



DJANGO

for

PROFESSIONALS

Production websites with Python & Django

WILLIAM S. VINCENT

Django for Professionals

Production websites with Python & Django

William S. Vincent

© 2018 - 2019 William S. Vincent

Also By William S. Vincent

Django for Beginners

Django for APIs

Contents

Introduction	1
Prerequisites	2
Book Structure	3
Book Layout	4
Text Editor	5
Conclusion	6
Chapter 1: Docker	7
What is Docker?	8
Containers vs. Virtual Environments	9
Install Docker	10
Docker Hello, World	11
Django Hello, World	13
Pages App	15
Images, Containers, and the Docker Host	20
Git	26
Conclusion	26
Chapter 2: PostgreSQL	28
Starting	29
Docker	30
Detached Mode	33
PostgreSQL	36

CONTENTS

Settings	39
Pyscopg	41
New Database	43
Git	46
Conclusion	47
Chapter 3: Bookstore Project	48
Docker	50
PostgreSQL	53
Custom User Model	54
Custom User Forms	57
Custom User Admin	59
Superuser	60
Tests	61
Unit Tests	62
Git	64
Conclusion	65
Chapter 4: Pages App	66
Templates	67
URLs and Views	70
Tests	72
Testing Templates	74
Testing HTML	76
setUp Method	78
Resolve	80
Git	82
Conclusion	83
Chapter 5: User Registration	84

CONTENTS

Auth App	84
Auth URLs and Views	86
Homepage	87
Redirects	90
Django Source Code	91
Log Out	94
Log In	96
Sign Up	98
Tests	104
setUpTestData()	107
Git	108
Conclusion	108
Chapter 6: Static Assets	109
staticfiles app	109
STATIC_URL	109
STATICFILES_DIRS	110
STATIC_ROOT	110
STATICFILES_FINDERS	111
Static Directory	112
Images	114
JavaScript	116
collectstatic	118
Bootstrap	119
About Page	122
Django Crispy Forms	125
Tests	130
Git	132
Conclusion	132

CONTENTS

Chapter 7: Advanced User Registration	133
django-allauth	133
AUTHENTICATION_BACKENDS	135
EMAIL_BACKEND	136
ACCOUNT_LOGOUT_REDIRECT	137
URLs	139
Templates	140
Log In	142
Log Out	145
Sign Up	146
Admin	150
Email Only Login	153
Tests	157
Social	160
Git	160
Conclusion	161
Chapter 8: Environment Variables	162
.env files	163
SECRET_KEY	163
DEBUG	166
Databases	167
Git	168
Conclusion	168
Chapter 9: Email	169
Custom Confirmation Emails	169
Email Confirmation Page	176
Password Reset and Password Change	179
Email Service	179

CONTENTS

Git	180
Conclusion	181
Chapter 10: Books App	182
Models	183
Admin	185
URLs	189
Views	190
Templates	191
object_list	192
Individual Book Page	194
context_object_name	198
get_absolute_url	199
Primary Keys vs. IDs	201
Slugs vs. UUIDs	202
Navbar	206
Tests	207
Git	209
Conclusion	209
Chapter 11: Reviews App	211
Foreign Keys	211
Reviews model	212
Admin	215
Templates	219
Tests	221
Git	224
Conclusion	224
Chapter 12: File/Image Uploads	226

CONTENTS

Media Files	226
Models	228
Admin	231
Template	233
Next Steps	237
Git	237
Conclusion	238
Chapter 13: Permissions	239
Logged-In Users Only	239
Permissions	241
Custom Permissions	244
User Permissions	245
PermissionRequiredMixin	247
Groups & UserPassesTestMixin	250
Git	250
Conclusion	250
Chapter 14: Orders with Stripe	252
Payments Flow	253
Orders app	253
Stripe	258
Publishable & Secret Keys	262
Stripe Checkout	264
Charges	270
Stripe + Permissions	274
Templates	276
Tests	278
Git	278
Conclusion	278

CONTENTS

Chapter 15: Search	280
Search Results Page	280
Basic Filtering	283
Q Objects	285
Forms	286
Search Form	287
Git	290
Conclusion	290
Chapter 16: Performance	292
django-debug-toolbar	292
Analyzing Pages	297
select_related and prefetch_related	299
Caching	299
Indexes	302
django-extensions	304
Front-end Assets	304
Git	305
Conclusion	305
Chapter 17: Security	306
Social Engineering	306
Django updates	307
Deployment Checklist	308
Local vs. Production	308
DEBUG	310
ALLOWED HOSTS	311
Web Security	312
SQL injection	313
XSS (Cross Site Scripting)	313

CONTENTS

Cross-Site Request Forgery (CSRF)	315
Clickjacking Protection	316
HTTPS/SSL	317
HTTP Strict Transport Security (HSTS)	318
Secure Cookies	319
Admin Hardening	320
Git	322
Conclusion	322
Chapter 18: Deployment	323
PaaS vs IaaS	323
WhiteNoise	324
Gunicorn	326
dj-database-url	327
Heroku	328
Deploying with Docker	329
heroku.yml	330
Heroku Deployment	332
SECURE_PROXY_SSL_HEADER	339
Heroku Logs	340
Stripe Live Payments	341
Heroku Add-ons	342
PonyCheckup	343
Conclusion	344
Conclusion	345

Introduction

Welcome to *Django for Professionals*, a guide to building professional websites with the [Django web framework](#). There is a massive gulf between building simple “toy apps” that can be created and deployed quickly and what it takes to build a “production-ready” web application suitable for deployment to thousands or even millions of users. This book will show you how to bridge that gap.

