','
10           r'(?P<month>\d{1,2})/','
11        r'(?P<slug>[\w\-]+)/$','
12        post_detail,
13        name='blog_post_detail'),
14    ]
```

Warning!

In Example 5.51, we are passing one regular expression pattern, despite that there
appears to be three. Python allows strings to be split into string fragments as long as
there is only whitespace between the string fragments. Note how lines 9 and 10 do
not end with a comma, while line 11 does. This is because the strings on lines 9, 10,
and 11 are all a single string to Python, split this way to fit on the pages of this book.
The r preceding the string fragments makes each fragment a raw string.

Our function view will thus accept four parameters: request, year, month, and slug.
In Example 5.52, in /blog/views.py, start by changing the import to include
get_object_or_404, which we will need for our detail page.

Example 5.52: Project Code
blog/views.py in cb5dd59383

```python
1    from django.shortcuts import {
2       get_object_or_404, render)
```
We must now build a query for the database. Our Post model contains a pub_date field, which we could compare to a datetime.date, but we don’t have the necessary information to build one (we lack the day). For the occasion, Django provides DateField and DateTimeField objects with special field lookups that break each field down by its constituents, allowing us to query pub_date.year and pub_date.month to filter results. In the case of our example URL, http://site.django-unleashed.com/blog/2013/1/django-training/, this functionality allows us to write the query shown in Example 5.53.

Example 5.53: Python Code

```
Post.objects
    .filter(pub__date__year=2014)
    .filter(pub__date__month=11)
    .get(slug__iexact='django-training')
```

While the query to our Post model manager will work, it is more desirable to use the get_object_or_404 to minimize developer-written code. Recall that get_object_or_404 wants a model class and a query string as parameters. Django does not limit the number of query strings passed to get_object_or_404, allowing developers to pass as many as necessary. Given n arguments, the first n-1 will call filter(), while the nth will result in a call to get(). Practically, this means Django will re-create the query in Example 5.53 for us exactly, with the call shown in Example 5.54.

Example 5.54: Project Code

```
post = get_object_or_404(
    Post,
    pub_date__year=year,
    pub_date__month=month,
    slug=slug)
```

The rest of our view is exactly like any other. The view passes the HttpRequest object, a dictionary, and a string to render(). The render() shortcut uses the HttpRequest object and the dictionary to build a RequestContext object. The string passes a path to the template file, allowing render() to load the template and render the template with the RequestContext object. The shortcut then returns an HttpResponse object to the view, which the view passes on to Django. Our final view is thus shown in Example 5.55.

Example 5.55: Project Code

```
def post_detail(request, year, month, slug):
    post = get_object_or_404(
```
WARNING!

This section deals with Python methods and HTTP methods. I will refer to Python methods simply as methods and to HTTP methods as HTTP methods, typically referring to the actual HTTP method in capitals (such as the HTTP GET method or the HTTP OPTIONS method).

Any Python callable that accepts an HttpRequest object as argument and returns an HttpResponse object is deemed a view in Django. So far, we’ve stuck exclusively to using Python functions to create views. Prior to Django 1.3, this was the only recommended way to create views. However, starting in version 1.3, Django introduced a class to allow developers to create view objects.

Django introduced a class to create view objects because coding the class for the view is actually rather tricky and prone to security issues. For this reason, despite the ability to use any Python callable as a view, developers stick to using the Django recommended class or else simply use functions.

The class itself is simply called View, and developers refer to classes that inherit View as class-based views (CBVs). These classes behave exactly like function views but come with several unexpected benefits.

To begin, let’s replace our Post list function view with a class-based view. Example 5.56 shows our current view.

**Example 5.56: Project Code**

```python
blog/views.py in cb5dd59383

    def post_list(request):
        return render(request, 'blog/post_list.html',
        {'post_list': Post.objects.all()})
```

We are not going to change the logic of the function. However, the function must become a method belonging to a class (which implies the addition of the self parameter, required for Python methods). We may name the class whatever we wish, so we shall call it
PostList, but for reasons discussed shortly, the name of the method must be get(), as shown in Example 5.57.

**Example 5.57: Project Code**

```python
from django.views import View

class PostList(View):
    def get(self, request):
        return render(request, 'blog/post_list.html',
                      {'post_list': Post.objects.all()})
```

The import of View typically causes beginners confusion because it implies that View is
generic, leading people to confuse View and class-based views with generic class-based views
(GCBVs). *GCBVs are not the same as CBVs, and making a distinction between the two is crucial.*
We wait until Chapter 17 and Chapter 18 to deal with GCBVs. For the moment, know that
we are building CBVs and that they are different from GCBVs.

Our PostList class inherits from the View class we imported, imbuing it with
(currently unseen) behavior.

The significance of the name of the method get() is that it refers to the HTTP method
used to access it (a primer on HTTP methods is provided in Appendix A). Therefore, our
method will be called only if the user’s browser issues an HTTP GET request to a URL that
is matched by our URL pattern. To contrast, if an HTTP POST request is made, Django
will attempt to call the post() method, which will result in an error because we have not
programmed such a method. We’ll come back to this shortly.

In /blog/urls.py, import the PostList class and then change the URL pattern
to the pattern shown in Example 5.58.

**Example 5.58: Project Code**

```python
from .views import PostList, post_detail
.
.
urlpatterns = [
...
    url(r'^$','PostList.as_view(),
    name='blog_post_list'),
...
]
```

The as_view() method is provided by the inheritance of the View superclass and
ensures that the proper method in our CBV is called. When Django receives an HTTP
GET request to a URL that matches the regular expression in our URL pattern, `as_view()` will direct Django to the `get()` method we programmed. We’ll take a much closer look at exactly how shortly.

### 5.9.1 Comparing Class-Based Views to Functions Views

A CBV can do everything a function view can do. We’ve not seen the use of the URL pattern dictionary previously, and so we’ll now take the opportunity to use a dictionary in both a function view and a CBV to demonstrate similarities. The practical purpose of our dictionary is to override the base template of our view (which we defined in Chapter 4 in the template as `parent_template`), and the learning purpose is to familiarize you with the URL pattern dictionary and CBVs.

To start, we add the dictionary to both blog URL patterns, as shown in Example 5.59.

**Example 5.59: Project Code**

```python
blog/urls.py in d3030ee8d3

5  urlpatterns = [
6    url(r'^$',
7        PostList.as_view(),
8        {'parent_template': 'base.html'},
9        name='blog_post_list'),
10   url(r'^(?P<year>[\d\{4\}])/(?P<month>[\d\{1,2\}])/(?P<slug>\w\-+)/$",
11       post_detail,
12       {'parent_template': 'base.html'},
13       name='blog_post_detail'),
14 ]
```

In our `post_detail` function view, shown in Example 5.60, we must add a named parameter that’s the same as the key in the dictionary (if we had several keys, we’d add several parameters).

**Example 5.60: Project Code**

```python
blog/views.py in d3030ee8d3

8    def post_detail(request, year, month,
9           slug, parent_template=None):
```

To follow through with our example, we need to pass the argument to our template. In Example 5.61, we add `parent_template` to the context dictionary defined in the `render()` shortcut.
Example 5.61: Project Code
blog/views.py in d3030ee8d3

```
15     return render(
16        request,
17        'blog/post_detail.html',
18        {'post': post,
19        'parent_template': parent_template})
```

The process for using the dictionary is almost identical to a CBV. We first add a new parameter to the `get()` method and then pass the new argument to `render()`, as shown in Example 5.62.

Example 5.62: Project Code
blog/views.py in d3030ee8d3

```
22     class PostList(View):
23         def get(self, request, parent_template=None):
24             return render(
25                request,
26                'blog/post_list.html',
27                {'post_list': Post.objects.all(),
28                'parent_template': parent_template})
```

The modification illustrates a key point with CBVs: the view is entirely encapsulated by the class methods. The CBV is a container for multiple views, organized according to HTTP methods. At the moment, illustrating this more directly is impossible, but we revisit the concept in depth in Chapter 9. The bottom line at the moment is that any modification you might make to a function view occurs at the method level of a CBV.

We’re not actually interested in overriding the base templates of our views and so should revert the few changes we’ve made in this section.

### 5.9.2 Advantages of Class-Based Views

The key advantages and disadvantages of CBVs over function views are exactly the same advantages and disadvantages that classes and objects have over functions: encapsulating data and behavior is typically more intuitive but can easily grow in complexity, which comes at the cost of functional purity.

A staple of object-oriented programming (OOP) is the use of instance variables, typically referred to as attributes in Python. For instance, we can usually better adhere to DRY in classes by defining important values as attributes. In `PostList`, we replace the string in `render()` with an attribute (which contains the same value), as shown in Example 5.63.
At the moment, this does us little good on the DRY side of things, but it does offer us a level of control that function views do not offer. Quite powerfully, CBVs allow for existing class attributes to be overridden by values passed to as_view(). Should we wish to change the value of the template_name class attribute, for example, we need only pass it as a named argument to as_view() in the blog_post_list URL pattern, as shown in Example 5.64.

Example 5.64: Project Code
blog/urls.py in 78947978fd

```python
url(r'^$', PostList.as_view(
    template_name='blog/post_list.html'),
    name='blog_post_list'),
```

Even if the template_name attribute is unset, the view will still work as expected because of the value passed to as_view(), as shown in Example 5.65.

Example 5.65: Project Code
blog/views.py in 78947978fd

```python
class PostList(View):
    template_name = ''
```

However, if the template_name attribute is undefined (we never set it in the class definition), then as_view will ignore it.

In the event that template_name is unset and the developer forgets to pass it, we should be raising an ImproperlyConfigured exception. We will see its use in Chapter 17.

Once again, we’re not actually interested in the advantages presented by the changes made in this section, and so I will revert all of the changes made here in the project code.
5.9.3 View Internals

CBVs also come with several much subtler advantages. To best understand these advantages, it’s worth diving into the internals of `View` and seeing exactly what we’re inheriting when we create a CBV.

The easiest place to start is with `as_view()`. In a URL pattern, we use `as_view()` to reference the CBV. Example 5.66 shows an example generic URL pattern.

**Example 5.66: Python Code**

```python
class CBV:
    @classmethod
    def as_view(cls, **initkwargs):
        return cls.as_view(**initkwargs)

class PostList(CBV):
    def as_view(request, *args, **kwargs):
        # magic!
        return view
```

The `as_view()` method is a static class method (note that we call `PostList.as_view()` and not `PostList().as_view()`) and acts as a factory; `as_view()` returns a view (a method on the instance of `PostList`). Its main purpose is to define a (nested) function that acts as an intermediary view: it receives all the data, figures out which CBV method to call (using the HTTP method), and then passes all the data to that method, as shown in Example 5.67.

**Example 5.67: Python Code**

```python
# grossly simplified for your benefit
@classonlymethod
def as_view(cls, **initkwargs):
    def view(request, *args, **kwargs):
        # magic!
        return view
    return view
```

In Example 5.67, the `cls` parameter will be the CBV. In our blog post list URL pattern, `as_view()` will be called with `cls` set to `PostList`. When we passed `template_name` to `as_view()` in `blog/post_list`, `initkwargs` received a dictionary in which `template_name` was a key. Example 5.68 shows the result.

**Example 5.68: Python Code**

```python
as_view(
    cls=PostList,
    initkwargs={
        'template_name': 'blog/post_list.html',
    })
```
To best behave like a view, the nested `view()` method first instantiates the CBV as the `self` variable (demonstrating exactly how flexible Python is as a language). The `view()` method then sets a few attributes (removed from the example code) and calls the `dispatch()` method on the newly instantiated object, as shown in Example 5.69.

**Example 5.69: Python Code**

```python
# still quite simplified
@classonlymethod
def as_view(cls, **initkwargs):
    def view(request, *args, **kwargs):
        self = cls(**initkwargs)
        ...
        return self.dispatch(request, *args, **kwargs)
    return view
```

For clarity’s sake, I want to reiterate that passing undefined attributes to `as_view()` will result in problems because `as_view()` specifically checks for the existence of these attributes and raises an `TypeError` if it cannot find the attribute, as shown in Example 5.70.

**Example 5.70: Python Code**

```python
# still quite simplified
@classonlymethod
def as_view(cls, **initkwargs):
    for key in initkwargs:
        ...
        if not hasattr(cls, key):
            raise TypeError(...)
    def view(request, *args, **kwargs):
        self = cls(**initkwargs)
        ...
        return self.dispatch(request, *args, **kwargs)
    return view
```

If `as_view()` is the heart of View, then `dispatch()` is the brain. The `dispatch()` method, returned by `view()`, is actually where the class figures out which method to use. `dispatch()` anticipates the following developer-defined methods: `get()`, `post()`, `put()`, `patch()`, `delete()`, `head()`, `options()`, `trace()`. In our `PostList` example, we defined a `get()` method. If a `get()` method is defined, View will automatically provide a `head()` method based on the `get()` method. In all cases, View implements an `options()` method for us (the HTTP OPTIONS method is used to see which methods are valid at that path).

In the event the CBV receives a request for a method that is not implemented, then `dispatch()` will call the `http_method_not_allowed()` method, which simply returns
an HttpResponseNotAllowed object. The HttpResponseNotAllowed class is a subclass of HttpResponse and raises an HTTP 405 “Method Not Allowed” code, informing the user that that HTTP method is not handled by this path.

This behavior is subtle but very important: by default, function views are not technically compliant with HTTP methods. At the moment, all of our views are programmed to handle GET requests, the most basic of requests. However, if someone were to issue a PUT or TRACE request to our pages, only the PostList CBV will behave correctly by raising a 405 error. All of the other views (function views) will behave as if a GET request had been issued.

If we wanted, we could use the require_http_methods function decorator to set which HTTP methods are allowed on each of our function views. The decorator works as you might expect: you tell it which HTTP methods are valid, and any request with other methods will return an HTTP 405 error. For example, to limit the use of GET and HEAD methods on our Post detail view, we can add the decorator, as demonstrated in Example 5.71.

