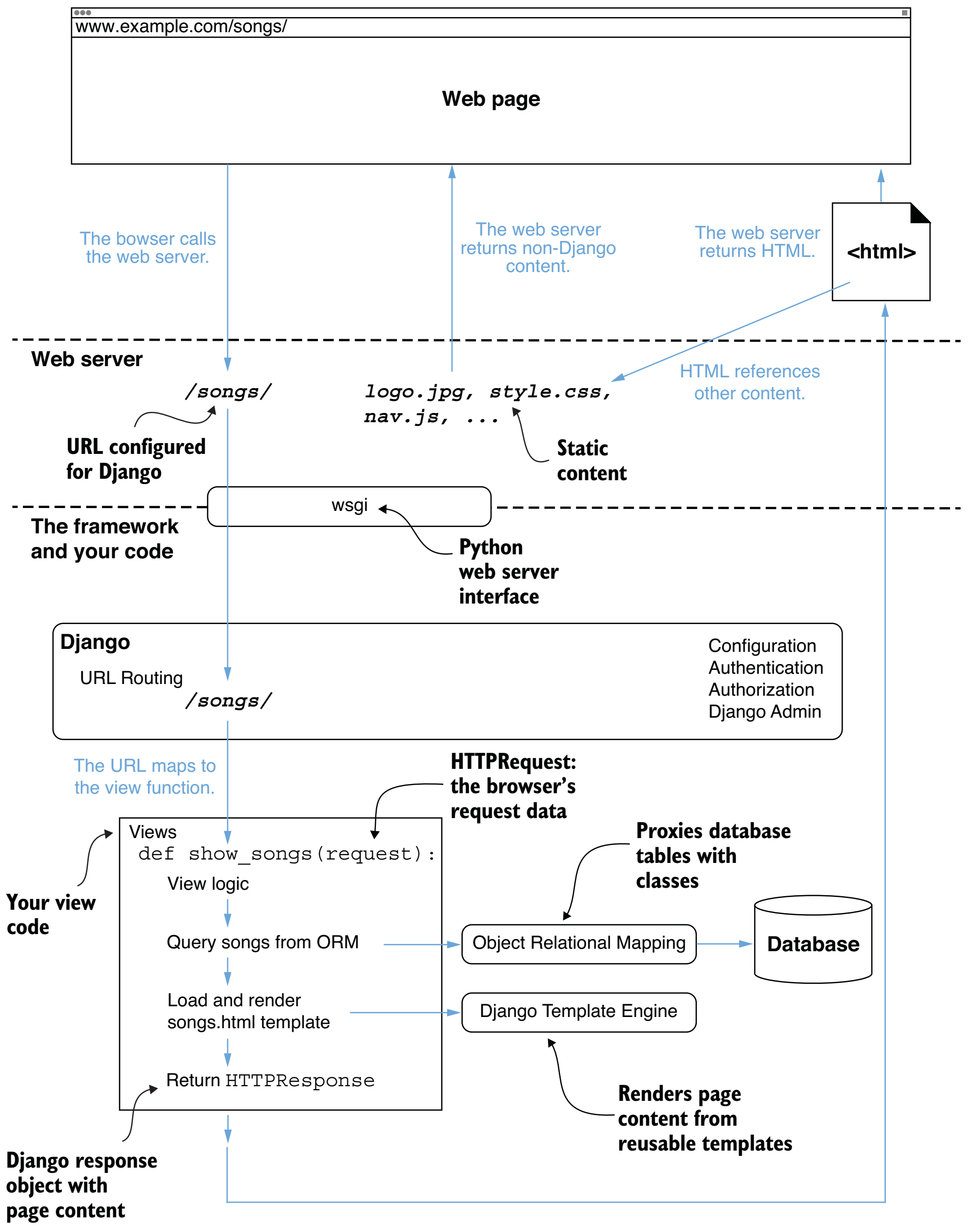
Django IN ACTION

Christopher Trudeau

Foreword by Michael Kennedy

MANNING

Rendering a web page in the Django framework



www.example.com/songs/
Web page
The bowser calls the web server.
The web server returns non-Django content.
The web server returns HTML.
<html>
Web server
/songs/
logo.jpg, style.css, nav.js, ...
HTML references other content.
URL configured for Django
Static content
wsgi
The framework and your code
Python web server interface
Django
URL Routing
/songs/
Configuration
Authentication
Authorization
Django Admin
The URL maps to the view function.
HTTPRequest: the browser's request data
Views
def show_songs(request):
View logic
Query songs from ORM
Load and render songs.html template
Return HTTPResponse
Your view code
Proxies database tables with classes
Object Relational Mapping
Database
Django Template Engine
Renders page content from reusable templates
Django response object with page content

Django in Action

CHRISTOPHER TRUDEAU
FOREWORD BY MICHAEL KENNEDY

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor:	Connor O'Brien
Technical editor:	Asif Saif Uddin
Review editor:	Kishor Rit
Production editor:	Deirdre Hiam
Copy editor:	Christian Berk
Proofreader:	Jason Everett
Technical proofreader:	Ninoslav Cerkez
Typesetter and cover designer:	Marija Tudor

ISBN 9781633438163
Printed in the United States of America

For Michelina

brief contents

PART 1 DJANGO ESSENTIALS 1

1 ■ Django unfolds 3
2 ■ Your first Django site 21
3 ■ Templates 40
4 ■ Django ORM 65
5 ■ Django Admin 105

PART 2 DJANGO BUILDING BLOCKS 127

6 ■ User management 129
7 ■ Forms, user data, static files, and uploads 152
8 ■ Testing your project 191
9 ■ Management commands 213
10 ■ Migration 232

PART 3 DJANGO PROJECTS 251

11 ■ APIs 253
12 ■ Making your pages more dynamic with HTMX 283
13 ■ Django power tools 301
14 ■ Where to go next 315

contents

foreword *xi*
preface *xiii*
acknowledgments *xv*
about this book *xvii*
about the author *xxi*
about the cover illustration *xxii*

PART 1 DJANGO ESSENTIALS 1

1 *Django unfolds* *3*

1.1 Django's parts 4

Mapping URLs, Django views, and the MVC model *8*
Databases: Dynamic sites need data *11* ▪ *Structuring HTML for reuse* *13* ▪ *Multiuser sites* *15*

1.2 What can you do with Django? 15

Server-side, single-page, and mixed-page applications *16*
When and where you should use it *17* ▪ *Potential projects* *18*
The RiffMates project *19*

2 *Your first Django site* *21*

2.1 The RiffMates project 21
2.2 Creating a Django project 22
2.3 Projects vs. apps 29
2.4 Your first Django app 32

2.5 Your first Django view 33
2.6 Registering a route 36
2.7 Visiting your view 37
2.8 Exercises 38

3 *Templates 40*

3.1 Where to use templates 40
3.2 Context objects and the parts of a Template 41
3.3 Django shell 43
3.4 Rendering a Template 43
3.5 Common tags and filters 46

Conditional blocks 46 ▪ Looping blocks 47 ▪ Comment blocks 49 ▪ Verbatim blocks 49 ▪ Referencing URLs 50 Common filters 51

3.6 Using render() in a view 51
3.7 Escaping special characters 56
3.8 Composing templates out of blocks 59

Writing a base template file 60 ▪ Including an HTML snippet 62

3.9 Exercises 63

4 *Django ORM 65*

4.1 Interacting with a database 65
4.2 ORM Models 68
4.3 SQLite and dbshell 71
4.4 Model queries 73

Using .all() and .filter() to fetch QuerySet results 75 Field look-ups 77

4.5 Modifying your data 79
4.6 Querying models in views 81

Using URL arguments 81 ▪ Listing pages 86 ▪ Using query parameters for pagination 87

4.7 Model relations 89

One-to-many relationships 90 ▪ Many-to-many relationships 94

4.8 Model fields 97

Popular cross-field field options 97 ▪ Popular fields 98

4.9 Fixtures 99

4.10 Danger zone, or know thy SQL 103
4.11 Exercises 103

5 *Django Admin 105*
5.1 Touring the Django Admin 105
Creating an admin user 107 ▪ The default Django Admin 108 Adding your own models 109
5.2 Customizing a listing page 113
Sorting and searching 114 ▪ Filtering 115 ▪ New in Django 5: Facets 119
5.3 Cross-linking between related objects 119
5.4 Model Meta properties 122
Sort order 122 ▪ Indexes 122 ▪ Uniqueness 123 Object names 124
5.5 Exercises 124

PART 2 DJANGO BUILDING BLOCKS 127

6 *User management 129*
6.1 Multiuser websites 129
6.2 Storing user data 131
Choosing a user account mechanism 131 ▪ Adding a user profile 134
6.3 Restricting access to views 137
6.4 Authorization in views 139
Handling logouts 141
6.5 Using signals to automatically create a user profile 142
6.6 Password management 145
Password reset page 146 ▪ Password reset email 147 New password form 149
6.7 Exercises 150

7 *Forms, user data, static files, and uploads 152*
7.1 Web forms 153
Handling GET and POST in views 155 ▪ CSRF 160 Beautifying web forms 160 ▪ New in Django 5: Reusable field group templates 161

7.2 Django ModelForm 162

Validating fields in a model 162 ▪ A ModelAdmin for SeekingAd using Truncate 165 ▪ Writing a ModelForm 166 Using SeekingAdForm in a view 169 ▪ Editing existing data with forms 172

7.3 Serving static files 175

7.4 Uploads 178

Configuring your project for uploads 178 ▪ Storing and referencing uploaded files 180 ▪ File upload forms and views 183 ▪ Restricted file downloads 188

7.5 Exercises 188

8 *Testing your project 191*

8.1 Automated testing 191

The scope of tests 192 ▪ Automated testing in Python 194

8.2 Your first Django test 194

8.3 Testing with the database 197

8.4 Expecting errors 201

8.5 Authenticating and posting data 202

8.6 Testing with file uploads 206

8.7 More testing techniques 207

LiveServerTestCase for Selenium 208 ▪ Third-party helpers 208

8.8 Measuring tests and code quality 209

8.9 Exercises 210

9 *Management commands 213*

9.1 Management commands in practice 213

Built-in management commands 214 ▪ Changing a password 216 ▪ Configuration 217 ▪ Flushing the database 217 ▪ SQL commands 218

9.2 Writing your own management command 219

9.3 Handling arguments 222

Built-in arguments 227

9.4 Testing, and using STDOUT and STDERR 228

9.5 Exercises 229

10 *Migration 232*

10.1 Migration scripts in detail 232

An initial migration 234 ▪ Changing a model 236

10.2 Migration with existing data 238

10.3 Squashing a migration 243

10.4 More migration choices 245

10.5 Exercises 248

PART 3 DJANGO PROJECTS 251

11 *APIs 253*

11.1 Why use an API? 253

CRUD and REST 255 ▪ Versioning an API 258

11.2 Building an API with Django Ninja 258

Serializing objects 261 ▪ Handling arguments 264
Filtering your listings 268 ▪ Nesting data 271
Authenticating your API 272 ▪ Creating objects with the API 274 ▪ Updating and deleting with the API 277

11.3 Other API libraries 279

11.4 Exercises 281

12 *Making your pages more dynamic with HTMX 283*

12.1 Dynamic web pages 283

12.2 HTMX attributes 284

12.3 Lazy loading 286

12.4 Search-as-you-type with infinite scroll 288

12.5 Click to edit 295

12.6 Exercises 299

13 *Django power tools 301*

13.1 Django Debug Toolbar 302

13.2 Tool libraries 304

Django Extensions 305 ▪ Django Awl 305

13.3 Look and feel 307

Django Grappelli 307 ▪ Crispy forms 307 ▪ Favicons 309

13.4 Feature flags 310
13.5 Static sites 311

14 *Where to go next 315*
14.1 Within Django 315
14.2 Configuration 317
14.3 Dealing with data 318
14.4 Web tools 319
14.5 Asynchronous actions 320
14.6 A wide world 321

appendix A Installation and setup 323
appendix B Django in a production environment 330
appendix C Django template tag and filter reference 347
appendix D Django ORM field reference 357
appendix E Sample exercise solutions 364

index 401

foreword

Welcome to *Django in Action*! I'm sure you're excited to begin your Django journey. First, though, let me give you a little background on my friend, Christopher Trudeau, who has put a huge amount of effort into writing this book for you.

Christopher has a passion for communicating technical details, especially around Python and Django. Christopher has been on my podcast, *Talk Python to Me*, where his appearances have ranked in the top 10% of episodes, spanning almost 10 years of interviews. He frequently co-hosts the *Real Python* podcast. Christopher has also written and recorded many courses (much of them focusing on Django), and he has worked in such industries as cloud computing, gaming, travel, and finance, to name a few. In short, he doesn't just talk the talk—he has experience to back it up.

With this book, you hold the key to the fascinating world of Python web development. There are many types of applications you could choose to build: desktop GUI apps, mobile apps, embedded apps, and more. But web applications, the kind you build with Django, are special. They are immediately accessible to everyone, everywhere. There is no approval process, nor are there any gatekeepers or developer fees. You have an idea. You figure out how to build it with Python. You publish it to the web, and you're on your way.

So why Django? Django is one of the "big three" web frameworks in the Python space, which together represent the vast majority of adoptions. The people behind Django have spent years building their community.

Only Django, amongst the web frameworks, has the Django Software Foundation, a 501(c)(3) nonprofit foundation, to ensure the community remains strong and provides direction for the project. Only Django has outreach programs, such as Django Girls, a nonprofit foundation that organizes free Python and Django workshops for women interested in coding, and Djangonauts, a community mentorship program that helps people become contributors to Django.

So what you will learn in this book is how to build web apps with Django. But, perhaps unknowingly, you'll also be dipping your foot into this rich and supportive community behind Django. It's quite special, and there are few places like it.

I know you'll enjoy your journey into Django, both the technology and the community. And Christopher will be an excellent guide to lead you. Build something amazing and have a great time along the way!

—MICHAEL KENNEDY
FOUNDER, TALK PYTHON

preface

I was in university when the World Wide Web first came to be, and I remember using the Mosaic browser in its early form. When I entered grad school, we were already using the web to write content for students, including both static HTML and early CGI. I graduated from single Perl scripts to working with Java, eventually working with large teams of developers, writing scaled systems for the banking and telecom sectors.

In 2008, I was managing development teams at a company that got acquired, and during the transition, I had a lot of time on my hands. I decided to pick up a new language: Python. I used it to solve a couple of smaller problems we had at work and pretty much have never looked back. My next company was a start-up, building web tools. I looked around in the Python ecosystem for a web framework and landed on Django. Even back in its 1.1 days, Django was a complete system. It included everything we needed to build websites, which allowed my team and me to concentrate on the code specific to our clients.

Django's core is based on mapping a URL to some code, running the code that generates a page and returning that page as a response. The page can be built using Django's template engine, meaning HTML can be composed like code, removing the need for repetition. Most websites require some storage mechanism, and Django's object relational mapping (ORM) abstracts away the database, making it easier to query and manipulate data in Python.

This alone would be enough to recommend Django to build websites, but it contains much more. Out of the box, Django is built for multiuser websites, meaning it includes mechanisms for authentication, authorization, and user management. It comes with tools to help write automated tests, both at the unit level and with hooks for tools like Selenium for doing cross-site integration testing.

It doesn't stop there. Django is built using a pluggable architecture, which makes it easier for you to build content, but it also means there are plenty of people who've written tools you can take advantage of. The Django Packages site currently lists over 5,000 add-ons you can take advantage of. Need to write an API for your site? Django Ninja and the Django REST Framework have your back. Need a CMS? Wagtail and Django CMS are ready for you. Leveraging all these tools allows you to concentrate on what is unique to your project, rather than having to re-invent the wheel.

Django has been around for a long time, but it is still growing and adapting. Whether you need to put together a quick website prototype or write something to scale, Django is a great choice.

Django's documentation is already excellent, so why write a book? Django is so comprehensive that it can be difficult to wrap your mind around all its pieces. This book is based around a single project, adding to its features as you progress. This gives you a better understanding of how all the pieces fit together as well as practical advice on how to build your own site. My intent isn't to replace the documentation but to act as a companion, hopefully helping you along your learning journey.

acknowledgments

You and I have something in common: I also read everything in a book, including the acknowledgments. As a reader, I'd always wondered just how much the author was expected to say—and how much was genuine. Now that I find myself writing my own acknowledgments section, I find myself at a loss: How do I properly express gratitude to all those deserving?

Cliche as it may be, I'll start with my wife. She had unwavering enthusiasm throughout this project and confidence when mine was waning. Her support means everything to me. I'm a very lucky man.

Although my parents think Python has something to do with slithering animals, I'm still grateful for everything they have provided me. Whether nature or nurture, they can take more credit than most.

Before starting on this journey, I was ignorant of the fact that the term "editor" actually refers to a collective. Michael Stephens is my procurement editor, which means he was responsible for challenging me with doing this in the first place. Without him, none of this would have got started. Connor O'Brien is my development editor; he's what you think of when you hear the word "editor." He was my first reader, my guide, and a very patient man. He agreed to sign on when things were rough, at the beginning, and I'm grateful he took the chance on me. Thank you, Connor; it wouldn't have gotten done without you. And to complete the triumvirate, is Christian Berk, my copy editor, making me sound far more literate than I am. He diligently replaced every "which" with "that" and vice versa—I never seem to get those right.

To fellow Manning author (that's fun to write) and Real Python alumni, Dane Hillard, thanks for lending me your setup, saving me days of technical work, and listening to my moaning. Your perspective was invaluable.

There are loads more people at Manning who helped out along the way: Deirdre Hiam, my project editor; Jason Everett, my page proofer; technical editor, Asif Saif Uddin; reviewing editor, Kishor Rit; and technical proofreader Ninoslav Cerkez. Thanks to all of you for helping to turn some random scribbling into a professional affair.

Finally, to my internal reviewers and those who participated in the Manning Early Release Program, I send a giant thank you: Alfredo Alessandrini, Ankit Anchlia, Carmelo San Giovanni, Gregor Zurowski, Henry Stamerjohann, Ian De La Cruz, James Matlock, Jason Li, Jeff Smith, Jelte Derksen, Jereme Allen, Jonathan Reeves, Laud Bentil, Mafinar Khan, Mike Baran, Nitendra Bhosle, Omar Ebrahim, Oren Zeev Ben Mordehai, Patrick Regan, Radhakrishna MV, Richard Meinsen, Richard Tobias, Rodney Weis, Rupa Lahiri, Shahnawaz Ali, Simone Sguazza, Sudeep Batra, Sudheer Kumar Reddy Gowrigari, and Walter Alexander Mata López. Your feedback along the way helped me make better content and find those parts that needed more work. I'm sure some problems remain, but I'll take the blame for that.

about this book

Django in Action was written to help you learn how to use Django to build multiuser websites. Throughout the book, you'll be building *RiffMates*, a musicians' classifieds site. You'll start by creating a basic site with a few simple pages and eventually progress to multiuser capabilities.

How this book is organized: A roadmap

The book is divided into three parts. Part 1 gives the basic structure of all Django projects, teaching you how to get started.

- Chapter 1 provides a background on how Django sites work and introduces you to the differences between writing with a framework and writing using a library.
- Chapter 2 gets you started writing code, showing you how to create your first project and build your first web page.
- Chapter 3 does a deep dive into the Django Template Engine, which allows you to compose and re-use HTML, like you would with object-oriented code.
- Chapter 4 is all about the object relational mapping (ORM), an abstraction to a database, giving you the ability to write Python classes to interact with database tables and queries.
- Chapter 5 shows you the Django Admin, an out-of-the-box web tool for administering ORM classes. This includes both those you write and those which are built-in, such as the user management classes.

Once through part 1, you'll have enough to build your own single-user site. Part 2 introduces you to more advanced features in Django, including multiuser sites, accepting user content, testing, and the database migration system.

- Chapter 6 is about user management. You learn how to add and control users as well as all the work involved in password resets.
- Chapter 7 shows you how to deal with content for your site: data submitted by users; files uploaded by users; and static content, such as images, CSS, and JavaScript files.
- Chapter 8 covers the testing tools that come with Django. Unit testing a website requires extra work, but Django provides mechanisms to ensure you can write good automated tests with a high degree of coverage.
- Chapter 9 describes the Django management command interface: command-line tools you use to manage your site. It also shows you how to write your own custom commands.
- Chapter 10 does a deep dive on how Django manages changes to your ORM classes through its migration system. When you add fields to your database abstraction, the migration system keeps the corresponding database tables in sync.

The chapters in part 3 are independent of each other and show you how to use third-party tools to add capabilities to your website. If you've covered parts 1 and 2, you have enough knowledge to cherry-pick the chapters of interest in part 3.

- Chapter 11 shows you how to add API capabilities to your website, allowing your users to access data through programmatic interfaces. This chapter concentrates on the third-party Django Ninja tool but also points you toward the Django REST Framework as well as tools for GraphQL.
- Chapter 12 introduces you to writing more dynamic web pages using HTMX. In this chapter, you learn how to do search-as-you-type, lazy loading, infinite scroll, and more.
- Chapter 13 covers a wide variety of third-party Django tools that help you write less code and be more productive when you're building and debugging your sites.
- Chapter 14 gives a brief listing of other useful features in Django as well as third-party libraries that allow you to build more functionality into your websites.

Django In Action also includes a deep set of appendices, covering how to get Python and Django environments going, considerations when putting your code in production, and two quick-reference guides for template tags and ORM fields.

About the code

Throughout the book, you'll be building RiffMates, a musicians' classified site. Since Django is a framework, most code snippets will not work on their own—you've got to put it into the context of a project. You'll be guided on how to do this along the way, but a version of the project is available at https://github.com/cltrudeau/django-in -action.

The sample code is divided into sections, which correspond to chapters, or parts of chapters. For example, `code/ch04a_bands` contains the first few examples from chapter 4, while `code/ch04d_relationships` contains all the changes done in chapter 4.

Every chapter includes exercises for you to test what you've learned, and the sample code contains answers. For example, `code/ch04e_exercises` contains all the code from chapter 4, including the answers to the exercises.

As RiffMates gets built while you go along, each subsequent directory includes code for everything before it. The `ch05a_admin` directory includes all the code, including all the exercise answers from chapters 2 through 4. Spoiler alert, if you go digging around in the directories for later chapters, you may come across some exercise answers. Sample code was processed using the Black formatter (https://pypi.org/project/black/), and as such, you may find cosmetic differences between it and code generated by the Django framework itself.

Source code and commands you need to run are represented in the book using a `fixed width font`. Code listings may include line-continuation markers (➥) for code that is too long to fit on a single line. Code annotations accompany most of the listings, highlighting important concepts.

Writing about a framework is a moving target. The code in this book was written and tested against Django 5.0. Django's version numbering system uses what is referred to as "loose semantic versioning," offering a new release every 8 months. Each major version number (the *5* in *5.0*) corresponds to three releases. For the *5* family, that's 5.0, 5.1, and 5.2. The *.2* release is a long-term support version, meaning that's where the bug fixes will go for a 28-month support period. A full list of release timings and support windows is available on the downloads page: https://www.djangoproject.com/download/.

Don't let major release numbers scare you; they may or may not correspond to breaking changes. When I started writing the book, I was coding against 4.2, and upgrading to 5.0 required no changes in the project. Between versions, features may get deprecated—see the release documentation for specifics if you're upgrading a project.