```
Example 5.71: Project Code
blog/views.py in 34baa4dfc3

3   from django.views.decorators.http import *
4    require_http_methods
5    ...
10  @require_http_methods(['HEAD', 'GET'])
11  def post_detail(request, year, month, slug):
```

---

**Info**

The use of @require_http_methods(['GET', 'HEAD']) is common enough that Django provides a shortcut decorator called require_safe to help shorten your code by just a bit.

Even so, the decorator doesn’t provide automatic handling of OPTIONS, and organizing multiple views according to HTTP method results in simpler code, as we shall see in Chapter 9.

### 5.9.4 Class-Based Views Review

A CBV is simply a class that inherits View and meets the basic requirements of being a Django view: a view is a Python callable that always accepts an HttpRequest object and always returns an HttpResponse object.

The CBV organizes view behavior for a URI or set of URIs (when using named groups in a regular expression pattern) according to HTTP methods. Specifically, View is built such that it expects us to define any of the following: get(), post(), put(), patch(), delete(), trace(). We could additionally define head(), options(), but View will automatically generate these for us (for head() to be automatically generated, we must define get()).
Internally, the CBV actually steps through multiple view methods for each view. The \texttt{as\_view()} method used in URL patterns accepts \texttt{initkwargs} and acts as a factory by returning an actual view called \texttt{view()}, which uses the \texttt{initkwargs} to instantiate our CBV and then calls \texttt{dispatch()} on the new CBV object. \texttt{dispatch()} selects one of the methods defined by the developer, based on the HTTP method used to request the URI. In the event that the method is undefined, the CBV raises an HTTP 405 error.

In a nutshell, \texttt{as\_view()} is a view factory, while the combination of \texttt{view()}, \texttt{dispatch()}, and any of the developer-defined methods (\texttt{get()}, \texttt{post()}, etc.) are the actual view. Much like a function view, any of these view methods must accept an \texttt{HttpRequest} object, a URL dictionary, and any regular expression group data (such as \texttt{slug}). In turn, the full combined chain (\texttt{view()}, \texttt{dispatch()}, etc.) must return an \texttt{HttpResponse} object.

At first glance, CBVs are far more complex than function views. However, CBVs are more clearly organized, allow for shared behavior according to OOP, and better adhere to the rules of HTTP out of the box. We will further expand on these advantages, returning to the topic first in Chapter 9.

Our understanding of views will change in Chapter 9 and Chapter 17, but at the moment, the rule of thumb is as follows: if the view shares behavior with another view, use a CBV. If not, you have the choice between a CBV and a function view with a \texttt{require\_http\_methods} decorator, and the choice is pure preference. I personally stick with CBVs because I find the automatic addition of the HTTP OPTIONS method appealing, but many opt instead to use function views.

## 5.10 Redirecting the Homepage

If you run Django’s development server and navigate to the root of the website, you’ll discover that we’ve missed a spot, as shown in Example 5.72.

### Example 5.72: Shell Code

```
$ ./manage.py runserver
```

Browsing to \url{http://127.0.0.1:8000/} will display an error page telling us the URL configuration doesn’t have a route for this page. While we’ve created a very detailed and clean URL configuration for all of our URLs, we’ve omitted the homepage, the root of our website.

We want to show the list of blog posts on the homepage. There are several ways we can go about doing so.

### 5.10.1 Directing the Homepage with URL Configurations

The first and perhaps most obvious way would be to create a new URL pattern to send the route to the view we have already built. In \texttt{/suorganizer/urls.py}, we could add the URL pattern shown in Example 5.73 to the URL configuration.
Example 5.73: Project Code
suorganizer/urls.py in 3ddb5f3810

20 from blog.views import PostList
  ... 
23 urlpatterns = [
24    url(r'^$', PostList.as_view()),
    ... 
29  ]

The regular expression pattern: ^ starts the pattern, while $ ends the pattern. This matches '', which is what the root of the URL is to Django, given that it always strips the first /.

Similarly, given that the PostList view is the root of the blog URL configuration, the URL pattern could also be as shown in Example 5.74.

Example 5.74: Project Code
suorganizer/urls.py in 4dc1d03a79

23    url(r'^$', include(blog_urls)),

Neither of the solutions presented above is desirable, as they both corrupt the cleanliness and simplicity of our site URLs. In the first instance, http://site.django-unleashed.com/blog/ and http://site.django-unleashed.com/ are now exactly the same. In the second case, we have created an entire branch of URLs, which is far worse. Not only are http://site.django-unleashed.com/blog/ and http://site.django-unleashed.com/ the same page, but so is http://site.django-unleashed.com/blog/2013/1/django-training/ and http://site.django-unleashed.com/2013/1/django-training/ (note the missing blog/ in the second URL path). This will effectively create a duplicate of every URL the blog already matched.

Our website should maintain a clean URL scheme. Short of creating a separate homepage view, directing our homepage to an existing view as above is undesirable.

5.10.2 Redirecting the Homepage with Views

Rather than simply displaying a webpage on our homepage, we will instead redirect the user to the desired URL. In this instance, http://site.django-unleashed.com/ will redirect to http://site.django-unleashed.com/blog/, which is the post_list() view.

To redirect a URL, we need a view. This creates a minor problem: we are redirecting our site-wide homepage with a view, which at this point exists only in app directories. However, this code does not belong in either our organizer or blog apps. Although Django does not
anticipate the need for site-wide views.py, nothing is stopping us from creating /suorganizer/views.py. Inside, we write the code shown in Example 5.75.

Example 5.75: Project Code

{% highlight python %}
from django.http import HttpResponseRedirect

def redirect_root(request):
    return HttpResponseRedirect('/blog/')
{% endhighlight %}

The HttpResponseRedirect class is a subclass of HttpResponse with special properties, just like HttpResponseNotFound. Given a URL path, it will redirect the page using an HTTP 302 code (temporary redirect). Should you wish for an HTTP 301 code (permanent redirect), you could instead use HttpResponsePermanentRedirect. Note that doing so in development can result in unexpected behavior because the browser will typically cache this response, resulting in difficulties should you change the behavior.

In /suorganizer/urls.py, we can import the new view and replace our previous URL pattern with the one in Example 5.76.

Example 5.76: Project Code

{% highlight python %}
from .views import redirect_root

urlpatterns = [
    url(r'^$', redirect_root),
    ...
]
{% endhighlight %}

Running the deployment server with $ ./manage.py runserver and navigating a browser to http://127.0.0.1:8000/ will result in a redirect to http://127.0.0.1:8000/blog/.

The behavior is what we desire, but our implementation could be improved. Instead of using HttpResponseRedirect, we can use a Django shortcut, redirect(). Our /suorganizer/views.py will now look like Example 5.77.

Example 5.77: Project Code

{% highlight python %
from django.shortcuts import redirect

def redirect_root(request):
    return redirect('/blog/')
{% endhighlight %}
The code in Example 5.77 will work exactly as if we were still using `HttpResponseRedirect`, with an HTTP 302 code. Should we wish to switch to an HTTP 301 code, we could pass `permanent=True` to the shortcut, as in:

```python
redirect('/blog/', permanent=True).
```

The advantage to `redirect()` is that, unlike `HttpResponseRedirect`, it does not need a URL path (currently used in Example 5.77). To better adhere to the DRY principle, we can instead use the name of the URL pattern we wish to redirect to, as shown in Example 5.78.

Example 5.78: Project Code
```
suorganizer/views.py in ba8c7c5e89
4    def redirect_root(request):
5        return redirect('blog_post_list')
```

The shortcut in Example 5.78 is exactly what we want, but it may be a little opaque. Unlike the shortcuts we’ve seen before, we don’t currently understand everything going on under the hood. Specifically, we don’t know how the shortcut builds a proper URL path from the URL pattern. We will see exactly how to do this in the next chapter and revisit this shortcut then.

Note that our way of redirecting, with a site-wide function view, is not the way you would redirect in an actual project. Our method is in direct violation of DRY, but we won’t be able to fix that until Chapter 17.

In short, the behavior above is exactly what we want, and the code is the best we can write given our current knowledge. Please keep this function and behavior in mind going forward, as we will revisit it in Chapter 6 and replace it in Chapter 17.

5.11 Putting It All Together

In Chapter 5, we examined Django views and URL configurations, which make up the Controller of Django’s Model-View-Controller architecture. The Controller acts as the glue between the Model and View. In Django, the Controller receives, selects, processes, and then returns data.

A Django view is any callable that receives an `HttpRequest` object (with any other additional arguments) and returns an `HttpResponse` object. Originally, Django recommended using functions as views, but modern Django also provides a canonical way of creating classes to create object views.

To make writing views easier, Django supplies shortcut functions. We saw three shortcuts, starting with `get_object_or_404()`, which gets an object according to a model class and query while accounting for the possibility that the query will not match a row in the database, in which case it raises a `Http404` exception. We then saw the `render_to_response()` and `render()`. Both load a template, render the template with a context, and instantiate an `HttpResponse` object with the result. However, the `render()` shortcut uses a `RequestContext` instead of a `Context` object, allowing Django to run context processors on the `RequestContext`, effectively adding values for the template to
During the rendering phase. This added functionality is key to certain views and functionality, and thus `render()` has become the favored shortcut, even if it is marginally slower than `render_to_response()`.

To direct Django to the views the developer writes, Django provides the URL configuration mechanism. The URL configuration is contained in a file pointed to by the project settings. Inside the file, Django expects (by convention) to find the `urlpatterns` variable, which is a list of `RegexURLPattern` objects. Each `RegexURLPattern` object is created by a call to `url()`, which expects at the very least (1) a regular expression pattern and (2) a reference to a Python callable. A call to `url()` may also optionally be called with (3) a dictionary of values that are passed as keywords to the view that the resulting `RegexURLPattern` points to. Finally, `url()` can receive (4) the keyword argument `name`, which sets the name of the `RegexURLPattern`, a feature that will become crucial in the next chapter.

URL configurations may be connected using `include()`. A URL pattern may include another URL configuration, allowing for the creation of a URL scheme that is a tree, where the root URL configuration specified in the Django project settings is the root of the tree. This feature furthermore allows for app encapsulation, allowing each app to define its own URL patterns, extending those of the project, which `include()` achieves by truncating the regular expression match from the URL path requested of Django.

In short, the URL configuration directs Django to the view thanks to URL patterns, which contains the logic required to make a webpage.
This page intentionally left blank
Symbols and Numbers

\{\}, \{%.\}, \{#.#\} (Delimiters), in template project code, 80
200 (Request Valid), HTTP response codes, 758
400 (Bad Request), HTTP response codes, 727–728, 758
403 (Forbidden), HTTP response codes, 483, 485–487, 727–728, 758
404 (Invalid Query), HTTP response codes, 132–135, 758
500 (Internal Server Error), HTTP response codes, 727–728, 741, 758

A

About page, 356–359. See also Flatpages app
Abstract syntax tree (AST), 773
AbstractUser, User model inheriting from, 559–560
Accounts. See User accounts
Activation cleaning up URL patterns, 545
creating user accounts, 517, 529–535
resending account activation, 538–544
Actor model, signals and, 660
Add page, configuring admin app, 596–604
Add view, admin app for providing for models. See Creation pages (add view)
Add-ons. See Backing services (add-ons)
Admin actions, creating, 616–618
Admin app adding change password page, 604–612
adding information to list view, 589–592
adding Profile to UserAdmin, 613–615
changing passwords, 504}

configuring add and edit pages, 584–588, 596–604
configuring for User model, 593
configuring list of Post objects, 581–584
configuring list page, 593–596
creating admin actions, 616–618
creating user apps, 464–465
importing and registering app models with, 580
interactions with, 578–579
introduction to, 577
modifying admin controls for blog posts, 581
permissions and, 559
summary of, 618
URL pattern conflict and, 468
working with flatpages form, 355

Admin library. See Admin app
Admin pages, optimizing, 679
Admin panel, 617–618
Aggregate functions, performing database calculations with, 591
all(), manager methods, 81
Alphanumeric characters. See Characters/character sets
Amazon, deployment services, 726
Anonymous users, User model, 462–463
API, options for interacting with email services, 743–745
App registry, listing available apps, 282
AppConfig object, loading, 650–651
Apps. See also by individual types adding static content to, 374–376
adding to projects, 353
building app-generic templates, 108–109
connecting to new apps in settings.py, 19–21
contributed apps, 328
<table>
<thead>
<tr>
<th>Key (continued)</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apps</td>
<td>13, 17–19</td>
</tr>
<tr>
<td>creating</td>
<td></td>
</tr>
<tr>
<td>loading AppConfig objects for</td>
<td>650–651</td>
</tr>
<tr>
<td>refactoring code to adhere to app encapsulation standard</td>
<td>143</td>
</tr>
<tr>
<td>selecting third-party apps for project</td>
<td>751–752</td>
</tr>
<tr>
<td>using app-generic templates</td>
<td>109–112</td>
</tr>
<tr>
<td>App objects, interacting directly with</td>
<td>650–651</td>
</tr>
<tr>
<td>Archiving blog posts</td>
<td></td>
</tr>
<tr>
<td>adding behaviors for indexing archive</td>
<td>414–415</td>
</tr>
<tr>
<td>adding behaviors for monthly archive</td>
<td>408</td>
</tr>
<tr>
<td>adding behaviors for yearly archive</td>
<td>401</td>
</tr>
<tr>
<td>linking to monthly archive</td>
<td>414</td>
</tr>
<tr>
<td>selecting dates for yearly archive</td>
<td>402–403</td>
</tr>
<tr>
<td>sitemap for</td>
<td>715–718</td>
</tr>
<tr>
<td>template for monthly archive</td>
<td>410–413</td>
</tr>
<tr>
<td>template for yearly archive</td>
<td>404–408</td>
</tr>
<tr>
<td>views and URL configurations for monthly archive</td>
<td>408–410</td>
</tr>
<tr>
<td>views and URL configurations for yearly archive</td>
<td>403–404</td>
</tr>
<tr>
<td>Arguments, template</td>
<td>690</td>
</tr>
<tr>
<td>as_manager() method, in QuerySet class</td>
<td>624</td>
</tr>
<tr>
<td>as_view() method, inspecting internals of class-based views</td>
<td>160–162</td>
</tr>
<tr>
<td>ASCII character set</td>
<td>772</td>
</tr>
<tr>
<td>asctime variable, in logging</td>
<td>453</td>
</tr>
<tr>
<td>AST (abstract syntax tree)</td>
<td>773</td>
</tr>
<tr>
<td>Atom feeds</td>
<td>707–714</td>
</tr>
<tr>
<td>Attributes</td>
<td></td>
</tr>
<tr>
<td>advantages of class-based views</td>
<td>158</td>
</tr>
<tr>
<td>for feeds</td>
<td>709</td>
</tr>
<tr>
<td>of fields in Django Post model</td>
<td>38</td>
</tr>
<tr>
<td>ordering attribute</td>
<td>594</td>
</tr>
<tr>
<td>ordering lists by date</td>
<td>48</td>
</tr>
<tr>
<td>organizing data by</td>
<td>33–34</td>
</tr>
<tr>
<td>priority attribute of webpages</td>
<td>717–718</td>
</tr>
<tr>
<td>Auth app</td>
<td></td>
</tr>
<tr>
<td>adding login and logout features to</td>
<td>458–463</td>
</tr>
<tr>
<td>anatomy of</td>
<td>457–458</td>
</tr>
<tr>
<td>creating user apps</td>
<td>464–465</td>
</tr>
<tr>
<td>forcing</td>
<td>484</td>
</tr>
<tr>
<td>introduction to</td>
<td>451</td>
</tr>
<tr>
<td>logging configuration</td>
<td>452–455</td>
</tr>
<tr>
<td>post-authentication redirection</td>
<td>471–472</td>
</tr>
<tr>
<td>sessions and cookies and</td>
<td>456–457</td>
</tr>
<tr>
<td>starting projects and</td>
<td>752</td>
</tr>
<tr>
<td>summary of</td>
<td>472</td>
</tr>
<tr>
<td>views for making login and logout pages</td>
<td>465–471</td>
</tr>
<tr>
<td>Authentication, extending functionality of anatomy of auth app</td>
<td>545–547</td>
</tr>
<tr>
<td>building forms</td>
<td>527–529</td>
</tr>
<tr>
<td>changing passwords</td>
<td>503–506</td>
</tr>
<tr>
<td>cleaning up URL patterns</td>
<td>544–545</td>
</tr>
<tr>
<td>creating user accounts</td>
<td>517</td>
</tr>
<tr>
<td>disabling user accounts</td>
<td>513–516</td>
</tr>
<tr>
<td>introduction to</td>
<td>501</td>
</tr>
<tr>
<td>mixins for sending and logging emails</td>
<td>518–527</td>
</tr>
<tr>
<td>password views</td>
<td>501–503</td>
</tr>
<tr>
<td>resending account activation</td>
<td>538–544</td>
</tr>
<tr>
<td>resetting passwords</td>
<td>506–513</td>
</tr>
<tr>
<td>summary of</td>
<td>547</td>
</tr>
<tr>
<td>templates for creating accounts</td>
<td>535–538</td>
</tr>
<tr>
<td>views for creating and activating accounts</td>
<td>529–535</td>
</tr>
<tr>
<td>Author, adding to blog posts</td>
<td>572–576</td>
</tr>
<tr>
<td>Automatons, forms as</td>
<td>190</td>
</tr>
</tbody>
</table>

B

<table>
<thead>
<tr>
<th>Key</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Back-end programming</td>
<td>5–6</td>
</tr>
<tr>
<td>Backing services (add-ons)</td>
<td></td>
</tr>
<tr>
<td>caching with memcache</td>
<td>745–748</td>
</tr>
<tr>
<td>Heroku</td>
<td>726–727</td>
</tr>
<tr>
<td>logging service</td>
<td>741–743</td>
</tr>
<tr>
<td>overview of</td>
<td>741</td>
</tr>
<tr>
<td>sending email with Postmark email service provider</td>
<td>743–745</td>
</tr>
</tbody>
</table>
Base classes, Manager class as, 623
BaseUserManager, UserManager inheriting from, 561–562
BDD (behavior-driven development), 753
Behaviors
  anticipating behavior overrides, 388–392
  extending pagination behavior with GCBVs, 423–425
  organizing data by, 33–34
  signals for handling, See Signals
Behaviors, generic
  in GCBVs, 385–388
  for indexing blog post archive, 414–415
  linking to monthly archive of blog posts, 414
  for monthly archive of blog posts, 408
  overview of, 401
  selecting dates for yearly archive of blog posts, 402–403
  template for monthly archive of blog posts, 410–413
  template for yearly archive of blog posts, 404–408
  views and URL configurations for monthly archive of blog posts, 408–410
  views and URL configurations for yearly archive of blog posts, 403–404
  for yearly archive of blog posts, 401
Berners-Lee, Sir Tim, 35
Binding data, to forms, 190
Blog app (start up organizer)
  adding author to blog posts, 572–576
  adding styles to, 381
  adding URL pattern dictionary, 157–158
  in app registry, 282
  automatically assigning Tag objects to Post instances, 655–660
  blog post archive sitemap, 715–718
  building detail view for blog posts, 152–155
  building page for listing blog posts, 151–152
  class mixins for applying permissions to blog post, 495–496
  class-based views in, 155–157
  completing Post model for running, 38–40
  configuring list of Post objects, 581–584
  connecting to new apps in settings.py, 19–21
  connecting URL configuration to, 151
  creating apps with manage.py, 17–19
  creating custom managers, 622–624
  creating links on object detail pages, 184–186
  creating new project and apps, 13
  custom template tag for displaying related blog posts, 690–697
  custom template tag for listing latest blog posts, 701–706
  displaying future posts in template, 497–499
  generating project structure, 14–15
  group permissions and, 481–482
  importing decorator into, 494–495
  inspecting internals of class-based views, 160–162
  linking list pages to detail pages, 178
  migration use with, 280
  modifying admin controls for blog posts, 581
  Post sitemap, 715–718
  preparing deployment settings, 730–734
  project specifications, 12–13, 749–751
  redirecting homepages and, 163–164, 186–187
  replacing CBVs with GCBVs, 397
  root project directory, 15
  selecting Django and Python versions, 11–12
  summary, 21–22
  template for list of blog posts, 101–102
  template for single blog post, 100–101
  viewing project installation via manage.py, 15–17
Blog posts. See Posts, blog
Booleans, using with permissions, 495
Bound forms
  displaying unbound and invalid forms, 242
  values in tag_form.html, 216–217
Bug fixes, source code and versioning and, 324
Buttons, for linking to object creation forms, 379
@cache_page() decorator, 684
@cached_property decorator, 664–665

Caching
- entire webpages, 682–684
- information, 662
- limiting database queries, 663
- with memcache, 745–748
- permissions, 481
- properties, 664–665
- template files, 680–681
- template variables, 665–667

Canonical URLs, 173

cap_first(), importing tools from Django, 633

CAPTCHA system, controlling account reset and activation, 539

Case sensitivity
cap_first(), 633
string capitalization criteria, 68–71