Django releases are tied to Python versions. I tested using Python 3.12, but as long as you're using a version that matches the Django release, you'll be just fine. I intentionally avoided any Python syntax changes only available in more-recent releases. A table showing which Python versions are supported with which versions of Django is available on the FAQ page: https://docs.djangoproject.com/en/dev/faq/install/#what-python-version-can-i-use-with-django. You can get executable snippets of code from the liveBook (online) version of this book at https://livebook.manning.com/book/django-in-action. The complete code for the examples in the book is available for download from the Manning website at https://www.manning.com/books/django-in-action, and from GitHub at https://github.com/cltrudeau/django-in-action.

liveBook discussion forum

Purchase of Django in Action includes free access to liveBook, Manning's online reading platform. Using liveBook's exclusive discussion features, you can attach comments to the book globally or to specific sections or paragraphs. It's a snap to make notes for yourself, ask and answer technical questions, and receive help from the author and other users. To access the forum, go to https://livebook.manning.com/book/django-in-action/discussion. You can also learn more about Manning's forums and the rules of conduct at https://livebook.manning.com/discussion.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

Other online resources

Django's documentation is great, and it's the best place to start to get more information: https://docs.djangoproject.com/. Note that the documentation is divided by release number. Visiting the main URL automatically points you at the most-recent release version. Additionally, the "dev" version of the docs is one step ahead of the release, being the target for the next version. This book uses the dev version of the URL when pointing at any documentation.

For example, the following three URLs all point to information about the `django-admin` command, but for the dev, 5.0, and 4.2 versions, respectively:

- https://docs.djangoproject.com/en/dev/ref/django-admin/
- https://docs.djangoproject.com/en/5.0/ref/django-admin/
- https://docs.djangoproject.com/en/4.2/ref/django-admin/

The bottom-right corner of any Django documentation page has a Documentation Version widget that allows you to view the same page but for another release. Above the documentation widget, there is also a Language widget, showing the available translations for any given release.

The Django project's home page is also a great place to learn about what is going on in the Django world:

- *Django's News page*—https://www.djangoproject.com/weblog/
- *Django's Community page*—https://www.djangoproject.com/community/

There are also plenty of other sites that report on the Django ecosystem:

- Django News (https://django-news.com/) is a weekly email newsletter.
- Django Chat (https://djangochat.com/) is a biweekly podcast from two of the core developers.

For inspiration on what to build or to learn about the many packages out there, see

- *Django Projects*—https://builtwithdjango.com/projects/
- *Django Packages*—https://djangopackages.org/

about the author

CHRISTOPHER TRUDEAU is a fractional-CTO, who helps companies with both their tech stack and the processes their development teams use to build things. He is a co-host of the *Real Python* podcast, he is author of over 50 online Python and Django courses, and he has taught over 4,000 students about tech and technical processes.

Christopher has been writing code for over 40 years in a variety of languages and has been building websites using Django since its 1.1 days. In his spare time, he dabbles in photography, reads everything he can get his hands on, cycles the streets of his Toronto home, wrestles Komodo dragons, and enjoys adding lies in his bio.

about the cover illustration

The figure on the cover of *Django in Action*, titled "Le Créole," is taken from a book by Louis Curmer published in 1841.

In those days, it was easy to identify where people lived and what their trade or station in life was just by their dress. Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional culture centuries ago, brought back to life by pictures from collections such as this one.

Part 1

Django essentials

The first part of the book is all about getting you going with Django, giving you the tools you need to write a website. You'll learn how coding with a framework is different from writing against a library, how to create a Django project, how to use the Django Template Engine to compose and re-use HTML, how to interact with databases using Django's object relational mapping (ORM), and how to take advantage of the built-in Django Admin tool. When you're finished with this part of the book, you'll have written the first part of the RiffMates project and have enough knowledge to write Django-based websites.

1 Django unfolds

This chapter covers

- What the Django Framework is and why to use it
- The actions Django performs when you enter a URL in your browser
- Server-side rendering vs single-page applications
- The kinds of projects you can complete with Django

You've written a brilliant Python script that runs on your local machine, and now you want other people to be able to use it. You can use magical packaging tools and send your program to your users, but then, you're still stuck with the challenge of whether they have Python installed. Python does not come by default on many computing platforms, and you have the added problem of making sure the Python version your script requires is installed on your user's machine.

Alternatively, you can turn your Python program into a web application. In this case, your user's interface becomes a web browser, and those are installed everywhere. Your users no longer even need to know what Python is; they simply point their browser at the right URL and can use your software. To do this, your program needs to be adapted to run on a web server. This requires a bit of work, but using a single environment also has a distinct advantage: you are in full control over what version of Python gets run.

If you already have some experience with Python, a great answer to building web applications is Django. Django is a third-party Python framework that lets you write code that runs on a web server. Using Django, you write Python code, called a *view*, that is tied to a URL. When your users visit the URL, the Django view runs and returns results to the user's browser. Django does a lot more than that though—it includes tools for doing the following:

- Routing and managing URLs
- Encapsulating what code gets run per page visit
- Reading and writing to databases
- Composing HTML output based on reusable chunks
- Writing multiuser-enabled sites
- Managing user authentication and authorization to your site
- Web-based administrative tools that manage all these features

Django started out as a tool for writing newspaper articles at the *Lawrence Journal-World* in Lawrence, Kansas. Rather than having each article be a single file on a web server, the newspaper wanted a way to reuse pieces of HTML. You don't want to write the newspaper's banner in every file—you want to compose it like you would with code. You also probably don't want your reporters writing HTML; it isn't their expertise.

Django evolved as both a coding framework and a series of tools to manage web content. Among other things, the framework provides the ability to compose HTML out of reusable pieces solving the banner problem, and one of the tools it includes is a web-based interface for the creation of content. Reporters can write articles without worrying about HTML. Django's tools are built on top of the framework, giving you the power to customize them to suit your needs.

The robustness of the Django framework has led to widespread industry adoption. Reddit, YouTube, *The Onion*, Pinterest, Netflix, Dropbox, and Spotify are just a few of the organizations that use Django. Sites like these need scale, and Django has proven itself. Whether you want to build something small or hope to become a massive site, Django can help you. It is a mature platform with a vibrant community, focused on maintaining and evolving a well-tested, secure framework. To top it all off, the original developers at the *Journal-World* made it open source, meaning it is freely available for you to both use and modify for your needs.

1.1 Django's parts

Let's start with a high-level view of what makes up Django, knowing that each of these topics is covered in detail later in the book. One way to understand Django is to think about it in three parts:

1. A mapper between URLs and view code
2. An abstraction for interacting with a database
3. A templating system to manage your HTML-like code

Of course, Django is way more than just three parts, but all the other parts are built using these three concepts. Django includes tools like a web interface for modifying content in the database and the code needed to write multiuser sites. Each of these is built on those same three ideas. This means once you've learned the key concepts, you can incorporate the tools into your site and customize them using the same knowledge you use to write your web application.

The first of the three ideas is about how code gets invoked when a user visits a web page. When a browser hits a URL, a web server is responsible for replying. Django isn't a web server; it is a framework called by a web server. The web server is configured to map certain URLs to certain content, and in the case of Django, some of the URLs get passed to the framework. This handoff is completed through a generic Python web interface called the *Web Server Gateway Interface* (WSGI).

When you visit a Django URL, the web server calls the framework through WSGI, at which point Django is responsible for generating the page response. The first core part of Django determines what code to call based on the URL. This is done through a mapping in your Django project. The mapping is between URLs and response handlers, known as *views*. A view is a function or class that generates content, which Django then parcels as a response to the web browser. You'll write your first view in chapter 2, shortly after you've created your first Django project.

The complete request flow from browser to response is shown in figure 1.1. It starts with the browser interacting with the web server, the web server calling Django through WSGI, Django mapping the visited URL to a view, and the view formulating a response.

The mapping between URLs and views is accomplished through a list data structure inside a Python file. Each mapping gets defined through the construction of a `path` object, which contains the URL to map and a reference to a view function. Mappings can be given names, so you can reference them elsewhere in your code; you use the name in your code, or in your HTML, and Django replaces it for you. This way, changing a URL only needs to be done in one place.

Visiting a URL causes Django to loop through all its mappings and look for a match. If a match is found, the corresponding view gets invoked. If no match is found, Django shows a 404 error page.

The vast majority of code you write for a Django project is the actual view code. Views return a response object that encapsulates the information to send back to the browser. The most common of these objects is `HttpResponse`, which returns text over HTTP, the body of which is typically HTML.

Inside your view, you determine what your visitor sees on the web page. This may involve interacting with a database. Your database might contain the inventory in your warehouse, a list of the books in your library, or the names of users who can access

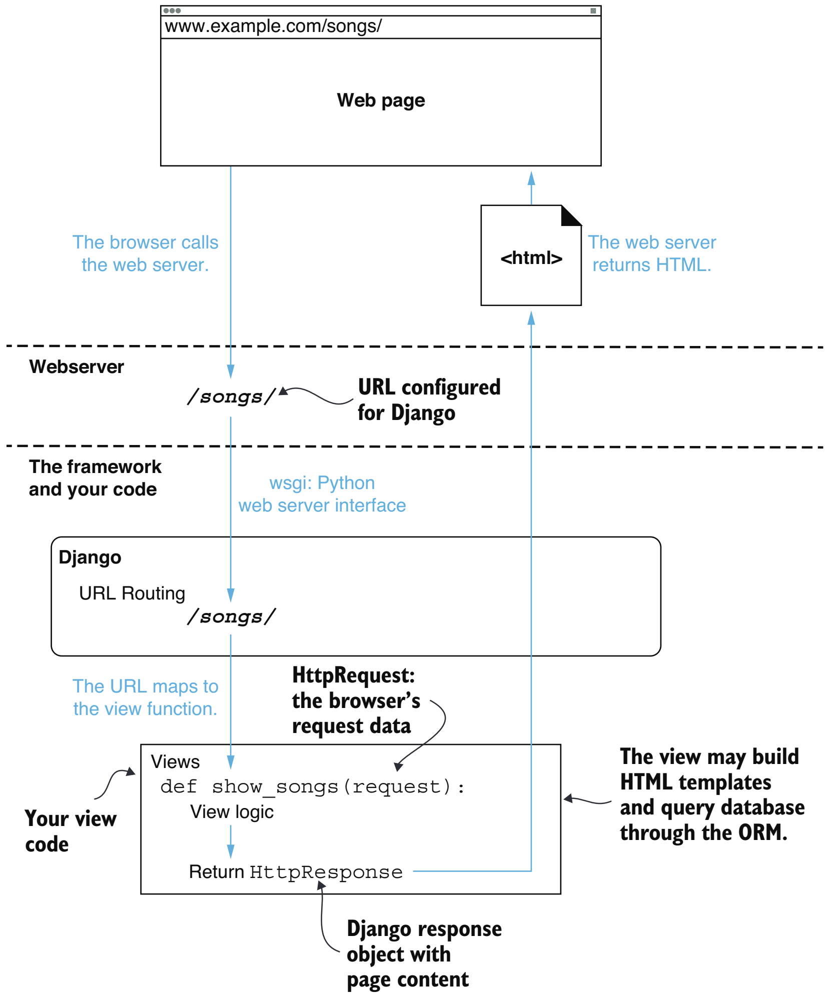

www.example.com/songs/
Web page
The browser calls the web server.
<html>
The web server returns HTML.
Webserver
/songs/
URL configured for Django
The framework and your code
wsgi: Python web server interface
Django
URL Routing
/songs/
The URL maps to the view function.
HttpRequest: the browser's request data
Views
def show_songs(request):
View logic
Return HttpResponse
Your view code
The view may build HTML templates and query database through the ORM.
Django response object with page content

Figure 1.1 Django handling a page visit

particular pages. To facilitate communication with a database, Django includes an *object relational mapping* (ORM), the second of the three core parts. This is an abstraction that allows you to manage content in a database through Python classes and objects. If your web page requires information from the database, the ORM is used to look up a `QuerySet` object that contains database results. How to use the ORM as well as how it maps to a database are covered in chapter 4.

The third core part of Django is the templating engine, which helps you create and manage the HTML within your page response. HTML is a fairly verbose text format often containing a lot of repeated content. Using the templating engine, you define HTML templates that describe parts of a web page. The parts get assembled, composed, and reused to form the response to the user.

For example, navigation bars or footers that appear on every page only need to be defined once, and then the template engine helps you include them everywhere. Templates are not HTML-specific; they also get used for populating email content and anywhere else you wish to manage text, like reusable code. Figure 1.2 shows a view in more detail and its interactions with the ORM, database, and templating engine.

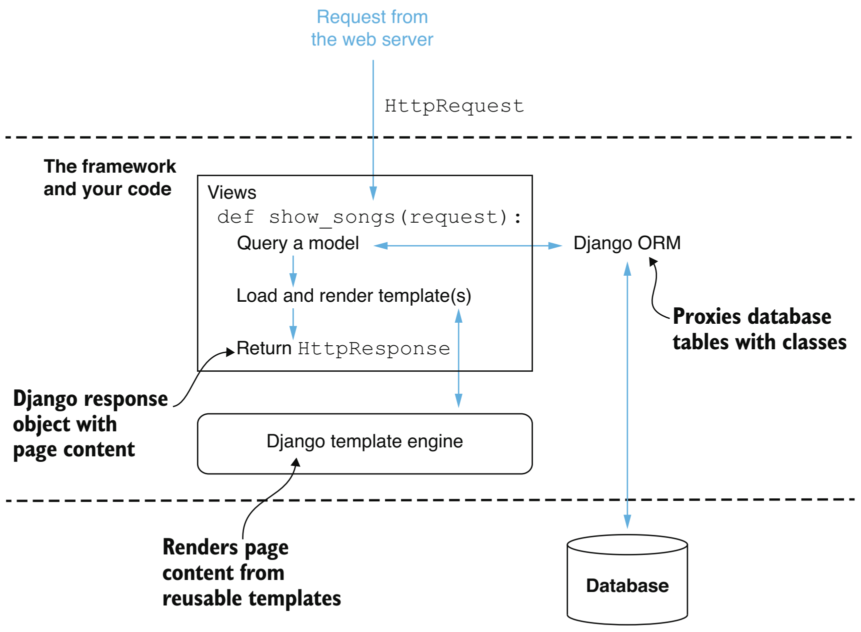

Request from the web server
HttpRequest
The framework and your code
Views
def show_songs(request):
Query a model
Load and render template(s)
Return HttpResponse
Django ORM
Proxies database tables with classes
Django response object with page content
Django template engine
Renders page content from reusable templates
Database

Figure 1.2 A view interacting with the database and template engine.

Consider a web page that shows the songs released in a given year. The URL for all the songs in 1942 might be http://example.com/song_by_year/1942/. The code for displaying this page starts with a mapping between the /song_by_year/ portion of the URL and a function to handle the response. The mapping is capable of parsing the /1942/ portion of the URL as a parameter and passing it into the view function that generates the page. Figure 1.3 shows the parts of the URL handled by the view and how the view interacts with the ORM.

The `song_by_year` view does a look-up in the database for all the songs in 1942. The database query gets abstracted through a `Song` class in the ORM, and a query gets performed by invoking a function on that class. The result from the database gets encapsulated in a `QuerySet` response object containing all the corresponding songs.

With the data in hand, the view creates the HTML response. It loads the songs.html template, which inherits from a common template that defines the look-

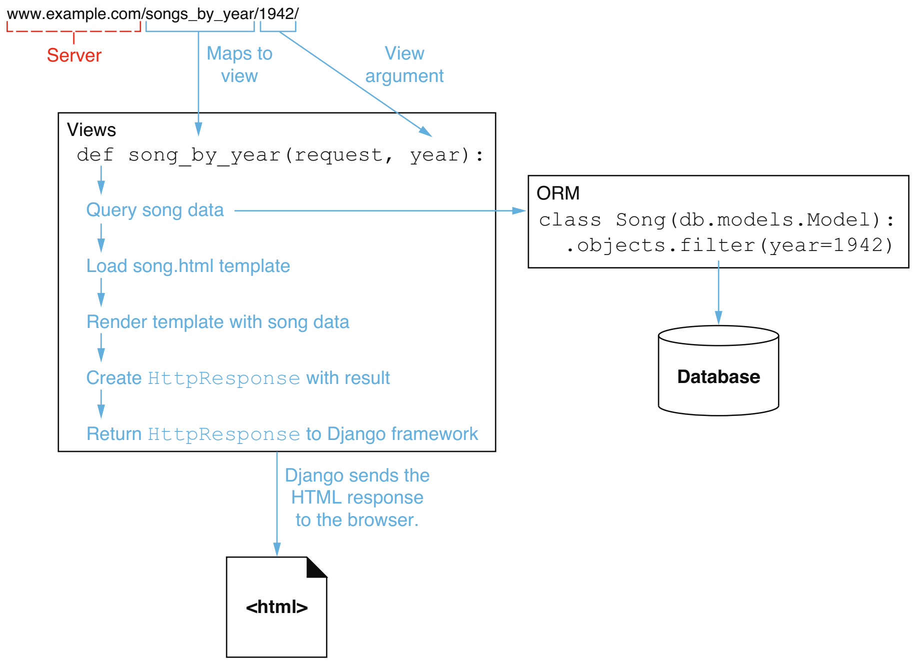

www.example.com/songs_by_year/1942/
Server
Maps to view
View argument
Views
def song_by_year(request, year):
Query song data
Load song.html template
Render template with song data
Create HttpResponse with result
Return HttpResponse to Django framework
ORM
class Song(db.models.Model):
.objects.filter(year=1942)
Database
Django sends the HTML response to the browser.
<html>

Figure 1.3 A song-by-year view rendering a web page based on an argument

and-feel for the website. This means songs.html only contains the parts that are unique to that page. The template engine has a tag language, which includes a feature that lets you loop over content, generating the rows in an HTML `<table>`, populating each row with the information from a song from 1942. You'll learn all about Django templates, their tags, and how to compose them together in chapter 3.

Once the HTML has been rendered, it gets packaged in an `HttpResponse` object and sent back to the Django framework. The loading and rendering of a template is so common that Django provides shortcuts to do this.

Django uses the response object to construct the data the web server needs for the browser. This includes all the HTTP headers and the body containing the HTML. The content gets returned to the web server through WSGI, and the server sends it down to the browser. That's how you see your page full of songs.

1.1.1 Mapping URLs, Django views, and the MVC model

The simplest form of any website is a mapping between a file and a URL. A web page that shows a picture of a kitten is at least two files: one for the HTML document and one for the image of the kitten. Simple file-to-URL mapped sites like this are called *static sites*. If you want your users to be able to upload their own images or add comments to your page, your site can no longer be just static files. A *dynamic site* is one that determines a page's content when it gets visited.

Dynamic sites are more complicated than static ones; they need an interface for the users to interact with, somewhere to store data like uploaded pictures and comments, and logic for knowing how to combine these into content. Django's primary purpose is building dynamic websites. In doing so, it uses a common architectural pattern for user interfaces known as the *model—view—controller* (MVC) method. This method breaks software into three pieces, as shown in the following list and figure 1.4:

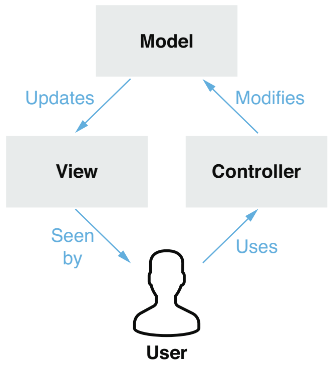


Figure 1.4 Model—view—controller architectural pattern

1. *Model*—Represents the logic and data of an application independent of its user interface
2. *View*—Presents data to the user, possibly with multiple views for the same data
3. *Controller*—An input and control mechanism the user uses to interact with models and views

The MVC is a good way of thinking about how software interacts with users, but Django is more inspired by it than strictly governed by it. The line between what is a view and a controller is a little fuzzy. This fuzziness provides the added benefit of creating entertaining debate on the internet.

When a web server passes a URL request to Django through WSGI, Django looks that URL up in its master mapping file, named urls.py, and as the name implies, it is just Python. The file can be renamed, but there really isn't any reason to do so. The urls.py file contains a variable called `urlpatterns`, which is a `list` containing Django `path` objects. Each `path` object either maps a URL to a Django view or loads another file containing more mappings.

A Django view is just Python code. It can be a function that returns a response document or a class with methods that do the same. Each view takes at least one argument, called the `request`. This variable contains information about the HTTP Request that was made. The request object includes the URL that was hit; any query parameters in the URL; HTTP method information; HTTP headers; and in the case of pages with forms, the data the user filled in. Not every view needs all this information, but it is there when you require it.

Key steps in Django's request handling process straddle the boundaries defined by the MVC. Django mapping a URL to the code that handles it is considered an MVC controller. Django's view, which returns the HTML document, is a hybrid of the MVC view and controller. Inside a Django view, models get queried, which makes it an MVC controller, but it also returns HTML, which makes it an MVC view. Just because Django names it *view* doesn't mean it is an MVC view—don't let the overlapping names confuse you.

Consider the prior song-by-year page example. The same Django view outputs different results, depending on the year passed in. The HTML template used in the

Django view is an MVC view; it is responsible for formatting the look and feel of the response. The database lookup in the Django view is an MVC controller, changing what data gets rendered by the MVC view based on the year argument.

Each page in your web application is generated by a view. Most are views that you write, but some are views you configure, taking advantage of tools built into Django. Figure 1.5 shows four web pages—two of which are views you would write, one is a Django tool view, and the other is a hybrid of the two.

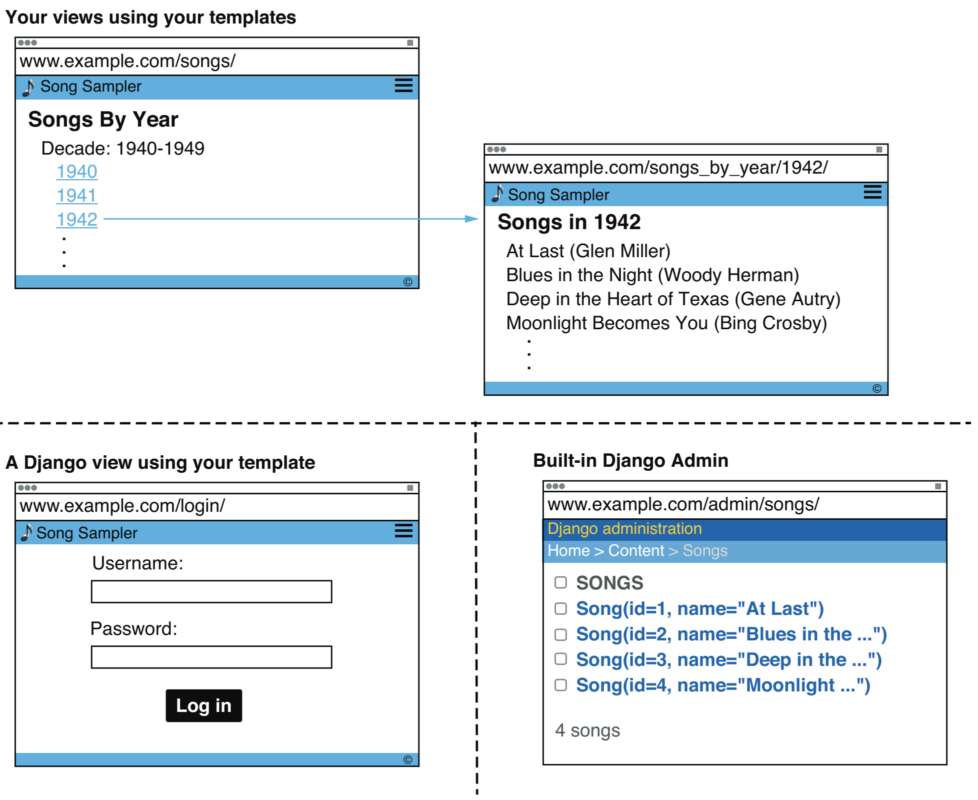

Your views using your templates
www.example.com/songs/
Song Sampler
Songs By Year
Decade: 1940-1949
1940
1941
1942
www.example.com/songs_by_year/1942/
Song Sampler
Songs in 1942
At Last (Glen Miller)
Blues in the Night (Woody Herman)
Deep in the Heart of Texas (Gene Autry)
Moonlight Becomes You (Bing Crosby)
A Django view using your template
www.example.com/login/
Song Sampler
Username:
Password:
Log in
Built-in Django Admin
www.example.com/admin/songs/
Django administration
Home > Content > Songs
SONGS
Song(id=1, name="At Last")
Song(id=2, name="Blues in the ...")
Song(id=3, name="Deep in the ...")
Song(id=4, name="Moonlight ...")
4 songs