CBGVs (class-based generic views).
See GCBVs (generic class-based views)

CBVs (class-based views)
- advantages of, 158–159
- building ContactView, 304–306
- class mixins, 254–256
- comparing with function views, 125, 157–158
- comparing with GCBVs, 394
- comparing with views, 125
- converting function views to, 337–338, 384–385
- creating with get and post methods, 302
- generic. See GCBVs (generic class-based views)
- inspecting internals of, 160–162
- multiple inheritance in Python and, 762
- overview of, 155–157
- replacing tag_create() with TagCreate CBV, 246–249
- review of, 162–163
- starting projects and, 753
- webpages and, 316

Change password page, adding to admin app, 604–612

Change view (edit pages). See Edit pages (change view)

Characters/character sets
- ASCII and Unicode, 772
- defining regular expression for character matching, 130–131
- reversing regular expressions patterns with, 171–172
- URL reversal and, 170–171

check command
- checking production settings, 738
- running before migration, 50
- check_unique(), validation tool, 636–637, 640–641

Class mixins
- ActivationMailFormMixin, 542
- applying permissions to blog post, 495–496
- BasePostFeedMixin, 709
- deleting Startup and Tag objects, 273–274
- GCBVs compared with, 383
- making PostGetMixin generic, 432–438
- PageLinkMixin, 419
- PostGetMixin, 429–432
- ProfileGetObjectMixin, 553
- for sending and logging emails, authentication, extending functionality of, 518–527
- transforming into GCBV, 394–395

Class-based generic views. See GCBVs (generic class-based views)

Classes. See also by individual types
- adding methods to models, 45–47
- attaching managers to model classes, 59
- base classes, 623
- controlling model behavior using nested meta classes, 47–49
- creating models and, 37–38
- for feeds, 708
- mapping Python classes to database tables, 64
- mixins. See Class mixins
- multiple inheritance in Python, 762–763
- reasons for using for generic views, 393–394

Classy Class-Based Views tool, 400, 418
Clean methods
  creating clean method for Tag model
  name field, 198–199
  creating clean method for Tag model
  slug field, 199–201
  validation techniques, 197–198
  `clean_value()`, validation tool, 636–637,
  640–641
Cleaned data
  compared with raw data, 192
  demonstrating use of `TagForm` in shell,
  193–197
  implementing `save()` method with
  `TagForm`, 192–193
  validation of contact form, 303
CLI (command-line interface), Heroku
  logging into Heroku, 738
  overview of, 728
Clients, HTTP, 757
Cloud computing
  deployment options, 725–726
  separating state from behavior, 729
Codebases. See Libraries (codebases)
Columns, table, 765
Compilation, 773–774
Conditions, replacing with variables,
  218–220
Contact Us webpage (contact app)
  building contact form, 302–304
  creating a contact app, 300–301
  creating contact webpage, 301–302
  Django functionality in, 326
  reasons for not using GCBVs, 400–401
  splitting `urls.py` file into smaller
  modules, 308–310
  summary of, 308–310
  template for displaying contact form,
  306–308
  URL pattern and view for interacting
  with contact form, 304–306
ContactView CBV, building for Contact
  Us app, 304–306
Contenttypes app
  authentication and, 451
  creating `ContentType` model
  from, 478
  keeping track of app content types,
  473–476
Context
  making changes to values in `Context`
  objects, 140–143
  for rendered templates, 318
  using templates in Python `Context` class,
  112–116
Contrib directory, 325–327
Contributed apps
  contributed Library (contrib), 328
  enabling flatpages and, 353
  staticfiles contributed app, 373
Contributed Library (contrib)
  contrib directory, 325–327
  contributed apps, 328
  creating app to interact with, 365
  introduction to, 323
  overview of, 19
  source code and versioning and,
  323–325
  staticfiles contributed app, 373
  summary of, 329
  translation framework and, 328–329
Controller, in MVC architecture
  advantages of Models and Views over
  controller, 27
  building website using only Controller,
  23
  developer preferences, 687
  function of, 8–9
  URL configuration and view interaction,
  121–122
  views. See views (Django)
Controls, modifying admin controls for blog
  posts, 581
Cookies
  authentication and, 451
  security of, 456–457
Core app
  creating for data migration in flatpages
  app, 365–366
  creating user apps, 464–465
  coverage tool, testing with, 777
Create, in CUD
  creating `NewsLink` objects in a view,
  252–254
  creating `Post` objects in a view, 249–250
  creating `Startup` objects in a view,
  251–252
Create, in CUD (continued)
custom template tag for displaying create or update forms, 697–701
template for creating StartupForm, NewsLinkForm, and PostForm, 227–229
template for creating Tag objects, 211–213
create_superuser(), 562–563
CreateAccount view, views for creating and activating accounts, 529–530
CreateModel
blog migration and, 55
creating Post model, 51–53
create_superuser command, 645–647
create_tag command, 628–630
createuser command handle() and, 638–643
importing tools from Django, 633–634
interactive and noninteractive, 630–633
prompting developer for password (getpass), 644–645
try.except block in interactive code, 643–644
validation tools, 636–637
Creation pages (add view)
admin app for providing for models, 581
building generic object, 394–395
configuring admin app, 584–588
cross-site request forgery. See CSRF
crash–site request forgery)
CRUD (created, read, updated, destroyed)
data management, 189
organizing URLs and, 750–751
Cryptography
creating user accounts, 517
resetting passwords and, 507
CSRF (cross-site request forgery)
creating Tag objects and, 212–213
issuing POST requests and, 268–270
protecting against, 769–770	tokens, 505
@csrf_protect decorator
disabling user accounts, 514
resending account activation, 541
CSS (Cascading Style Sheets)
adding to websites, 373
applying styles to fieldsets, 597
building custom template tag, 695–696
creating CSS files in directories, 374
creating stylesheet for entire website, 376
in custom template tag for displaying create or update forms, 699
in display of webpages, 6
Espresso tool, 777
integrating CSS content into sites, 377–381
CUD (create, update, delete)
adding CUD webpages for objects, 189
admin app and, 577
Custom
decorators, 488–495
managers, 622–624
template filters, 688–689
template tags. See Template tags, custom
Custom user, overriding authentication
adding author to blog posts, 572–576
creating manager for, 561–563
creating user profile, 550
extending User model, 558–561
integrating forms and templates, 566–567
introduction to, 549
Profile model, 550–552
ProfileDetail view, 552–555
ProfileUpdate, 555–557
PublicProfileDetail, 557–558
replacing old auth versions of User model, 564–566
summary of, 576
User migration, 568–572

D

Data
accessing, 34–35
binding to forms, 190
cleaned vs. raw, 192
connecting with through relations, 65–68
CRUD (created, read, updated, destroyed), 189
data in memory vs. data in database,
64–65
natural, 625
normalization of, 766
organizing into models, 32–34, 36
serialization of, 622, 625–626
Data migrations
  creating core app for, 365–366
  creating Profile model, 551
  for flatpages app, 369–370
  overview of, 280
  post data, 287–288
  for sites app, 365–369
  startup data, 285–287
  for swappable models, 565
  tag data, 280–284
User model, 568–572
database injection attacks, ORM protecting against, 769
Database managers, 765
Databases
  building query for, 154
  caching properties for optimization, 664–665
  creating or modifying via migrations, 53–55
  creating using migrate command, 16
  data in memory vs. data in database, 64–65
  interacting with via managers, 58–63
  interacting with via models, 56–58
  limiting queries, 663
  optimizing, 679–680
  performing calculations with aggregate functions, 591
  reasons for using, 32
  relational, 765–767
  schema of, 279
  selecting for deployment, 729
  selecting when starting project, 753
  tools for communicating with, 10
Datacenters, for deployment, 725
date template filter, for customizing output, 95–96
DATE_FORMAT argument, 95–96
DateDetailView GCBV, applied to PostDetail, 426–429
db.models package, 622
DDOS (distributed denial-of-service) attacks, 661
Deployment
  adding backing services (add-ons), 741
  caching with memcache, 745–748
  checking production settings, 738
  creating error pages, 727–728
  to Heroku, 738–741
  introduction to, 725–726
  logging service for, 741–743
  preparing for, 726–727
  running development server, 735
  running foreman's server, 735–737
  sending email with Postmark, 743–745
  settings, 730–735
  summary of, 745–748
  tools for, 728–729
Deprecation (Python), 684
Deprecation (Django), 626
Design consistency, template inheritance for, 102
Detail pages
  adding URL pattern, 130–132
  anticipating behavior overrides, 388–392
  building for blog posts, 152–155
  building for generic object, 384–388
**Detail pages (continued)**
- building for startup, 150
- building for tags, 128–130
- creating links for, 178–181
- creating links on, 184–186
- linking list pages to, 177–178
- overriding authentication, 552–555, 557–558
- reasons for using classes for generic views, 393–394
- replacing links with `get_absolute_url()`, 181–184
- upgrading website using GCBVs, 426–429

**DetailView**
- generic behavior in GCBVs, 385–388
- reasons for using classes for generic views, 393–394
- switching from custom GCBV to Django provided GCBV, 392

**Development**
- development-friendly optimization tools, 684
- preferences for controllers and views, 687
- running development server, 339, 735
- shortcuts for shorter development process, 135–136
- test-driven and behavior-driven, 753

**Development operations (devops), 726**
- `dev.py`
  - preparing deployment settings, 730–732
  - running development server, 735

**Diamond inheritance problem, 762–763**

**Dictionaries**, sitemap dictionary, 716

**DiNucci, Darcy, 189**

**Directories**
- blog project (start up organizer), 19–21
- contrib directory, 325–327
- creating CSS files in, 374
- for fixtures, 624
- Hello World page and, 24–25
- for templates, 78, 103

**Distributed denial-of-service (DDOS) attacks, 661**

**Django Contributed Library. See**
- Contributed Library (contrib)
- `django-admin` command-line tool, 14
- `django-admin.py` script, 14
- `django-toolbelt`, for deployment, 728–729
- DNS (domain name system), 43
- DOS (denial-of-service) attacks, 661
- DRY (Don’t Repeat Yourself) principle
  - advantages of class-based views, 158–159
  - avoiding duplication of attribute use, 419
  - building links and, 169
  - creating URL paths for navigation menu, 176–177
  - Django following, 11
  - migrations and, 279
  - shortening code and, 136
  - working with static content, 375

**DTL (Django Template Language)**
- controlling markup with templates, 318
- creating templates, 73
- custom template tags, 687
- security features, 770
- as template engine, 78
- template short-circuiting, 663–664

**Dummy cache, 684**

**Dynamic websites**
- building, 6–8
- overview of, 5–6
- Dynos (workers), Heroku, 726–727

---

**E**

**Edit pages (change view)**
- admin app providing, 581
- configuring admin app, 584–588, 596–604

**Email**
- configuring, 300–301
- message framework indicating status of, 306
- sending, 304
- sending and logging using mixins, 518–527
- sending using programming method, 303
- sending with Postmark, 743–745

**EmptyPage exception, in pagination, 336–337, 342–343**

**Encapsulation**
- clean apps and, 78
- refactoring code to adhere to standard, 143
- utility of app encapsulation, 27–28
Index

Encryption, 770
Engine, DTL as template engine for Django, 78
\texttt{error()}, for storing messages, 532
Escaping text, HTML rules for, 80
Espresso tool, for CSS code, 777
Event handling, with signals, 649
Exceptions (errors)
Bad Request (HTTP 400), 727–728, 758
creating error pages, 727–728
displaying form errors, 213–216
Forbidden (HTTP 403), 483, 485–487, 727–728, 758
Internal Server Error (HTTP 500), 727–728, 741, 758
Invalid Query (HTTP 404), 132–135, 758
\texttt{log.error()} (522, 524)
\texttt{NoReverseMatch} exceptions, 174–175
pagination, 336–337
source code and versioning and, 325
validation techniques, 197–198
\texttt{execute()} method, overriding, 634
Explicit relative import, \	exttt{Tag} model and,
75–76

F
Feeds. See News feeds
Fields
adding relational, 40–42
adding \texttt{SlugField}, 289–293
admin app manipulating, 585–586
controlling behavior of, 42–45
creating \texttt{clean} method for \texttt{Tag} model,
198–201
for forms vs. for models, 191–192
generating IDs and labels for, 220–221
hidden, 447
limiting in queries, 672–673
looping over, 222
\texttt{Post} model, 33–34, 36–38
structuring data and, 318
\texttt{fieldsets} attribute, add and edit pages, 596
Files

caching template files, 680–682
creating CSS files in directories, 374
loading template files, 112
splitting into smaller modules, 308–310
Filters
building custom template filter, 688–689
controlling variable output with, 86–89
log filters, 454–455
\texttt{safe} filter, 364–365
types of controls in DTL, 318
using \texttt{date} template filter for
customizing output, 95–96
using \texttt{linebreaks} template filter for
formatting paragraphs, 97–99
using \texttt{urlize} template filter for
automatic linking, 96–97
Finite-state machines, 190
Fixes, source code and versioning and, 324
Fixture

data handling and, 622
overview of, 624–627
\texttt{flake8} tool, checking syntax with, 777
Flatpages app

anatomy of, 355
creating About page, 356–359
creating core app for data migration,
365–366
creating template for, 355–356
data migration for, 369–370
disabling middleware and switching back
to URL configuration, 362
displaying \texttt{FlatPage} objects via
middleware, 360–362
displaying \texttt{FlatPage} objects via URL
configuration, 359–360
Django functionality in, 326
enabling, 353–355
introduction to, 353
linking to \texttt{FlatPage} objects, 363
replacing flatpages with GCBVs, 398–399