Figure 1.5 Most pages will be your views with your templates, but Django provides tools to reduce your work

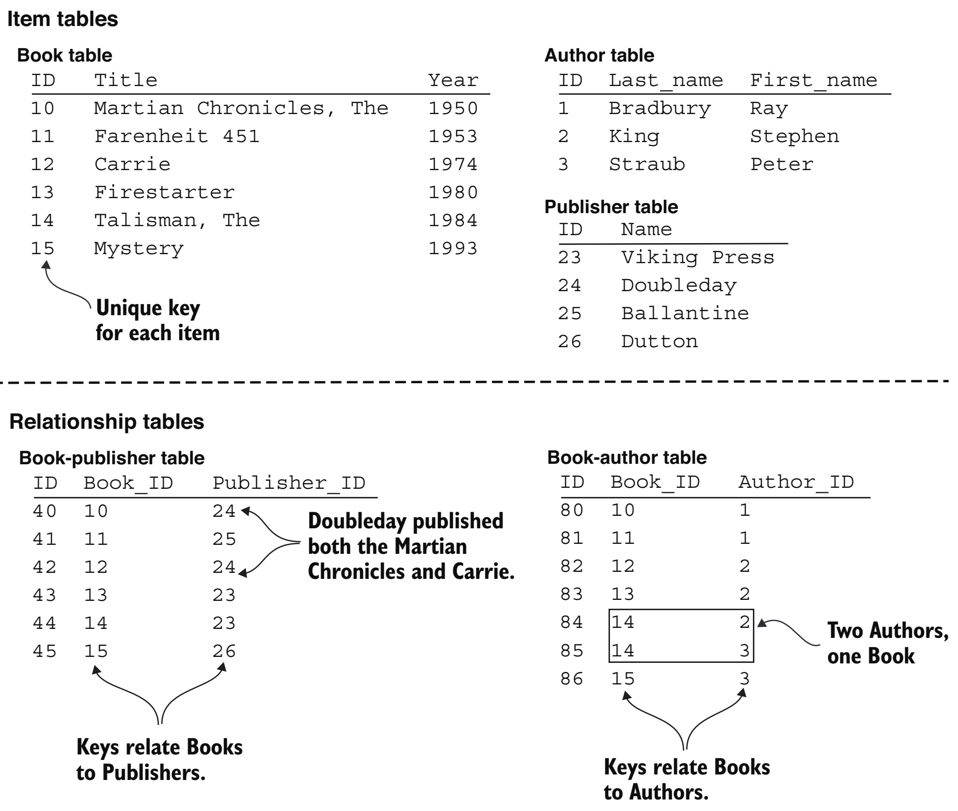
A typical view is responsible for querying data out of a database; making logic decisions about output; and then loading, rendering, and returning the result of a template. In the "Songs By Year" page, the view reads all the years from the song database and groups them by decade. The grouped data is sent to the template engine that renders the page, including links to each corresponding detail page with "Songs in 1942" being just one example. The same view generates all "Songs in" pages, with the year being passed as an argument. The argument changes how the MVC controller works but uses the same MVC view for the output.

1.1.2 Databases: Dynamic sites need data

Views are key to the *dynamic* part of a dynamic website. They can create what is on a page based on data or input from the user. That data might be products in a catalog, the score of last night's game, or your latest recipe for gnocchi. In all these cases, your data has to live somewhere. Enter the database.

Databases are a field in computing in their own right. There are different kinds, each with its advantages and disadvantages. There are many implementations of each kind and specialty languages written just for communicating with them. All of this can be a little overwhelming. Django implements an ORM to abstract this away.

Consider a database containing information about books, like the one shown in figure 1.6. You are going to have different kinds of data: an author (that's me), a publisher (that's Manning), and a book (that's what you're holding). There are many authors, many books, and many publishers. Some books have multiple authors, and some authors have written multiple books. A relational database stores this information in tables similar to a sheet in an Excel spreadsheet. It then expresses relationships between these items with keys. The Author table might reference the Book table using a key that is a unique identifier for a Book.



Item tables

Book table

ID	Title	Year
10	Martian Chronicles, The	1950
11	Farenheit 451	1953
12	Carrie	1974
13	Firestarter	1980
14	Talisman, The	1984
15	Mystery	1993

Author table

ID	Last_name	First_name
1	Bradbury	Ray
2	King	Stephen
3	Straub	Peter

Publisher table

ID	Name
23	Viking Press
24	Doubleday
25	Ballantine
26	Dutton

Relationship tables

Book-publisher table

ID	Book_ID	Publisher_ID
40	10	24
41	11	25
42	12	24
43	13	23
44	14	23
45	15	26

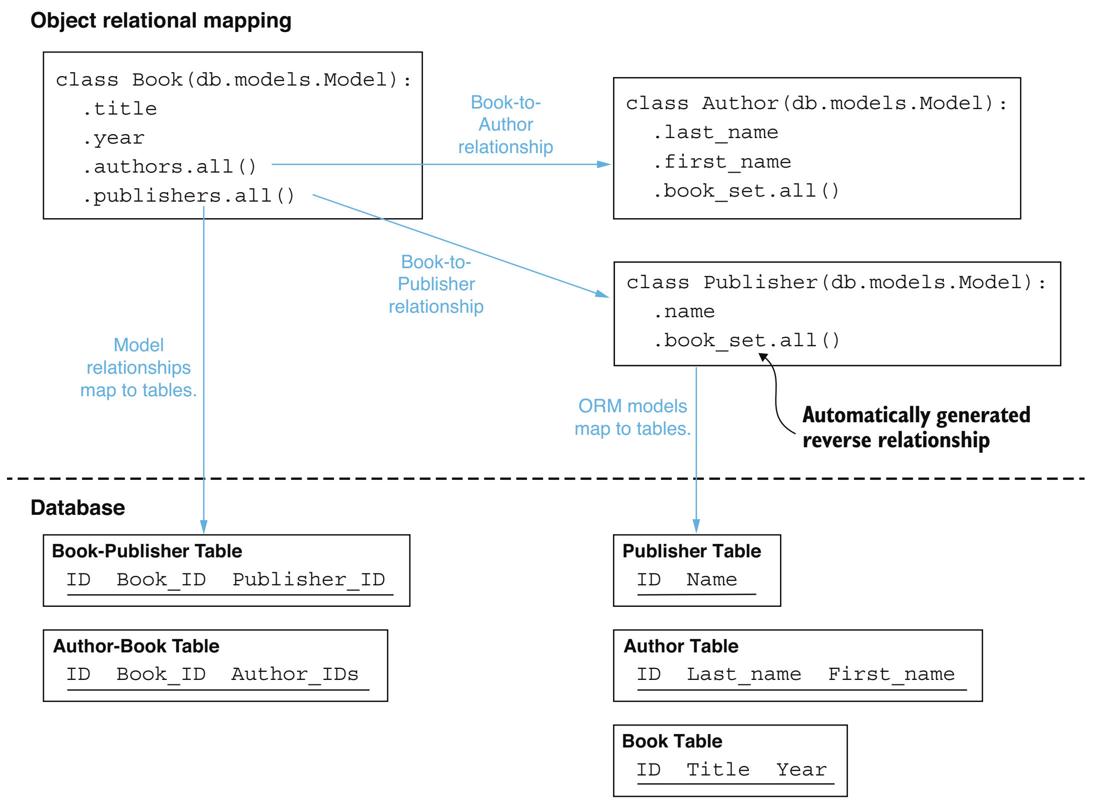
Book-author table

ID	Book_ID	Author_ID
80	10	1
81	11	1
82	12	2
83	13	2
84	14	2
85	14	3
86	15	3

Unique key for each item
Doubleday published both the Martian Chronicles and Carrie.
Two Authors, one Book
Keys relate Books to Publishers.
Keys relate Books to Authors.

Figure 1.6 Tables store either item information or relationship information

An ORM is an object-oriented way of representing different data elements and the relationships between them. Each row in a database table gets expressed as an object. Instead of thinking about the Book table, you think about the `Book` class. An ORM lets you query a specific book as well as link to the book's associated author and publisher, which themselves are objects. Django automatically creates reverse relationships, so if you have an `Author` object, you can query all the author's books without having to write any additional code. The relationships between the `Book`, `Author`, and `Publisher` objects are shown in figure 1.7, along with their corresponding database table representations.



Object relational mapping
class Book(db.models.Model):
.title
.year
.authors.all()
.publishers.all()
Book-to-Author relationship
class Author(db.models.Model):
.last_name
.first_name
.book_set.all()
Book-to-Publisher relationship
class Publisher(db.models.Model):
.name
.book_set.all()
Model relationships map to tables.
ORM models map to tables.
Automatically generated reverse relationship
Database
Book-Publisher Table
ID Book_ID Publisher_ID
Publisher Table
ID Name
Author-Book Table
ID Book_ID Author_IDs
Author Table
ID Last_name First_name
Book Table
ID Title Year

Figure 1.7 ORM Models are data objects that map to underlying item and relationship tables

Django's ORM allows you to define data models that map to tables in a database. You create rows in those tables by creating and saving objects. You can create, read, update, and delete data using the ORM, never having to touch the database itself.

Django's ORM comes with support for multiple databases, including PostgreSQL, MariaDB, MySQL, Oracle, and SQLite. There are also many third-party plugins for other databases. There are even plugins for non-relational databases, like the

MongoDB NoSQL server. One of the beauties of an ORM is that it abstracts this away; your code is the same regardless of what database is connected underneath.

When a Django view requires information out of the database it uses one or more ORM classes that have been defined by the programmer. Each class inherits from an ORM base class called `Model`. The base class provides methods for querying the database. For example, calling `.objects.filter(last_name__startswith="A")` on `Author` returns all the authors with a last name beginning with `"A"`. Query results get represented by a common Django object called a `QuerySet`. The `QuerySet` can be passed as part of a data context to the template engine. The template engine generates HTML using the results. A list of authors might generate a bullet-point list in the output. Django also provides a way to deal with the database directly if the abstraction is insufficient in specific cases.