security implications, 363–365
summary of, 370
FlyData logging service, 742
\texttt{fold} (reduce) tool, 675
\texttt{force_str()}, importing tools from Django,
633
Foreign keys

database relations, 34
one-to many relationships and, 475
unique identifiers, 766
Foreman’s server, running, 735–737
\texttt{Form} class, 302
format tag, HTML, 318
formatting
  linebreaks template filter for formatting paragraphs, 97–99
  for templates, 77–78
Forms
  auth app anatomy and, 545–546
  AuthenticationForm, 458
  building, 527–529
  contact form, 302–304
  creating user accounts, 517
  displaying create or update forms, 697–701
  displaying delete confirmation forms, 701
  flatpages form, 355
  hidden fields in, 447
  integrating forms and templates, 566–567
  interacting with data via, 318–319
  making relations optional on, 295–296
  password form, 503–506
  states, 235–236
  tag form, 211–213
  understanding expected behavior, 234–238
  URL pattern and view for interacting with, 304–306
  validating, 518
Forms, controlling using views
  adding URL pattern and hyperlink, 244–246
  creating NewsLink objects, 252–254
  creating Post objects, 249–250
  creating Startup objects, 251–252
  creating Tag objects, 238–244
  deleting NewsLink objects, 271–272
  deleting objects, 268–269
  deleting Post objects, 269–271
  deleting Startup objects, 273, 275–276
  deleting Tag objects, 273–275
  introduction to, 233–234
  modifying NewsLink objects, 261–264
  modifying Post objects, 257–261
  replacing tag_create() with TagCreate CBV, 246–249
  shortening organizer views, 254–256
  summary of, 276–277
  understanding expected behavior, 234–238
  updating links for TagUpdate and StartupUpdate, 267–268
  updating objects, 256–257, 264–265
  updating Startup objects, 266–267
  updating Tag objects, 265–266
Forms, for user input
  connecting TagForm to Tag model using inheritance, 201–203
  creating clean method for Tag model, 198–201
  creating PostForm, 206–208
  creating StartupForm and NewsLinkForm, 208–210
  creating TagForm, 190–192
  demonstrating use of TagForm in shell, 193–197
  implementing save() method with TagForm, 192–193
  introduction to, 189
  as state machines, 190
  summary of, 210
  understanding ModelForm validation, 203–205
  updating objects using ModelForm, 205–206
  validation techniques, 197–198
Forms, templates for displaying
  bound form values, 216–217
  contact form, 306–308
  creating for StartupForm, NewsLinkForm, and PostForm, 227–229
  creating for Tag objects, 211–213
  deleting Tag objects, 226–227
  displaying errors in tag_form.html, 213–216
  DRY principles, 218
  generating field IDs and labels, 220–221
  inheritance of, 229–231
  introduction to, 211
  looping over form fields, 222
  printing forms directly, 222–224
  replacing loops and conditions with variables, 218–220
  summary of, 229–231
  template variables making TagForm template dynamic, 213
  updating Tag objects, 224–225
Frameworks
in building websites, 6–8
CSS, 377–378
generating project structure, 14
message framework, 306
Python web framework, 8–11
translation framework, 328–329
Front-end programming, of webpages, 5–6
Function views (FV). See also views (Django)
adding URL pattern, 130–132
building for Tag detail, 128–130
comparing to class-based views, 125,
157–158, 160–162
converting to class-based views, 337–338,
384–385
creating tags and, 238–244
data returned by, 74
greeting() in Hello World page, 25
Invalid Query (HTTP 404) error,
132–135
limitations as generic view, 393
replacing tag_create() with
TagCreate CBV, 246–249
starting projects and, 753
using slug argument with, 150
webpages and, 316
Functions, Python, 25

G
GCBVs (generic class-based views)
adding behaviors for blog post archive,
408, 414–415
adding behaviors with, 401
allow_future attribute, 495
anticipating behavior overrides, 388–392
building generic object creation pages,
394–395
building object detail pages, 384
cleaning up URL patterns, 544–545
comparing with class-based views, 156,
394
converting function views to, 384–385
generic behavior, 385–388
introduction to, 383
linking to blog post archive, 414
multiple inheritance in Python and, 762
optimizing views with related content,
676–678
overriding methods, 400
reasons for using classes, 393–394
redirection with RedirectView GCBV,
398
replacing CBVs, 395–397
replacing flatpages, 398–399
review of, 418
selecting dates for yearly archive of blog
posts, 402–403
starting projects and, 753
summary of, 416
switching from custom GCBV to Django
provided GCBV, 392
templates for blog post archives, 404–408,
410–413
views and URL configurations for blog
post archive, 403–404, 408–410
when to use/when not to use, 400–401
GCBVs (generic class-based views),
upgrading website with
applying DateDetailView to
PostDetail, 426–429
automating Startup selection, 444–449
benefits of class mixins, 429–432
extending pagination behavior, 423–425
fixing URL patterns in NewsLink,
438–444
generating pagination links, 419–423
introduction to, 417
making class mixin generic, 432–438
pagination of StartupList, 421–422
pagination of TagList, 422
review of GCBVs, 418
setting template suffix for UpdateView
GCBV, 419
summary of, 449–450
Generic relations
cumenttypes app and, 475–476
permissions and, 476
Generic templates
applying in Tag list, 106–108
building app-generic templates, 108–109
building site-wide generic template,
104–106
informing Django of site-wide templates,
103–104
Generic views, reasons for using classes for, 393–394
GET method. See HTTP GET
get() method
    accessing page query with, 342
    creating view for modifying Post objects, 259–261
    using with TagCreate class, 247
get_absolute_url()
    canonical URLs and, 173
    inversion of control and, 192
    replacing detail page links with, 181–184
    returning URL path of new Tag object, 240–241
get_object_or_404(), 136–137
get_user_model(), 459
getattr(), 367
Git repository
    deployment tools, 728
    for project and example code, 4
Gondor deployment service, 726
Google Webmaster Central Blog, 333
greeting(), Hello World page, 25
Group model
    in auth app, 476
    many-to many relationships and, 480
Groups, permissions and, 480–482

H
handle(), createuser command and,
    638–643
Hardcoding, navigation links, 176
has_perm method, permissions, 476
Hashes
    creating user accounts and, 517
    resetting passwords and, 507
HEAD, HTTP, 759
Hello World page
    advantages of models and views over controller, 27
    creating helloworld app, 24–25
    data for, 25
    displaying, 26–27
    introduction to, 23–24
    removing helloworld app, 27–28
    summary of, 29
    template benefits, 74–76
    URL for, 25–26
Help, displaying help text in forms, 219
Heroku
    adding backing services (add-ons), 741
    caching with memcache, 745–748
    CLI, 728
    as deployment service, 726
    deployment to, 738–741
    logging service of, 741–743
    preparing for deployment, 726–727
Historical (frozen) model
    accessing for sites app, 367
    migration system and, 282–286
Hollywood principle. See Inversion of control
Homepage
    redirecting, 186–187
    redirecting with URL configurations, 163–164
    redirecting with views, 164–166
    homepage() view, restructuring, 148–149
Hosting services, deployment options, 725
href attribute, HTML, 375
HTML (HyperText Markup Language).
    See also Forms
    building links, 169, 178–181
    building navigation menu, 175–176
    coding templates, 78–79
    Django supported output formats, 10
    escaping text, 80
    form tag, 318
    hidden fields, 447
    href attribute, 375
    HTTP transferring files written in,
        757–758
    inheritance for design consistency, 102
    web browsers and, 76
    in webpage display, 6
    writing templates in HTML5, 77
HTTP (HyperText Transfer Protocol)
    adding state to, 456
    creating URL for new webpage, 74–75
    primer, 757–759
    Python web framework and, 8
    submitting data to websites, 235–238,
        242–243
    view requirements, 125–126
    website basics and, 4
    HTTP 200 (Request Valid), 758
HTTP 400 (Bad Request), 727–728, 758
HTTP 403 (Forbidden), 483, 485–487, 727–728, 758
HTTP 404 (Invalid Query), 132–135, 758
HTTP 500 (Internal Server Error), 727–728, 741, 758
HTTP GET
  deleting objects and, 268–269
  form states and, 243, 246
  function of, 759
  interacting with contact forms, 305
  submitting data to websites, 235–238
  updating objects and, 256
  view states and, 249
HTTP HEAD, 759
HTTP OPTIONS, 759
HTTP POST
  deleting objects and, 268–269
  form states and, 243
  function of, 759
  interacting with contact form, 306
  submitting data to websites, 235–238
  view states and, 249
  webpages for updating objects, 256
HTTP PUT, 759
HTTP requests
  middleware modifying HTTP objects, 360–361
  overview of, 757
  Python web framework and, 8
  receiving, 313
  step-by-step code examination of views and URL configuration, 126–128
  tools for intervening in control flow, 319
  view requirements, 125–126, 316
  website basics and, 4
HTTP responses
  middleware modifying HTTP objects, 360–361
  overview of, 757
  Python web framework and, 8
  response codes, 758
  step-by-step code examination of views and URL configuration, 126–128
  tools for intervening in control flow, 319
  view requirements, 125–126, 316
  website basics and, 4
Hyperlinks. See Links, between webpages
Identification, authentication and, 451
IDEs, choosing, 777
IDs, generating field IDs and labels, 220–221
if conditions, performing conditional value checks, 218–219
Images
  adding logo to websites, 377
  adding to websites, 373
Importing
  app models, 356, 580
  decorators, 494–495
  explicit relative import, 75–76
  tools from Django, 633–634
  view class, 156
include(), creating hierarchy of URL configurations, 145–148
Indexing, sitemaps and, 707
Inheritance
  connecting TagForm to Tag model, 201–203
  for design consistency, 102
  feeds from superclass, 712
  fields from ModelForm, 221
  forms from ModelForm, 301, 319
  of group permissions, 480
  manager from Manager class, 622
  manager ORM from models.Model, 71
  multiple inheritance in Python, 762–763
  querysets from QuerySet class, 623
  review of GCBVs and, 418
  TagForm from ModelForm, 240
  of templates, 229–231
User model from AbstractUser, 559–560
UserManager from
  BaseUserManager, 561–562
  .init.py file, in packages, 365
Input validation, 197–198
Input/output (I/O), optimization and, 662
INSTALLED_APPS
  connecting to new apps in
    settings.py, 19–21, 650
  preparing deployment settings, 730
  staticfiles contributed app, 374
Installing
  Django, 776
  Python, 776–777
Instance variables, 158. See also Attributes
Instantiation, of Django models, 57
Inversion of control
  frameworks use of, 7
  generating project structure, 14
  model methods and, 192
  redirection and, 241
  step-by-step code examination of views and URL configuration, 128
  tools for intervening in control flow, 319
I/O (input/output), optimization and, 662
IP addresses, locating webpages, 5
is_bound attribute, form attributes, 194
is_valid attribute
  form attributes, 194–195
  validation techniques, 197–198, 239
isort tools, for best practices, 777

J
JavaScript, in display of webpages, 6
JSON
  Django supported output formats, 10
  fixtures and, 622
  serialization of data and, 624
  web browsers and, 76

K
Keywords, signal handlers, 656–657

L
label attribute, AppConfig object, 650
Labels, generating field IDs and labels, 220–221
Lexes, compilation and, 773
Libraries (codebases)
  Admin library. See Admin app contrib. See Contributed Library (contrib)
  frameworks compared with, 7
  logging library, 452
  Python, 10
linebreaks template filter, for formatting paragraphs, 97–99
Links
  adding pagination links, 379
  to blog post archive, 414
  to CSS stylesheets, 375, 377
  displaying template links conditionally, 496–497
  to FlatPage objects, 363
  generating pagination links, 419–423
  to object creation forms, 379
  urlize template filter for automatic linking, 96–97
Links, between webpages
  adding for TagDelete and StartupDelete, 275
  adding to feeds, 710–711
  adding to form view, 244–246
  adding URL pattern and, 244–246
  building navigation menu, 175–176
  canonical URLs and, 173
  creating detail page links, 178–181, 184–186
  creating URL paths for navigation menu, 176–177
  creating using URL query, 341
  list pages to detail pages, 177–178
  NoReverseMatch exceptions, 174–175
  redirecting homepages and, 186–187
  replacing detail page links with get_absolute_url(), 181–184
  reversing regular expressions patterns, 171–172
  reversing URL patterns, 170–171
  summary of, 187–188
  updating for TagUpdate and StartupUpdate, 267–268
Linode deployment services, 726
List pages
  adding information to, 589–592
  applying generic template to Tag list, 106–108
  building for blog posts, 151–152
  building for startup page, 149
  building template for blog post app, 101–102
  building template for startup objects, 99
  building template for tag objects, 90–93
  configuring admin app, 593–596
  configuring for Post objects, 581–584
creating according to function, 675
iterating through QuerySet to print, 81–86
linking to, 177–178
ordering by date, 48
providing for models, 581
ListView
adding information to, 589–592
generating pagination links, 421–422
loader class, using templates in, 112
Local memory cache. See also Caching
replacing with dummy cache, 684
types of caches, 682
log_mail.error(), 522, 524
Logentries, logging service and, 742
Loggers, 453–454
Logging
configuring, 452–455
mixins for logging email, 518–527
setting log level to critical, 525–526
logging library, 452
Logging service, adding backing services, 741–743
Logic
decoupling from presentation, 76
using template tags to add, 81
@login_required decorator
creating custom decorators, 486, 488, 490
disabling user accounts, 514
Login/logout
auth app anatomy and, 546
auth app settings, 458
signals for, 652–655
views for making login and logout pages, 465–471
Logos
adding to websites, 377