1.1.3 Structuring HTML for reuse

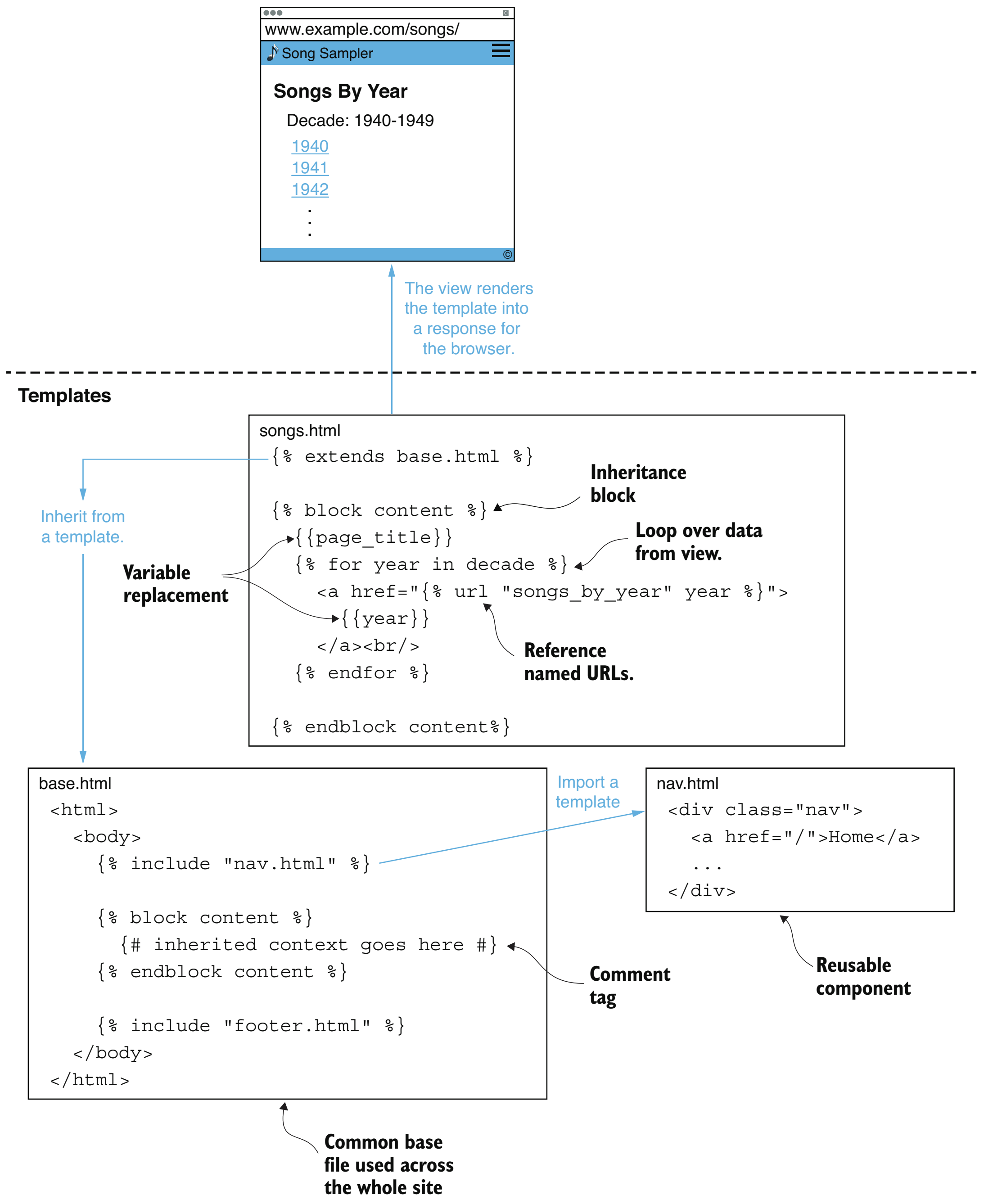
Your Django view is responsible for returning content to the browser, which most of the time is HTML. HTML was originally designed for physicists to write easily accessible articles for other physicists. Modern web pages have more in common with marketing brochures than physics articles. Each page in a website tends to include a lot of repetition—for example, the navigation header that appears on every page.

Django includes a templating engine. The engine's job is to compose pieces of content together into a page. Using it, you only have to write the navigation header once, and then it can be included in every page of your site. To change the navigation header, you only have to do it in one place.

Your view likely wants to show some of that previously mentioned data on your page. Lists of things can get quite repetitive. In Python, if you want to print a long list, you don't hardcode it. Instead, you put a `print` statement inside a `for` loop that iterates on the list. The templating engine lets you do the same using *tags* and *filters*. Tags are instructions to the engine for conditional rendering, while filters change how data is displayed. Using these, you can create loops, conditionally render blocks, print a date in your user's format, and much more. Figure 1.8 shows how templates are composed and how to use tags, denoted by `{% %}`. All of this is covered in detail in the chapter on the template engine.

Django is able to interact with different template engines. The original templating language that ships with Django, understandably, is called the *Django Templating Language* (DTL). Django also ships with the popular third-party templating language Jinja2. This isn't maintained by the Django core team, and won't be covered in this book, but is included in the framework for the convenience of those familiar with it.

The DTL interface has three concepts: variable rendering, tags, and filters. Variable rendering replaces values in the template with data sent into the engine. Tags are like functions and provide looping, conditional blocks, and a number of operations on your templated text. Filters are variable modifiers changing values before they are displayed. For example, one filter takes a date object and formats it to a desired output. The DTL is also pluggable, allowing you to write your own tags and filters.



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

{% block content %}
 {{page_title}}
 {% for year in decade %}
   <a href="{% url "songs_by_year" year %}">
     {{year}}
   </a><br/>
 {% endfor %}

{% endblock content%}
```

base.html
```
<html>
  <body>
    {% include "nav.html" %}

    {% block content %}
      {# inherited context goes here #}
    {% endblock content %}