style for, 378
Logout. See Login/logout
LogRecord objects
creating, 452
filtering, 454–455
Long-term support (LTS), versions and, 324
Lookups, managers and querysets using, 61
Loops
over form fields, 222
replacing with variables, 218–220
 Loose coupling
actor model and, 660
in Django, 649
LTS (long-term support), versions and, 324

M

mail.managers(), 304
makemigrations command
adding author to blog posts, 572
creating migrations, 50–53
ensuring unique identifier for NewsLink, 294–295
migration system and, 282
passing name argument, 285
Management commands
checking production settings (check command), 738
creating tags (createtag), 628–630
creating users (createuser/createsuperuser), 630
handling data, 622
locating settings (Procfile command), 739
overview of, 627–628
manage.py
creating apps, 17–19
creating migrations, 50–53
displaying Hello World page, 26–27
invoking shell, 112
viewing project installation, 15–17
Manager class, 622, 678–679
Managers
adding methods to, 621
configuring email settings, 301
creating for custom user, 561–563
custom managers and querysets, 622–624
interacting with databases, 58–63
iterating through QuerySet to print lists, 81–84
lookups, 61
mail.managers(), 304
for many-to-many relationships, 65–68
methods, 60–62
model managers returning QuerySet object, 116
Managers (continued)
optimizing Manager classes directly, 678–679
ORM inherited from models.Model, 71
Many-to-many relationships
adding relational fields to models, 41
customizing or adding information to relations, 55
field format, 52
forward and reverse relations, 656, 658–659
generic relations, 475–476
Group model, 480
managers for, 65–68
in relational database, 766–767
Markup languages. See also HTML (HyperText Markup Language)
controlling markup with templates, 318
web browsers and, 76
writing templates in, 77
Memcache, caching with, 745–748
Memcached Cloud, 745
Memcachier, 745–748
Memory, data in memory vs. data in database, 64–65
Message framework, 306
Messages, logging and, 453
Messages app
creating admin actions and, 616
displaying login/logout signals, 652–653
displaying message on login page, 517
error() for storing messages, 532
informing user of account activation status, 534–535
loose coupling, 649
storing messages, 326
Meta classes
attributes defined in, 52
controlling model behavior using, 47–49
Method resolution order, multiple inheritance and, 762
@method_decorator() creating custom decorators, 489–491
disabling user accounts, 514
Methods
adding to models, 45–47
anticipating behavior overrides, 388–392
for feeds, 709
HTTP GET and HTTP POST requests, 156
managers and querysets, 60–62
overriding, 400
permissions, 476
protecting HTTP methods, 484
Python methods vs. HTTP methods, 155
Middleware
 caching entire webpages, 683
disabling and switching back to URL configuration, 362
displaying FlatPage objects, 360–362
flatsites app, 355
preparing deployment settings, 730–731
sessions app, 456–457
tools for intervening in control flow, 319
view middleware, 361–362
migrate command, 16, 53–54
Migrations
adding author to blog posts, 572–573
adding slug to news link, 289–293
creating, 50–53
creating core app for data migration, 365–366
creating data migration for sites app, 365–369
for creating or modifying databases, 53–55
data migrations, 280
ensuring unique identifier for news link, 294–295
introduction to, 279–280
making relations optional on forms, 295–296
of news link data, 288
optimizing webpages and, 673–676
of post data, 287–288
schema migrations, 288
signals for, 652
of startup data, 285–287
steps in building websites, 299
summary of, 296–297
of swappable models, 565
of tag data, 280–284
understanding, 49–50
of User model, 568–572
Mixins. See Class mixins
Mobile apps, building, 7
Mobile frameworks, 7
Model, in MVC architecture
  advantages of Models and Views over controller, 27
  function of, 8–9
Model managers, ORM tool, 621
ModelForm
  blank field option, 295
  creating form class that inherits from, 301
  fields and inheritance, 221
  forms inheriting from, 319
  TagForm inheriting from, 240
  updating objects, 205–206
  validation, 203–205
Models (Django)
  accessing data, 34–35
  adding methods, 45–47
  adding relational fields, 40–42
  auth app and, 476, 545
  connecting with data through relations, 65–68
  controlling behavior using nested meta classes, 47–49
  controlling field behavior, 42–45
  creating migrations, 50–53
  creating or modifying databases using migration, 53–55
  creating PostForm model, 206–208
  creating StartupForm and NewsLinkForm models, 208–210
  data in memory vs. data in database, 64–65
  in Django core, 315
  extending User model, 558–561
  fields, 36–38, 191–192
  granting permissions to users, 478
  importing from sites app, 356
  instantiation of, 57
  interacting with databases via, 56–58
  introduction to, 31–32
  managers interacting with, 58–63
  migrations and, 49–50
  organizing data, 32–34, 36
  Post model for running start up organizer, 38–40
  Profile model, 550–552
  project specifications, 750
  reasons for using databases, 32
  registering, 580
  steps in building websites, 299
  string capitalization criteria and, 68–71
  structuring and communicating with databases, 314
  structuring and storing data, 317–318
  summary of, 71–72
  swappable, 564–565
  updating objects, 205
  User model, 458–463
  validating, 203–205
MTV (Model-Template-View) architecture, 313, 316
Multiple inheritance, Python, 762–763
MVC (Model-View-Controller) architecture
  applying to Hello World page, 23–24
  custom template tags as example of Django not adhering to, 687, 706
  decoupling presentation from logic, 76
  developer preferences for controllers and views, 687
  Django models and, 210
  limitations of using with Django, 316
  MTV architecture compared with, 313
  for structure of Django projects, 8–10
  URL configuration and view interaction, 121–122
MySQL, for deploying public websites in cloud, 729

N
name attribute, AppConfig object, 650
Natural data, 625
natural_key() method, of dumping data, 626–627
Navigation, pagination for. See Pagination
Navigation menu
  building in HTML, 175–176
  creating URL paths for, 176–177
Nested meta classes, controlling model behavior, 47–49
@never_cache() decorator, 684
News feeds
  introduction to, 707
  RSS and Atom formats, 707–714
  summary of, 724
News link app
  adding slug to NewsLink object, 289–293
  attributes of Post model, 33–34, 36
  automating Startup selection in NewsLink forms, 444–449
  completing Post model, 39
  creating NewsLink objects in a view, 252–254
  creating NewsLinkForm, 210
  creating templates for NewsLinkForm, 227–229
  creating view for modifying NewsLink objects, 261–264
  deleting NewsLink objects, 271–272
  ensuring unique identifier for NewsLink objects, 294–295
  fixing URL patterns, 438–444
  importing and registering app models, 580
  overriding methods in NewsLinkDelete, 400
  overriding with news article, 48–49
  schema migrations, 288
  setting template suffix for UpdateView GCBV, 419

NewsLink objects
  adding slug to, 289–293
  creating, 252–254
  deleting, 271–272
  ensuring unique identifier for, 294–295
  modifying, 261–264
  schema migrations, 288

NewsLinkDelete, 400

NewsLinkForm
  automating Startup selection in, 444–449
  creating, 210
  templates for, 227–229

NewsLinkUpdate, 419

NoReverseMatch exceptions, 174–175

Normalization of data, benefits of databases, 32

null option, compared with blank field, 295

obj argument, checking object-level permissions, 482–483

Object detail pages
  building, 384
  creating links on, 184–186

Object-oriented programming (OOP)
  advantages of class-based views, 158
  origin in actor model, 660
  Python as object-oriented language, 761

Object-relational mapper. See ORM (object-relational mapper)

Objects. See also by individual types
  building generic object creation pages, 394–395
  buttons for linking to object creation forms, 379
  identifying by primary key, 261
  linking to, 363
  managers working with, 58–59
  permissions, 482–483
  Python, 761

One-to-many relationships
  adding relational fields to models, 40–41
  creating with foreign keys, 766
  foreign keys and, 475

One-to-one relations, 550

OOP. See Object-oriented programming (OOP)

OPTIONS, HTTP, 759
  ordering attribute
    ordering lists by date, 48
    use on UserAdmin list page, 594

Organizer app
  adding styles to, 379–381
  adding URL pattern for tag detail function view, 131–132
  admin app and, 579
  in app registry, 282
  app-generic templates, 109–112
  building app-generic templates, 108–109
  building generic object detail pages, 384
  building startup detail page, 150
  building startup list page, 149
  building tag detail function view, 128–130
  building template for list of Tag objects, 90–93
  class mixins in, 254–256
  creating detail page links, 178–181
  creating feeds, 711–714
creating hierarchy of URL configurations, 145–148
creating links on object detail pages, 184–186
creating webpage with, 74–76
importing and registering app models, 580
Invalid Query (HTTP 404) error, 132–135
linking list pages to detail pages, 177–178
manage.py for creating, 17–19
migration, 280
replacing CBVs with GCBVs, 395–397
restructuring homepage() view, 148–149
reversing URL patterns, 170–171
shortening code with get_object_or_404(), 136–137
shortening code with render(), 139–143
shortening code with render_to_response(), 137–139
shortening organizer views, 254–256
organizer/urls.py file, 308–310
ORM (object-relational mapper)
communicating with databases, 37
connecting with data through relations, 65–68
core features at heart of Django, 621
in database communication, 190
directory location of code for, 325
identifying NewsLink objects, 261
interacting with databases via models, 56
manager object and, 58
manager ORM inherited from models.Model, 71
protecting against database injection attacks, 769
Output
date template filter for customizing, 95–96
I/O optimization, 662
Overrides
anticipating behavior overrides, 388–392
execute() method, 634
methods in NewsLinkDelete, 400
Packages
_init_.py file in, 365
package management, 775
url package, 308–310
PageLinkMixin, generating pagination links, 419–423
Pagination
adding pagination links, 379
extending pagination behavior, 423–425
generating pagination links, 419–423
introduction to, 331
shell use in working with, 333–337
of StartupList, 337–345, 421–422
summary of, 351
of TagList, 345–351, 422
URL options: query vs. path, 332–333
Paginator object, 339
Papertrail logging service, 741–743
Paragraphs, formatting, 97–99
Parsing, compilation and, 773
Passwords
adding change password page, 604–612
auth app and, 546
changing, 462, 503–506
forms in auth app, 503
prompting developer for (getpass), 644–645
resetting, 502, 506–513
starting projects and, 752
User model and, 459–460
user profile and, 554
views and, 501–503
Pattern matching, regular expressions for, 771–772
Performance. See also Website optimization
global changes to, 680
speed and, 661
Permission model, 476, 478
@permission_required decorator, 486, 490
Permissions
adding author to blog posts, 575
admin app using, 582, 591
class mixins applied to blog post, 495–496
contenttypes and generic relations and, 473–476
Permissions (continued)
custom decorators and, 488–495
data migration and, 568–570
displaying future posts in template, 497–499
displaying template links conditionally, 496–497
granting to users, 478
groups in shell, 480–482
introduction to, 473
model in auth app, 476
object-level, 482–483
protecting views, 483–487
in shell, 476–480
signals for, 652
summary of, 500
User model, 559
Persistence of data, benefits of databases, 32
POST method. See HTTP POST
post() method
deleting NewsLink objects, 271–272
deleting Post objects, 269–271
modifying Post objects, 259–261
TagCreate class, 247–248
Post model, 183–184
Post objects
automatically assigning Tag objects to
instances of, 655–660
configuring list of, 581–584
creating in a view, 249–250
deleting, 269–271
migration of, 287–288
modifying, 257–261
Post sitemap, 715–718
PostAdmin
adding information to list view, 589–592
configuring add and edit pages, 584–588
configuring list of Post objects, 581–584
PostCreate, 249–250
PostDetail, 426–429
PostForm
creating, 206–208
creating Post objects in a view, 249–250
creating templates for, 227–229
PostGetMixin
benefits of, 429–432
making generic, 432–438
PostManager, 622–624
Postmark email service, 743–745
PostQuerySet, connecting PostManager to, 623
PostgreSQL
for deploying public websites in cloud, 729
selecting database when starting project, 753
Posts, blog. See also Blog app (start up organizer)
adding author to, 572–576
adding behaviors for indexing archive of, 414–415
adding behaviors for monthly archive, 408
adding behaviors for yearly archive, 401
adding information to list view, 589–592
applying permissions to, 495–496
archive sitemap, 720–723
automatically assigning Tag objects to
Post instances, 655–660
benefits of PostGetMixin, 429–432
building detail view for, 152–155
building page for listing, 151–152
building query for database, 154
configuring add and edit pages, 584–588
configuring list of Post objects, 581–584
contributors groups permissions, 481
creating custom managers, 622–624
creating Post model, 37
creating Post objects in a view, 249–250
creating PostForm, 206–208
creating templates for PostForm, 227–229
creating via migration, 51–53
DateDetailView applied to
PostDetail, 426–429
deleting Post objects, 269–271
displaying future posts, 497–499
displaying related posts, 690–697
importing decorator into, 494–495
inspecting internals of class-based views, 160–162
iterating through QuerySet to print list of
Post objects, 84–86
linking to monthly archive of, 414
listing latest posts, 701–706
migration of Post objects, 287–288
modifying admin controls for, 581
modifying Post objects, 257–261
organizing data, 33–34
overview of, 12
Post model, 183–184
Post sitemap, 715–718
running start up organizer, 38–40
selecting dates for yearly archive, 402–403
template for list of, 101–102
template for monthly archive of, 410–413
template for single post, 100–101
template for yearly archive of, 404–408
views and URL configurations for monthly archive, 408–410
views and URL configurations for yearly archive, 403–404
PostUpdate, attributes, 262–263
Prefetch object, queryset optimization, 670–672
prefetch_related() fetching relations, 669
optimizing views with related content, 676–678
prepopulated_field option, admin app, 585–586
Presentation, decoupling from logic, 76
Primary keys
assigning to rows, 765–766
automatically assigning Tag objects to Post instances, 658
automatically created for Django models, 52
database relations and, 34
identifying objects by, 261
resetting passwords and, 506–507
in sites app, 367–368
Printing forms, 222–224
Priority attribute, of webpages, 717–718
Privileges/roles, User model, 460
Procfile command, locating settings with, 739
Production settings, 730–735, 738
production.py file, 732–733
Profile model, 550–552
ProfileDetail view, 552–555
ProfileUpdate, 555–557
PublicProfileDetail, 557–558
Profiling
analysis of computer processes, 661
querysets, 667–670
software, 662
Projects
adding static content to, 376–377
apps and, 650
Projects, starting
building project, 752
introduction to, 749
optimization and, 753–754
REST APIs and, 754
selecting database, 753
selecting third-party apps, 751–752
specification, 749–751
starting with generic views, 753
testing, 753
User model, 752
using reverse methods, 753
Properties, caching properties for optimization, 664–665
psycopg2, deployment tools, 729
PublicProfileDetail, 557–558
published() method
Post objects, 622
QuerySet class, 623
PUT, HTTP, 759
PyCharm IDE, 777
Python
decorators, 761–762
greeting() in Hello World page, 25
installing, 776
libraries, 10
multiple inheritance, 762–763
overview of, 761
regular expressions, 772
using templates with Template, Context, and loader classes, 112
versions, 11–12
web framework, 8–11
PythonAnywhere deployment service, 726
PyYAML package, supporting YAML use, 624
Q

Queries. See also URL queries
building for blog post database, 154
Invalid Query (HTTP 404) error,
132–135, 758
limiting database queries, 663
limiting fields in, 672–673
pagination options, 332–333

Querysets
inheriting from QuerySet class, 623
interacting with databases, 56
iterating through to print list of Post
objects, 84–86
iterating through to print list of Startup
objects, 81–84
lookups, 61
managers returning, 623–624
methods, 60–63
model managers returning QuerySet
object, 116
optimization, 667–670
optimizing QuerySet classes directly,
678–679

R

Rackspace deployment service, 726
Raw data, compared with cleaned data, 192
@receiver() decorator, 654, 656
redirect(), 186–187, 240
Redirection
of homepage, 163–167, 186–187
inversion of control and, 241
post-authentication, 471–472
with RedirectView GCBV, 398
of webpage to new Tag object, 240
RedirectView GCBV, 398
reduce(fold) tool, 675
RegexURLPattern objects, 143. See also
URL patterns
@register decorator, 581, 593
Registration, app models, 580

Regular expressions
building detail view for blog posts, 153
matching characters, 130–131
matching URL patterns to URL
configuration, 145–146
overview of, 771–772

reversing character set patterns, 171–172
reversing URL patterns, 170–171
in URL patterns, 317

Relational databases, 765–767

Relations
adding relational fields to models, 40–42
connecting with data through r, 65–68
fetching with prefetch-related(), 669
forward and reverse, 656–657
generic, 475–476
making optional on forms, 295–296
primary and foreign key, 34

Relative import, Tag model and, 75

Render
ccontent using templates in Python classes,
112–116
templates, 318
render() rendering templates, 242–244
shortcut functions for simplifying code,
139–143
render_to_response(), 137–139
RequestContext class, 140–143
@require_authenticated_permission
decorator, 493
requirements.txt file, for deployment,
728

REST APIs, 754

reverse() DRY principle and, 179
redirecting homepages and, 187
Reverse methods, starting projects and, 753

Reversing URL patterns
DRY principle and, 179
NoReverseMatch exceptions, 174–175
overview of, 170–171
redirecting homepages and, 187
reversing regular expressions patterns
with character sets, 171–172

Roles/privileges, User model, 460
Rows, table, 765
RSS feeds, 707–714
RunPython operation
data migration and, 285
data migration in flatpages app, 370
data migration in sites app, 368
data migration of User model, 571
runserver command
invoking server with, 16
running development server, 339

S
safe filter, security implications of flatpages app, 364–365
save()
  implementing with TagForm, 192–193
saving information to databases, 303
Scaling websites, 661
Schema, database, 279
Schema migrations
  adding slug to NewsLink, 289–293
  ensuring unique identifier for NewsLink, 294–295
  making relations optional on forms, 295–296
  newslink data, 288
  overview of, 288
SchemaEditor, communicating with databases, 282
Scripts, migrations as, 279
Search engines, 707, 717
Secret keys, creating, 736
Secure Hash Algorithm (SHA), 507
Secure Sockets Layer (SSL), 770
Security
  authentication. See Authentication basics of, 769–770
  of data migration, 568–569
  flatpages app and, 363–365
  permissions. See Permissions
select_related(), optimizing views with related content, 676–678
_send_mail(), mixin for logging email, 521–524, 529
Serialization of data
  difficulty of working with, 625–626
  fixtures supporting, 622
Servers
  configuring email setting for contact app, 301
  defined, 4
  in generation of webpages, 6
  HTTP, 757
  invoking with runserver command, 16
  running development server, 339,
    735–737
  running foreman’s servers, 735–737
  viewing project installation via testing server, 15–17
Sessions, authentication and, 451, 456–457
Sessions app, 456–457
Settings
  checking production, 738
  preparing deployment, 730–735
settings.py file
  Blog project (start up organizer), 19–21
  Hello World page, 24–25
  listing project apps, 650
  preparing deployment settings, 730–735
  removing helloworld app, 27–28
SHA (Secure Hash Algorithm), 507
Shell
  data management and, 189
  demonstrating use of TagForm in, 193–197
  groups in, 480–482
  pagination with, 333–337
  permissions in, 476–480
  queryset optimization, 667–670
  templates in, 112–116
Short-circuiting, template short-circuiting for optimization, 663–664
Signals
  automatically assigning Tag objects to Post instances, 655–660
  introduction to, 649–650
  for login/logout actions, 652–655
  loose coupling, 649
  summary of, 660
Single-page applications, 6
Site model, 356
Sitemaps
  for basic webpages, 723–724
  for blog post, 715–718
  for blog post archive, 720–723
  overview of, 715
  for startup, 720
  summary of, 724
  for tag, 718–719
Sites app
  auth app reliance on, 457
  creating data migration for, 365–369
Sites app (continued)
   enabling, 354
   importing models from, 356
   purpose of, 354–355
Site-wide content, Django allowing, 625
Site-wide templates
   applying in Tag list, 106–108
   building, 104–106
   informing Django of, 103–104
Skeleton CSS framework, 378–381
Slicing, splitting tag list webpage, 333–334
Slugs
   adding to NewsLink, 261, 289–293
   building detail view for blog posts, 153
   controlling field behavior, 42
   creating Profile model, 551
   identifying data in URL, 35
   using with function view, 150
SMTP, options for interacting with email services, 744–745
Software, profiling, 662
Source code, availability of, 323–325
Specification of project, 749–751
Speed. See Website optimization
Spreadsheets, organization of relational databases and, 765
SQLite database
   creating via migration, 53–55
   creating with manage.py, 49
   for small projects, 729
SSL (Secure Sockets Layer), 770
Staging servers, 735
Start up organizer. See Blog app (start up organizer)
startproject subcommand, automating Django behavior, 14
Startup data, migration of, 285–287
Startup model, importing and registering, 580
Startup objects
   add_startup_data and remove_startup_data, 286–287
   automating selection in NewsLink forms, 444–449
   building startup detail page, 150
   building startup list page, 149
   building template for, 93–95
   converting function views to class-based views, 384–385
   creating, 251–252, 296
   deleting, 273, 275–276
   pagination of startup list, 337–345, 421–422
   printing list from queryset, 81–84
   replacing detail page links with get_absolute_url(), 183
   template for list of, 99
   updating, 264–267
Startup sitemap, 720
startup_detail page, building, 150
startup_list page, building, 149
StartupCreate, 251–252
StartupDelete, 275
StartupDetail
   anticipating behavior overrides, 388–392
   generic behavior in GCBVs, 385–388
StartupForm
   creating, 208–210
   creating Startup objects, 251–252
   templates for, 227–229
   updating Startup objects, 266–267
StartupList, pagination of, 337–345, 421–422
StartupUpdate
   setting template suffix for UpdateView GCBV, 419
   updating links, 267–268
   updating objects, 264–265
State
   adding to HTTP, 456
   form states, 235–236
State machines, forms as, 190
Static app (staticfiles contributed app), 373, 732–733
Static content
   adding to apps, 374–376
   adding to projects, 376–377
   integrating CSS content, 377–381
   introduction to, 373
   preparing deployment settings, 732–733
   summary of, 381
Static websites, compared with dynamic, 5
str(), inversion of control and, 192
String method, adding to Django models, 45–47
@stringfilter decorator, 688–689
Strings
capitalization criteria, 68–71
force_str(), 633
Styles/stylesheets. See CSS (Cascading Style Sheets)
suorganizer app. See Blog app (start up organizer)
Superusers
create_superuser(), 562–563
createsuperuser command, 645–647
permissions and, 477
User model, 462
Swappable models, 564–565

Tables
mapping Python classes to database tables, 64
in relational database, 765–767
Tag model
creating clean method for, 198–201
importing and registering, 580
project specifications, 750–751
relative import and, 75–76
replacing detail page links with
get_absolute_url(), 181–182
TagForm connecting to via inheritance, 201–203
Tag objects
adding content using template variables, 80–81
adding logic using template tags, 81
automatically assigning to Post instances, 655–660
building, 78
coding in HTML, 78–79
controlling variable output, 86–89
converting function views to class-based views, 384–385
createtag command, 628–630
creating, 238–244
demonstrating use of TagForm in shell, 193–197
deleting, 273–275
determining save() method with TagForm, 192–193
iterating through QuerySet to print list of, 81–86
optimizing webpages and, 673–676
pagination of tag list webpage, 345–351
splitting tag list webpage, 333–334
template for creating, 211–213
template for creating list of, 90–93
templates for deleting, 226–227
templates for updating, 224–225
updating, 264–266
Tag sitemap, 718–719
tag_create() function view
creating Tag object, 238–244
replacing with TagCreate CBV, 246–249
tag_detail() function view
adding URL pattern for, 130–132
coding, 128–130
TagCreate CBV
creating custom decorators, 491–492
creating Tag object, 238–244
replacing tag_create(), 246–249
TagDelete
adding link for, 275
deleting Tag objects, 273–275
TagDetail
anticipating behavior overrides, 388–392
creating detail page links, 178–181
generic behavior in GCBVs, 385–388
TagForm
bound form values in tag_form.html, 216–217
connecting Tag model using inheritance, 201–203
creating, 190–192
creating Tag object, 240
demonstrating use in shell, 193–197
displaying form errors in tag_form.html, 213–216
DRY principles in tag_form.html, 218
implementing save() method with, 192–193
looping over form fields, 222
TagForm (continued)
  template variables making TagForm
template dynamic, 213
  templates, 211
  updating Tag objects, 265–266
TagList
  applying generic template to, 106–108
  pagination of, 422
Tags, types of controls in DTL, 318
TagUpdate
  setting template suffix for UpdateView
    GCBV, 419
    updating links, 267–268
    updating objects, 264–265
TDD (test-driven development), 753
Template class, using templates in Python
  classes, 112–115
Template tags
  applying generic template to Tag list, 106–108
  building Tag objects, 78
  building template for list of Tag objects, 90–93
  function and subclass of Node class
    required, 701–702
  overview of, 118
  syntax of, 690
Template variables, making TagForm
template dynamic, 213
TemplateResponse
  auth’s view using, 514–516
  comparing with HttpResponse, 540
Templates
  adding content using template variables, 80–81
  adding stylesheets to, 375–376
  advantages of, 74–76
  app-generic, 109–112
  applying generic template, 106–108
  for blog post archive, 404–408, 410–413
  building for list of Tag objects, 90–93
  building for single Startup object, 93–95
  building for Tag objects, 78
  building generic templates, 108–109
  building navigation menu, 175–176
  building site-wide generic template, 104–106
  caching template files, 680–681
  caching template variables, 665–667
  changing passwords, 503
  choosing format, engine, and location
    for, 77–78
  coding in HTML, 78–79
  controlling markup with, 318
  controlling output with template filters, 86–89
  creating accounts, 535–538
  creating for contact app, 302
  creating for flatpages app, 355–356
  creating URL paths for navigation menu, 176
  date template filter for customizing
    output, 95–96
  displaying future posts, 497–499
  displaying template links conditionally, 496–497
  in Django core, 315
  generating URLs, 170
  informing Django of site-wide templates, 103–104
  inheritance for design consistency, 102
  integrating with CSS, 377–381
  integrating with forms, 566–567
  introduction to, 73
  linebreaks template filter for
    formatting paragraphs, 97–99
  for list of blog posts, 101–102
  for list of startup objects, 99
  loose coupling, 649
  printing list of Post objects, 81–86
  for profile update, 556
  for profile views, 554–555