    {% include "footer.html" %}
  </body>
</html>
```

nav.html
```
<div class="nav">
  <a href="/">Home</a>
  ...
</div>
```

Figure 1.8 Composing HTML templates into a rendered page

The output from a Django view is a response object—a web page uses one called `HttpResponse`. Django provides shortcut functions that wrap the process of loading and rendering a template and embedding the result in an `HttpResponse` object. The view returns an object to Django, and Django then uses the object to formulate the content that is sent back to the browser. This includes any HTTP headers along with the body containing the HTML.

1.1.4 Multiuser sites

From the very beginning, Django was meant for multiple users. The *Lawrence Journal-World's* web application consists of views for the readers, showing the newspaper pages as well as views for the journalists. The journalists' views are the tools they use to write articles.

Both the readers and the journalists are users, each using a different part of the web application. A regular reader should not be able to edit articles; they should only be able to read them. Controlling who can do what in a website can be broken into two parts: authentication and authorization. *Authentication* is the process of the user proving who they are, and *authorization* refers to the permissions associated with that user. Django provides tools for managing all of this.

User identities really are just another kind of data, so a multiuser Django site uses the ORM to track user accounts. Authentication compares a username and password from a login page with a known set of values in the database. Because users tend to be forgetful, you need more than just a login page—you also need a password reset screen. To manage users, you want tools for listing and editing them. You need pages for creating new users, either by an administrator or through self-sign-up. As users are people, you likely need to occasionally block one of them from logging in. Temporarily, of course—I'm sure they'll see the error of their ways. Django comes with tools to help you do all of these things. You'll learn all about users and the complications they cause in chapter 6.

In addition to providing user-management tools, Django includes a general tool called the Django Admin, which is a web-based administration interface for the database. Anything that you construct with the ORM can be managed through the Django Admin, including all of your user accounts. Having the Django Admin is a huge advantage and can save a large amount of time when building sites. The Django Admin is built on the same coding concepts as your own site: Django views, the ORM, and a series of templates. The tools that come with Django are built using the core parts of Django. You'll see how quickly you can create a site with a full administration interface in chapter 5.

1.2 What can you do with Django?

Django's original purpose at the *Lawrence Journal-World* was as a tool allowing journalists to upload articles to the web. The generic term for this kind of software is a *content management system*. Django is a general web framework, but it has a heavy leaning toward content management. For the journalists, the content in question is newspaper articles. For a blog site, the content is a blog post. In both of these cases, the content is text that makes up the page, but that isn't the only choice.

An e-commerce website has pages of items for sale, a shopping cart, and payment processing. This isn't really considered a content management system, but it does have some things in common. The catalog pages query the database for available items and display them—the "content" here is the merchandise. A shopping cart is a special case of this, where the contents of the cart are specific to the user and this visit.

Consider what blog and e-commerce sites have in common: there are "things" stored in the database (blog posts or merchandise), and users visit pages that display one or more of those things. The Django ORM provides an object-oriented abstraction for database storage, making it easier to code and manage these things, and the Django Admin provides a web interface for interacting with instances of them. Django views use the ORM to show the user either the blog posts or merchandise appropriate to the page and construct the page through the template engine. Its general approach of objects plus views rendering templates makes Django a good fit for any multipage website.

1.2.1 Server-side, single-page, and mixed-page applications

As web applications have evolved, the architectural choices in building them has grown. Inside a Django view, the template engine gets called to render HTML to be sent to the browser. This is called *server-side rendering*. By contrast, a *single-page application* (SPA) moves this rendering and some application logic into the browser itself. An SPA's approach to the MVC model is for the view and controller parts to be entirely in the web browser.

Server-Side Rendering vs. SPAs

The phrase *server-side rendering* can be a little confusing. Of course, in all situations, your browser displays the HTML. However, the distinction between server-side rendering and SPAs is where the HTML is composed. In server-side rendering, the server is responsible for assembling all of the pieces of the HTML and sending it down to the browser. The template engine runs on the server side, composing the parts of the template into a result, like in figure 1.8.

By contrast, SPAs create the HTML in the browser. All the logic for page creation happens in the browser, so you can think of this as a browser-based template engine. The server sends the code for building the templates to the browser as well as the data necessary, but all page composition happens on the client side.

This doesn't mean Django can't be used to build SPAs. When a user visits an SPA-based site, a JavaScript bundle gets downloaded. The bundle contains the complete user interface and a lot of the business logic for the application. The JavaScript presents a dynamic interface hosted in a single web page, hence the name. The application still needs the web server as a data store and to execute certain kinds of business logic, and this part of the architecture can be done with Django.

In a Django-based SPA, a view doesn't render HTML; instead, it serves data to the SPA. This is typically done using JSON. Plugin libraries like the *Django REST Framework* and *Django Ninja* provide features for building APIs based on views as well as tools for serializing your ORM data objects into a form the JavaScript application can consume.

The world isn't binary. There are degrees between a traditional server-side application and a full single-page application. Using libraries like Vue.js, to trigger API calls, or HTMX, which uses pieces of HTML, you can augment server-side rendering with client-side flexibility. Django fits this naturally, with views rendering full pages, supporting APIs, and HTML snippets, as required.

1.2.2 When and where you should use it

There are lots of choices in the Python world for writing web applications. A popular alternative to Django is *Flask*. Flask is smaller than Django and only directly offers the routing and view portion. Flask tends to require other libraries to build out a full system. It is often paired with SQLAlchemy for an ORM and Jinja2, for templating. By contrast, Django comes with an ORM and templating engine already. You may be able to get going with Flask a little faster than Django, but if your application grows, you're going to need the other pieces as well. With the routing, views, template rendering, and ORM all coming in one package, you don't have to worry about how and whether things will work together.

The flip side of this argument is that Django isn't really meant to be used in pieces. Although the ORM is fantastic, if you're not building a web application and only want the ORM, Django isn't the best fit. Django is intended to build web projects. You can use it as a tool for other things, but you'll find you're fighting it.

SPAs are designed with a backend server that only provides data, while the GUI component gets handled by the browser. The FastAPI library specializes in providing REST APIs and is a popular choice if you are using Python for the backend of an SPA.

If you are writing SPAs, there is a strong argument for using Node.js on the server side. Having JavaScript as both the client and server language has advantages. If you write your server in Python, you're going to have to duplicate some code, usually code for object to data conversion. Personally, my preference for Python far outweighs this extra work.

Django is sometimes viewed as a server-side only technology because that was what it was originally written for. The URL-maps-to-views approach has come full circle. Originally, it was how you built web pages. Then, single-page applications came along, and this approach was considered older tech. The same site structure works well for APIs though. This means Django can be used as a backend for single-page applications, as the server component for mobile apps, and when machine-to-machine communication is important. The availability of libraries like the Django Rest Framework and Django Ninja (heavily inspired by FastAPI) make writing APIs with Django relatively quick and painless. Being able to have both web pages and APIs in the same project means less code for you to write and maintain. How to use APIs with Django is covered in chapter 11.

Django has been around for a long time; its first release at the *Lawrence Journal-World* was in 2003. It can no longer be described as "the new kid on the block." The programming world has a tendency toward rejecting the old, but a distinction should be made between dinosaurs and sharks. Dinosaurs are extinct. Sharks were around with the dinosaurs, but they are still mean, eating machines today. Django is actively being developed and still releases major versions annually. Recent releases have focused on redesigning the underlying technology to better support asynchronous coding, making Django competitive with newer web development stacks.

Django isn't just a shark but a shark with accessories; it's modular in design and supports plugins. There are thousands of plugins available—one of which may already do what you were going to code for yourself. The Django Packages site (https://djangopackages.org/) has over 4,000 installable libraries that cover topics such as analytics, data tools, e-commerce, feed aggregation, forums, payment processing, polls, reporting, and social media. Whatever web application you want to build, there is likely a package that can help you.

Django's vast ecosystem is a real advantage, but even outside the Python world, there are things making Django better. HTMX is a JavaScript library that allows you to replace part of an active page with a snippet of HTML. You do this without having to write any JavaScript at all. Suddenly, Django's ability to serve and handle pieces of HTML, when combined with HTMX, means a resurgence of dynamic webpages without resorting to heavier, single-page application frameworks. Making your pages more dynamic using HTMX is covered in chapter 12.

1.2.3 Potential projects

There are many types of websites out there, and depending on what you're building, you may end up using different parts of the Django framework. The following is an incomplete list of the types of projects you might be interested in:

- *Blogs*—These are a natural fit for Django, with the ORM storing article content and views responsible for showing articles and table-of-contents pages. Wiki sites and newspapers are a variation on this, where instead of blog posts, you have encyclopedic information or newspaper articles as content.
- *e-commerce sites*—These sites are all about the catalog. The ORM stores information about your merchandise, while the views present them to the users. In complex sites, you may be interacting with an inventory backend instead or exporting shopping patterns for data analysis. The Django Admin provides an out-of-the-box inventory management system by working in conjunction with the ORM.
- *Content sites*—Many websites are repositories of specialty information. Consider the Internet Movie Database (IMDB); it is really a frontend to a database entry describing information about a movie. Django's ORM and the Django Admin make it a good tool for these kinds of sites.

- *Document management*—This is a broad class of sites where some content gets served from the filesystem instead of in the ORM. Typically, the ORM contains a lightweight object that points to the file on the server or CDN. Users then interact with pages to see or download the documents. Video sites are a version of this, where the document is a movie file. Data scientists use this kind of site for presenting reports or results from their analysis.
- *Utility sites*—It is quite common for developers and system administrators to create utility scripts solving specific problems in an organization. Often, these scripts are difficult to share, as they're environment specific. Creating a web frontend for these kinds of utilities makes them useful to a wider audience within an organization. Similarly, most companies have share folders full of Excel files that are ad-hoc solutions to a problem. Excel can be a fragile beast; there is seldom any input validation, and a small change in the wrong cell can produce the wrong result. Processes like this that are used by many people call out for a web version, converting the logic in Excel into a web application providing input validation and robustness.

Orthogonal to the type of site, is your architectural approach. Your project may use one or more of the following design concepts:

- *Single-page applications*—Not so much a type of project but a design approach, SPAs can be built with Django as their API backend. Third-party libraries, like Django Ninja, simplify the creation of these kinds of projects.
- *Multiuser sites*—Django includes user account management out of the box. The Django Admin can be used to administer user accounts, and Django provides views for login and password management.
- *Multilingual sites*—Django includes mechanisms for handling multiple written languages and localization, such as date and currency format. This can be especially powerful when mixed in with multiuser capabilities, as users see the site in their native language.
- *Static-page sites*—For speed of interaction, nothing beats a static website, where the server feeds the user straight HTML. Of course, maintaining these kinds of sites can be painful, unless you use tools that allow you to compose the page. The third-party library django-distill is a site generator; you build a Django site like you normally would then use distill to output it as a static site. This gives you all the advantages of the Django template engine while keeping the performance gain of a static site.

1.2.4 The RiffMates project

Books are great (you should buy multiple copies for your friends), but practice is where you really learn things. This book presents a project called *RiffMates*, where you help your cousin build a musicians-seeking-musicians personals site. This project is primarily the content site type, but it mixes in some document management aspects, all in a multiuser context. Coding along will help you better understand how to build your own projects. Each chapter includes exercises to test your progress, and sample code and answers are available online at https://github.com/cltrudeau/django-in-action.

This book is divided into three parts. The first part covers Django's core components, the second part delves into Django's built-in tools, and the third part covers how to use third-party libraries to build different kinds of sites.

In part 1, you start to build RiffMates. You will write views and register them against URLs. You'll learn about the template engine and how to load and render templates in your views to output HTML to the browser. You will also learn how to interact with the database doing queries on musician and venue data, while using the Django Admin to manage it all.

Part 2 shows you the power of Django and all the nifty stuff built into it. You'll turn RiffMates into a multiuser site and learn how to customize the provided views dealing with logins and password management. Once you've got user accounts happening, you will add the ability to get input from your users, including uploaded files.

Part 3 focuses on how key third-party libraries can help you extend the functionality of your website. Django was originally designed as a content management system framework, but its community has created all sorts of pluggable tools to do other things. You can build websites with REST APIs; deal with data import and export; and interact with popular JavaScript frameworks, like HTMX.

Happy web coding!

Summary

- Django maps a URL to a view, which is either a function or a class that returns a document to the browser.
- A Django view can render a document by composing pieces of HTML through a template engine.
- The model—view—controller architectural pattern divides the parts of a program into three pieces. The model contains data and business logic, the view is for visualizing the data, and the controller is for interacting with it.
- An object relational mapping is a way of abstracting database tables with programming classes.
- Django includes tools for managing multiuser websites.
- Single-page applications move the view and controller parts of the MVC into the browser.
- Django can be used as the backend for a single-page application.

2 Your first Django site

This chapter covers

- Creating a Django project
- Django projects and apps
- Creating a Django app
- Running the development server
- Writing a Django view

In this chapter, you'll start your Django coding journey by taking the first steps toward building a website for musicians and bands. You'll create your first web page using the Django framework: a credits page showing who built the site.

2.1 The RiffMates project

Your cousin Nicky is a guitarist, and she has a great idea for a website: classifieds for musicians. The site will cater to musicians looking for bands, bands looking for musicians, venues looking for bands, and bands looking for gigs. Nicky even has a name picked out, *RiffMates*. Over coffee, you and Nicky have talked at great length about what the site needs, including the following:

- Profile pages for musicians, bands, and venues
- Classified ad listings
- Announcement listings for tryouts
- Search capabilities based on instruments played and style of music
- Messaging between potential matches

Nicky has confidence in your coding skills because you've talked excitedly about the Python you've learned in the past. As you think about the problem, you realize that those features are just the business requirements. To build a working website that can do all those things, you'll also need features that are common to most websites, such as

- Public-facing and private-only web pages
- Logins
- Ability for users to submit both text and image content
- User sign-up
- Password management
- A database for storage
- Admin pages for managing the site

A lot goes into what seems like a simple website. Asking around, you've heard that Django provides all those capabilities for you, so you've decided to learn Django to help you build RiffMates.

Django is a *framework*. Programming with a framework is different than programming with a library. In the case of a library, your code determines the order of execution. By contrast, when you're building inside a framework, execution control belongs to the framework. If you were building a house, a library would contain the pieces you needed for the house, whereas a framework is already the rough outline of the house. Working within a framework means your application designs are more constrained, but what you lose in flexibility, you gain in efficiency—there is less code for you to write.

The partially completed building that you get with Django is the structure of a website. Django calls this a *project*. A Django project is what interacts with a web server to respond when a user visits a URL. Your code lives inside the project and gets invoked by Django. Each time you want to create a new site, you create a new project, and you need to have the project in place before you can start adding pages to it. The rest of this chapter covers creating a project and writing your first web page inside it.

2.2 Creating a Django project

The Django framework includes a command-line tool that is used for creating new projects, called `django-admin` (not to be confused with the Django Admin, the web tool for managing your data). Django is published on PyPI (https://pypi.org/) and can be installed in a Python virtual environment using the command `python -m pip install django`. Once you've got the package installed, you use the command `django-admin` to create a skeleton framework for your project.

Using virtual environments

Each time you install a package in Python, you're making it available to the entire system. As different projects have different needs, this can cause conflicts. To deal with this, Python uses *virtual environments*, which are self-contained Python setups that are isolated from each other. It is important to use a virtual environment when writing Python to avoid packaging conflicts.

Appendix A contains a brief overview of installing packages in Python virtual environments. For a full explanation on the technology, see http://mng.bz/JZVZ.

Each Django project is a Python program, but because it is built using a framework, the project provides the execution paths. When you create a new project, Django installs the bare minimum you need inside a new directory with the name that you give it.

At the heart of a Django project is a file named wsgi.py. WSGI, a Python specification that describes how code interacts with a web server, stands for *Web Server Gateway Interface*. You really don't need to know much about this, especially when you're working in your development environment; the file gets generated for you. Inside wsgi.py is a short program responsible for interfacing between your code and the web server. When you're ready to put your project in production, you'll configure your web server to point at this application in your project. This is why Django is considered a framework rather than a library: wsgi.py is the entry point, and most of the code you write will be called through it. Figure 2.1 shows you how the WSGI protocol acts as a bridge to your code through the wsgi.py file.

When you build RiffMates, or any other website, what you're really trying to do is make web pages. Each page lives at a URL, and each time a user visits a URL, you're going to want some code called to generate the page. In Django, this code is called a *view*.

When you load a page, or in some cases interact with one, your web browser is communicating with a web server. In the case of a Django project, the web server communicates with the WSGI application defined in wsgi.py. The WSGI application triggers the Django framework code. Django runs a view that is responsible for returning content that Django turns into an HTTP response for the browser.

Django needs to know what view to call for each URL on the website. This is done by mapping URLs to views. Within the framework, a URL sometimes is known as a *route*. Each Django project has a main urls.py file that contains a list of URL mappings, called `urlpatterns`. The list contains `django.urls.path` objects that specify a pattern that matches one or more URLs and what view to call when that URL gets visited.

The wsgi.py and urls.py files are key to your Django project. Every new website you build with Django requires its own copy of these files. To make your life easier, the `django-admin` command that comes with the package creates these files for you, customizing them based on your project name.

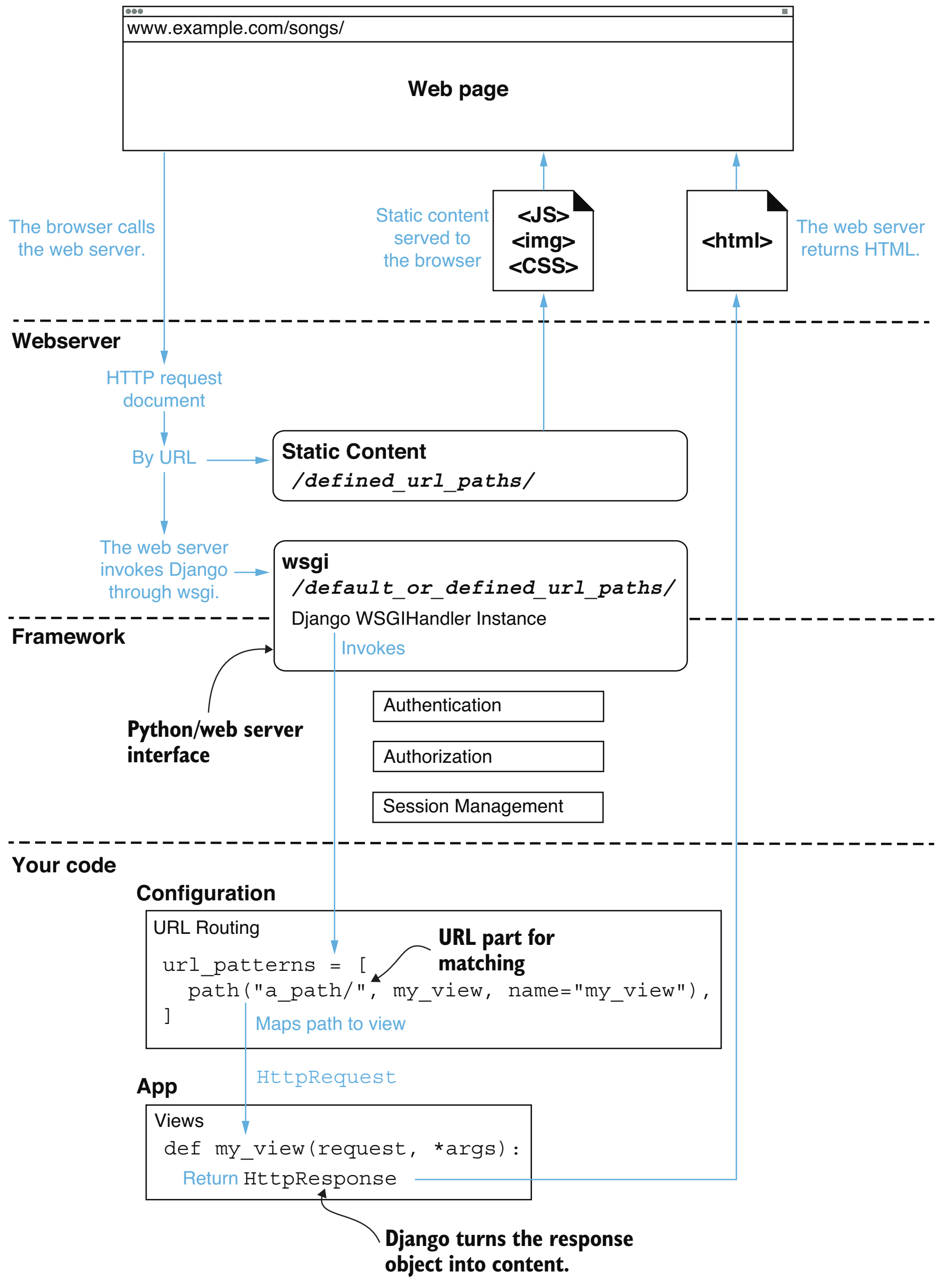

www.example.com/songs/
Web page
The browser calls the web server.
Static content served to the browser
<JS> <img> <CSS>
<html>
The web server returns HTML.
Webserver
HTTP request document
By URL
Static Content
/defined_url_paths/
The web server invokes Django through wsgi.
wsgi
/default_or_defined_url_paths/
Django WSGIHandler Instance
Invokes
Framework
Authentication
Authorization
Session Management
Python/web server interface
Your code
Configuration
URL Routing
url_patterns = [
path("a_path/", my_view, name="my_view"),
]
URL part for matching
Maps path to view
HttpRequest
App
Views
def my_view(request, *args):
Return HttpResponse
Django turns the response object into content.

Figure 2.1 A web server invokes Django through WSGI, Django calls your code, and then it formats the response for a browser

The `django-admin` program is actually the entry point for a number of configuration activities. Each activity gets invoked using a command provided as an argument to `django-admin`. To create a new Django project, you use the `startproject` command, giving it the name of your project. By default, `startproject` creates a new directory for your project, copies a script called manage.py inside, and creates a configuration directory. The manage.py script is similar to `django-admin` and is used to run various control and configuration actions in the project. The configuration directory contains wsgi.py, urls.py, and other files.

The configuration directory is confusingly named: it has the same name as your project. When you create RiffMates, you're going to get both a RiffMates directory and a RiffMates/RiffMates directory. It is unclear to me why the Django folks didn't name this something like *config*, as this causes confusion when you're talking to someone else about the code: Are you in the RiffMates directory? "Yes." Which one?

Naming your project

Python modules are typically named using *snake case*, or lowercase words separated by underscores. I'm not sure where I picked up the habit of naming my Django projects in violation of this convention—it may be because I'm an ex-Java programmer. If seeing *RiffMates* makes you cringe, feel free to name your project in a more "Pythonic" fashion.

Because your Django project is a directory itself, you have some flexibility about where you put your Python virtual environment. Some virtual environment tools prefer keeping all the environments together in a common directory, whereas others keep the environment with the project. In this latter case, you can put your virtual environment inside your project folder, which is what I'll be doing in the example here.

To create your first Django project, you're going to do the following:

1. Create a virtual environment.
2. Install Django into that virtual environment.
3. Use the Django `django-admin` tool to create your RiffMates project.

Once you've done all of this, you will be able to start the development server and point your browser at your new site. Open a command-line terminal, and change to the directory where you normally keep your code. Inside the terminal, create a directory for your project and a virtual environment inside of it:

```
$ mkdir RiffMates
$ cd RiffMates
RiffMates$ python3.12 -m venv ./venv
```

Create a directory for your project.

Create a virtual environment.

NOTE Most of this book will be spent in Python, but to get going, you need to run some command-line scripts. All command-line examples use Unix syntax with dollar-sign ($) denoting the prompt. The value before the dollar-sign indicates the current directory and whether a virtual environment has been activated. For details on how to install Unix-compatible tools on Windows, see appendix A.

With the project directory and virtual environment in place, the next step is to install Django. Activate the virtual environment, and then run `pip install`:

```
RiffMates$ source venv/bin/activate                  <-- Activate your virtual environment.
(venv) RiffMates$ python -m pip install django       <-- Install Django.
```

Django is dependent on a couple of other packages. The `pip install` command will install the dependencies. Your results will look something like this:

```
Collecting django
  Using cached Django-5.0.2-py3-none-any.whl.metadata (4.1 kB)
Collecting asgiref<4,>=3.7.0 (from django)
  Using cached asgiref-3.7.2-py3-none-any.whl.metadata (9.2 kB)
Collecting sqlparse>=0.3.1 (from django)
  Using cached sqlparse-0.4.4-py3-none-any.whl (41 kB)
Using cached Django-5.0.2-py3-none-any.whl (8.2 MB)
Using cached asgiref-3.7.2-py3-none-any.whl (24 kB)
Installing collected packages: sqlparse, asgiref, django
Successfully installed asgiref-3.7.2 django-5.0.2 sqlparse-0.4.4
```

Now that you have Django installed, the last step is actually creating your project. Run the `startproject` command:

```
(venv) RiffMates$ django-admin startproject RiffMates .   <-- Create your Django project.
```

Virtual environment placement and Django projects

The previous steps created the RiffMates project directory manually. It was done this way so that the virtual environment directory could be in the project directory.

By default, the `django-admin startproject` command creates your project directory for you. Some coders prefer to keep their virtual environments in a central location, rather than putting them inside the project. If you prefer this organization, you can skip the project directory creation step and use `django-admin startproject RiffMates` without the period at the end, instead:

```
code$ source ~/wherever-your-RiffMates-venv-is/bin/activate
(RIFF) code$ python -m pip install django
Collecting django
  Using cached Django-5.0.2-py3-none-any.whl.metadata (4.1 kB)
Collecting asgiref<4,>=3.7.0 (from django)
  Using cached asgiref-3.7.2-py3-none-any.whl.metadata (9.2 kB)
Collecting sqlparse>=0.3.1 (from django)
  Using cached sqlparse-0.4.4-py3-none-any.whl (41 kB)
Using cached Django-5.0.2-py3-none-any.whl (8.2 MB)
Using cached asgiref-3.7.2-py3-none-any.whl (24 kB)
Installing collected packages: sqlparse, asgiref, django
Successfully installed asgiref-3.7.2 django-5.0.2 sqlparse-0.4.4
(RIFF) code$ django-admin startproject RiffMates   <-- startproject without the trailing period (.)
(RIFF) code$ ls
RiffMates/
```

Running the `django-admin startproject` command creates a skeleton of your project. The new RiffMates project directory looks like this:

```
RiffMates/
├── RiffMates
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py
```

The project directory (RiffMates) contains the confusingly named configuration directory (RiffMates/RiffMates), the virtual environment, and the manage.py command. The manage.py command does pretty much the same things as `django-admin`, except it is connected to, and aware of, your project.

The RiffMates/RiffMates configuration directory contains the previously mentioned wsgi.py and urls.py files, along with a few others. Your project is a Python program, so the configuration directory is a module, meaning it contains an __init__.py file.

The asgi.py file is an asynchronous version of WSGI. Depending on the kind of web server you're using, you might point to asgi.py instead of wsgi.py. More details on when these two files get used is covered in appendix B.

The final file in the configuration directory is settings.py; it contains configuration values for changing the behavior of Django. (We'll have more on that in a bit.)

You've created your first Django project, and now, you actually have a website! Albeit, it's a very small, rather boring website, but it exists. You haven't built any pages yet, but Django provides a default one for you, and it has a fancy rocket ship on it! To see the website and bask in the glory of the rocket ship, you need a web server. Django ships with one, and when you're developing your site, this is the server to use.

You start the Django development server using the manage.py script and the `runserver` command. Note that you do this inside the RiffMates project directory, not the RiffMates/RiffMates configuration directory! Start the Django development server by running the following command:

```
(venv) RiffMates$ python manage.py runserver
```

Type less on Unix

Throughout this book, when it is time to run a Django management command, the instructions are to type python manage.py, as this is compatible across all operating systems. On Unix systems, including Linux and macOS, manage.py is executable, so you can use `./` execution. For example, the development server command gets shortened to the following:

```
(venv) RiffMates$ ./manage.py runserver
```

There are a number of management commands you'll be using frequently, so you may want to add tab completion to your shell. On bash, run the following (or add it to your .bash_profile or .bash_rc file):

(continued)

```
$ complete -f -d -W "runserver createsuperuser test shell dbshell \
migrate makemigrations loaddata dumpdata" ./manage.py
```

This allows you to type manage.py, the first letter of a management command, and then press `TAB` to auto-complete the name of the command, saving even more typing. Other shells have equivalent completion mechanisms; in fact, some versions of zsh ship with auto-completion for the Django command built-in.

Once going, the development server shows you some debug info and then tells you its base URL. By default, the server listens on port 8000 of your local machine. If you prefer a different port number, you can provide it as part of the `runserver` command (e.g., `python manage.py runserver 8800`). The output from the server command looks like this:

```
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly
until you apply the migrations for app(s): admin, auth, contenttypes,
sessions. Run 'python manage.py migrate' to apply them.

Django version 5.0.2, using settings 'RiffMates.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

A warning that can be ignored for now

Listening on port 8000

Don't worry about the `18 unapplied migration(s)` message—you'll deal with it shortly. Remember that `localhost` is an alias to the IP address 127.0.0.1, so you can also use the English alias http://localhost:8000/ to visit your site. Enter the base URL into your browser and see the rocket ship on the default Django project page (figure 2.2).

Dev servers are for development

For convenience, Django includes a web server. This means you don't need to install and manage one when you're creating and maintaining your website. This is not a hardened web server though, and it should never be put into an untrusted environment.

Do not expose the Django development server directly to the internet, and if you're using it internally at a company, speak with your IT department about whether it should be hosted behind your firewall. Furthermore, the default configuration for a Django project is in debug mode, which also should not be used in production. Appendix B discusses what you need to know to put your Django project into a hardened production environment. It is right there in the name: the *development server* should only be used for *development*!