  for public profiles, 558
Python with Template, Context, and loader classes, 112
resetting passwords, 502, 506, 509
setting template suffix for UpdateView
GCBV, 419
in shell, 112–116
for single blog post, 100–101
steps in building websites, 299
styling, 379
summary of, 118–119
template short-circuiting for
optimization, 663–664
understanding use and goals of, 76
urlize template filter for automatic
linking, 96–97
in views, 116–118
Templates, for displaying forms
bound form values in tag_form.html, 216–217
contact form, 306–308
creating for StartupForm,
NewsLinkForm, and PostForm,
227–229
creating for Tag objects, 211–213
deleting Tag objects, 226–227
displaying form errors in
tag_form.html, 213–216
DRY principles in tag_form.html, 218
generating field IDs and labels, 220–221
inheritance of, 229–231
introduction to, 211
looping over form fields, 222
printing forms directly, 222–224
replacing loops and conditions with
variables, 218–220
summary of, 229–231
template variables making TagForm
template dynamic, 213
updating Tag objects, 224–225
TemplateView GCBV
for account creation and confirmation,
517
replacing flatpages with GCBVs, 398–399
Test–driven development (TDD), 753
Tests
syntax and testing tools, 777
testing projects, 753
Text
displaying help text in forms, 219
HTML rules for escaping, 80
time, HTML tag, 101
TLS (Transport Layer Security)
authentication using TLS certificates, 456
security features, 770
token_generator(), 520
Tokens
calling tag as token, 704–705
compilation and, 773
CSRF, 505
resetting passwords and, 506–508
as unique identifier, 212, 456
Tools
deployment tools, 728–729
syntax and testing tools, 777
Transactional emails, 743
Translation framework, 328–329
Transport Layer Security (TLS)
authentication using TLS certificates, 456
security features, 770
truncatewords filter, 102
Try.except block, in createuser
interactive code, 643–644
U
ugettext(), 328
ugettext_lazy(), 328
Unbound forms, displaying, 242
Unicode, 772
Uniform Resource Identifiers (URIs), 123–124
Unique identifiers
ensuring for NewsLink, 294–295
foreign keys, 766
tokens as, 212
URLs (Uniform Resource Locators), 34–35
Update
displaying update forms, 697–701
of links for TagUpdate and
StartupUpdate, 267–268
of objects using ModelForm, 205
overview of, 256–257
ProfileUpdate, 555–557
of Startup objects, 265–267
Update (continued)
of Tag objects, 224–225, 264–265
view for modifying NewsLink objects, 261–264
view for modifying Post objects, 257–261
UpdateView GCBV
ProfileUpdate, 555–557
setting template suffix for, 419
URLs (Uniform Resource Identifiers), 123–124
url()
creating hierarchy of URL configurations, 145–148
creating URL pattern, 131–132
instantiation of URL patterns, 143
URL configuration
adding URL pattern for tag.detail function view, 130–132
applying to other pages of site, 148
for blog post archive, 403–404, 408–410
building startup detail page, 150
building startup list page, 149
changing passwords, 503–506
connecting to blog app, 151
creating hierarchy of, 145–148
disabling middleware and switching back to, 362
displaying FlatPage objects, 359–360
for feeds, 710
for Hello World page, 29
importance of order of URL patterns, 143–145
interacting with contact form, 304–306
as list of URL patterns, 317
overview of, 122–125
pagination of Tag List webpage, 346
purpose of, 122
redirecting homepage with, 163–164
step-by-step code examination of use of, 126–128
url package, 308–310
URL path
pagination of Tag List webpage, 345–351
pagination options, 332–333
URL patterns
adding, 244–246
adding for tag.detail function view, 131–132
adding to form view, 244–246
cleaning up, 544–545
creating for change password page, 606
creating sitemaps for basic pages, 723–724
in Django core, 315
for feeds, 713–714
fixing news links, 438–444
importance of order of, 143–145
interacting with contact form, 304–306
loading into URL configuration, 126
NoReverseMatch exceptions, 174–175
pagination of Tag List webpage, 345–351
redirecting homepage, 163–164
reversing, 170–171
splitting urls.py file into smaller modules, 308–310
step-by-step code examination of views and URL configuration, 127–128
steps in building websites, 299
using URL pattern dictionary, 157–158
views for making login and logout pages, 468
webpages and, 317
URL queries
creating links, 341
pagination options, 332–333
post-authentication redirection, 471–472
url template
creating detail page links, 178–181
creating URL paths for navigation menu, 176–177
DRY principle and, 179
overview of, 170–171
urlize template filter, for automatic linking, 96–97
URLs (Uniform Resource Locators)
canonical URLs, 173
connecting to views. See URL configuration
creating for new webpage, 74–75
creating URL paths for navigation menu, 176–177
for Hello World page, 25–26
for identifying webpages, 4–5
Invalid Query (HTTP 404) error, 132–135
Index

max_length parameter, 42–43
pagination options: query vs. path, 332–333
project specifications, 750–751
reversing URL patterns, 170–171
as subset of URIs, 123–124
uniquely identifiable, 34–35
using urlize template filter for automatic linking, 96–97
urls.py file, 308–310
User accounts
    creating, 517
disabling, 513–516
forms in auth app, 503
resending account activation, 538–544
templates for creating, 535–538
views for creating and activating, 529–535
User app
    creating, 464–465
custom AppConfig for, 655
importing decorator from, 494
User input. See Forms
User model
    configuring admin app, 593
    connecting UserManager to, 563
defining, 476
extending, 558–561
groups in shell, 480–482
has_perm and has_perms methods, 482
migration, 568–572
overview of, 458–463
permissions, 476–480
relation to Profile model, 549–550
starting projects and, 752
swapping out older versions with new custom version, 564–566
UserAdmin
    adding change password page, 604–612
    adding profile to, 613–615
    configuring add and edit pages, 596–604
    configuring list page, 593–596
    creating admin actions, 616–618
UserCreationForm
    building forms, 527–529
    configuring add and edit pages, 600–604
    creating Profile model, 551–552
    integrating forms and templates, 566–567
    views for creating and activating accounts, 529–535
UserManager
    auth app, 561
    connecting to User model, 563
Usernames, User model, 459
Users
    changing passwords, 504
    granting permissions to, 478
    listing permissions of, 482
    profiles. See Profiles
Validation
    check_unique() and clean_value(), 636–637, 640–641
    cleaned data, 192, 303
    form, 319, 518
    input validation for forms, 197–198
    ModelForm validation, 203–205
Variables
    adding template content using, 80–81
    caching, 665–667
    controlling output with template filters, 86–89
    replacing loops and conditions with, 218–220
    template variables making TagForm template dynamic, 213
Versions
    controlling, 776
    conventions for numbering, 323–325
View, in MVC architecture
    advantages of Models and Views over controller, 27
    developer preferences, 687
    function of, 8–9
View class. See also CBVs (class-based views)
    importing, 156
    overview of, 155
    webpages and, 316
View functions. See views (Django)
View middleware, 361–362
views (Django)
    applying to webpages, 148
    auth app and, 458, 546
views (continued)
for blog post archive, 403–404
building detail view for blog posts, 152–155
building Tag detail function view, 128–130
class-based. See CBVs (class-based views)
comparing Django view and with view in MVC architecture, 9
comparing with function views and class-based views, 125
connecting URL to. See URL configuration
core features at heart of Django, 621
creating and activating accounts, 529–535
creating user accounts, 517
disabling accounts, 513–516
in Django core, 315
greeting() in Hello World page, 25, 29
implementing ProfileDetail view, 552–555
interacting with contact form, 304–306
listing blog posts, 151–152
login() and logout(), 458
making login and logout pages, 465–471
for monthly archive of blog posts, 408–410
optimizing, 676–678
overview of, 125–126
for password interaction, 501–503
permissions protecting, 483–487
purpose of, 122
reasons for using classes for generic views, 393–394
redirecting homepage with, 164–166
replacing CBVs with GCBVs in blog app, 397
replacing CBVs with GCBVs in organizer app, 395–397
resetting passwords, 506
restructuring homepage() view, 148–149
shortcuts for shorter development process, 135–136
shortening code with get_object_or_404(), 136–137
shortening code with render(), 139–143
shortening code with render_to_response(), 137–139
step-by-step code examination of use of, 126–128
steps in building websites, 299
template use in, 116–118
URL patterns referencing, 317
webpages and, 316
views (Django), controlling forms
adding URL pattern and hyperlink, 244–246
creating NewsLink objects in, 252–254
creating Post objects in, 249–250
deleting Startup objects in, 271–272
deleting objects, 268–269
deleting Post objects, 269–271
deleting Startup objects, 273, 275–276
deleting Tag objects, 273–275
implementing webpage for creating tags, 238–244
introduction to, 233–234
modifying NewsLink objects, 261–264
modifying Post objects, 257–261
replacing tag_create() with TagCreate CBV, 246–249
shortening organizer views, 254–256
summary of, 276–277
understanding expected behavior, 234–238
updating links for TagUpdate and StartupUpdate, 267–268
updating objects, 256–257, 264–265
updating Startup objects, 266–267
updating Tag objects, 265–266
vim, choosing IDE, 777
Virtual environments, installation and, 776
Virtual private servers (VPNs), deployment options, 725
VPNs (virtual private servers), deployment options, 725

W
W3C (World Wide Web Consortium), 35
Web 2.0, 189
Web browsers
- contacting servers by scheme and network location, 332
- in generation of webpages, 6
- for identifying webpages, 4–5

markup languages, 76
Web framework, Python, 8–11
Web Server Gateway Interfaces (WSGIs), 726
Webdesign app, 326
Webpage links. See Links, between webpages
Webpages. See also by individual types
- Back-end and front-end programming and, 5–6
caching, 682–684
- creating, updating, deleting content, 577
defined, 4
generating, 317
- sitemaps for basic pages, 723–724
- sitemaps for dynamic pages, 715–723
- URL patterns and configurations, 317
views, 316
Webpages, creating
- adding URL patterns, 130–132
- advantages of class-based views, 158–159
- applying views and URL configurations to, 148
building detail view for blog posts, 152–155
building list view for blog posts, 151–152
building startup detail page, 150
building startup list page, 149
building Tag detail webpage, 128
class-based views in, 155–157
coding tag_detail() function view, 128–130
comparing class-based views to function views, 157–158
connecting URL configuration to app, 151
creating hierarchy of URL configurations, 145–148
examining code for views and URL configurations, 126–128
importance of order of URL patterns, 143–145
inspecting internals of class-based views, 160–162
introduction to, 121–122
Invalid Query (HTTP 404) error, 132–135
redirecting homepage with URL configurations, 163–164
redirecting homepage with views, 164–166
refactoring code to adhere to app encapsulation standard, 143
restructuring homepage() view, 148–149
shortening code with
- get_object_or_404(), 136–137
- shortening code with render(), 139–143
shortening code with render_to_response(), 137–139
summary of, 166–167
URL configurations, 122–125
view shortcuts for shorter development process, 135–136
views, 125–126
Webpages, for creating form objects
adding URL pattern and hyperlink, 244–246
creating NewsLink objects in a view, 252–254
creating Post objects in a view, 249–250
creating Startup objects in a view, 251–252
overview of, 233–234, 238–244
replacing tag_create() with TagCreate CBV, 246–249
shortening organizer views, 254–256
Webpages, for deleting form objects
deleting NewsLink objects, 271–272
deleting Post objects, 269–271
deleting Startup objects, 273, 275–276
deleting Tag objects, 273–275
overview of, 268–269
Webpages, for updating form objects
overview of, 256–257
updating links for TagUpdate and StartupUpdate, 267–268
updating Startup objects, 264–267
updating Tag objects, 264–266
view for modifying NewsLink objects, 261–264
view for modifying Post objects, 257–261
Website optimization
   caching entire webpages, 682–684
   caching properties and, 664–665
   caching template files, 680–681
   caching template variables, 665–667
   development and, 684
   DTL (Django Template Language) and,
   663–664
   global changes to performance, 680
   internal changes to database behavior,
   679–680
   introduction to, 661
   limiting database queries, 663
   limiting fields in queries, 672–673
   migrations and, 673–676
   optimizing admin pages, 679
   optimizing Manager and QuerySet
   classes directly, 678–679
   optimizing querysets, 667–670
   optimizing views with related content,
   676–678
   Prefetch object and, 670–672
   profiling software, 662
   starting projects and, 753–754
   summary of, 685

Websites
   basics of, 4–5
   building dynamic, 6–8
   creating, updating, deleting content,
   577–578
   creating stylesheet for, 376
   defined, 4
   deploying. See Deployment
   Django core and, 313–315
dynamic, 5–6
   iterative approach to building, 299
   Python web framework, 8–11
   scaling, 661
typically combining HTML, CSS,
   JavaScript, and media, 373
   upgrading using GCBVs. See GCBVs
   (generic class-based views), upgrading
   website with
whitenoise
   deployment tools, 729
   preparing deployment settings, 733
   Whitespace, formatting paragraphs, 97–99
   Widgets, input fields for HTML forms, 319
   Workers (dynos), Heroku, 726–727
   Workflow, understanding migrations, 49
   World Wide Web Consortium (W3C), 35
   @wraps() decorator, 489
   WSGIs (Web Server Gateway Interfaces),
   726

XML
   Django supported output formats, 10
   serialization of data and, 624

YAML
   fixtures supporting serialization of data,
   622
   serialization of data and, 624