When you first install Django and create a new project the default settings are geared towards fast local development. And this makes sense: there’s no need to add all the additional features required of a large website until you know you need them. These defaults include SQLite as the default database, a local web server, local static asset hosting, built-in `User` model, and `DEBUG` mode turned on.

But for a production project many, if not most, of these settings must be reconfigured. And even then there can be a frustrating lack of agreement among the experts. For example, what’s the best production database to use? Many Django developers, myself included, choose PostgreSQL. It is what we will use in this book. However an argument can be made for MySQL depending on the project. It really does all depend on the specific requirements of a project.

Rather than overwhelm the reader with the full array of choices available this book instead shows one approach, grounded in current Django community best practices, for building a professional website. The topics covered include using Docker for local development and deployment, PostgreSQL, a custom user model, robust user authentication flow with email, comprehensive testing, environment variables, security and performance improvements, and more.

By the end of this book you will have built a professional website and learned all

the necessary steps to do so. Whether you are starting a new project that hopes to be as large as Instagram (currently the largest Django website in the world) or making much-needed updates to an existing Django project, you will have the tools and knowledge to do so.

Prerequisites

If you're brand-new to either Django or web development, this is not the book for you. The pace will be far too fast. While you *could* read along, copy all the code, and have a working website at the end, I instead recommend starting with my book [Django for Beginners](#). It starts with the very basics and progressively introduces concepts via building five increasingly complex Django applications. After completing that book you will be ready for success with this book.

I have also written a book on transforming Django websites into web APIs called [Django for APIs](#). In practice most Django developers work in teams with other developers and focus on back-end APIs, not full-stack web applications that require dedicated JavaScript front-ends. Reading *Django for APIs* is therefore helpful to your education as a Django developer, but not required before reading this book.

We will use Docker throughout most of this book but still rely, briefly, on having Python 3, Django, and Pipenv installed locally. Git is also a necessary part of the developer toolchain. If you need help on these steps you can [find more details here](#).

Finally we will be using the command line extensively in this book as well so if you need a refresher on it, [please see here](#).

Book Structure

Chapter 1 starts with an introduction to Docker and explores how to “dockerize” a traditional Django project. In *Chapter 2* PostgreSQL is introduced, a production-ready database that we can run locally within our Docker environment. Then *Chapter 3* starts the main project in the book: an online Bookstore featuring a custom user model, payments, search, image uploads, permissions, and a host of other goodies.

Chapter 4 focuses on building out a `Pages` app for a basic homepage along with robust testing which is included with every new feature on the site. In *Chapter 5* a complete user registration flow is implemented from scratch using the built-in `auth` app for sign up, log in, and log out. *Chapter 6* introduces proper static asset configuration for CSS, JavaScript, and images as well as the addition of Bootstrap for styling.

In *Chapter 7* the focus shifts to advanced user registration, namely including email-only log in and social authentication via the third-party `django-allauth` package. *Chapter 8* introduces environment variables, a key component of Twelve-Factor App development and a best practice widely used in the web development community. Rounding out the set up of our project, *Chapter 9* focuses on email and adding a dedicated third-party provider.

The structure of the first half of the book is intentional. When it comes time to build your own Django projects, chances are you will be repeating many of the same steps from Chapters 3-9. After all, every new project needs proper configuration, user authentication, and environment variables. So treat these chapters as your detailed explanation and guide. The second half of the book focuses on specific features related to our Bookstore website.

Chapter 10 starts with building out the models, tests, and pages for our Bookstore via a `Books` app. There is also a discussion of URLs and switching from `id` to a slug to a UUID (Universally Unique Identifier) in the URLs. *Chapter 11* features the addition of

reviews to our Bookstore and a discussion of foreign keys.

In *Chapter 12* image-uploading is added and in *Chapter 13* permissions are set across the site to lock it down. An ordering option is added in *Chapter 14* via Stripe. For any site but especially e-commerce, search is a vital component and *Chapter 15* walks through building a form and increasingly complex search filters for the site.

In *Chapter 16* the focus switches to performance optimizations including the addition of `django-debug-toolbar` to inspect queries and templates, database indexes, front-end assets, and multiple built-in caching options. *Chapter 17* covers security in Django, both the built-in options as well as additional configurations that can—and should—be added for a production environment. The final section, *Chapter 18*, is on deployment, the standard upgrades needed to migrate away from the Django web server, local static file handling, and configuring `ALLOWED_HOSTS`.

The Conclusion touches upon various next steps to take with the project and additional Django best practices.

Book Layout

There are many code examples in this book, which are formatted as follows:

Code

```
# This is Python code  
print(Hello, World)
```

For brevity we will use dots ... to denote existing code that remains unchanged, for example, in a function we are updating.

Code

```
def make_my_website:  
    ...  
    print("All done!")
```

We will also use the command line console frequently to execute commands, which take the form of a \$ prefix in traditional Unix style.

Command Line

```
$ echo "hello, world"
```

The result of this particular command in the next line will state:

Command Line

```
"hello, world"
```

Typically both a command and its output will be combined for brevity. The command will always be prefaced by a \$ and the output will not. For example, the command and result above would be represented as follows:

Command Line

```
$ echo "hello, world"  
hello, world
```

Text Editor

A modern text editor is a must-have part of any software developer's toolkit. Among other features they come with plug-ins that help format and correct errors in Python code. Popular options include [Black](#), [autopep8](#), and [YAPF](#).

Seasoned developers may still prefer using [Vim](#) or [Emacs](#), but newcomers and increasingly experienced programmers as well prefer modern text editors such as [VSCode](#), [Atom](#), [Sublime Text](#), or [PyCharm](#).

Conclusion

Django is an excellent choice for any developer who wants to build modern, robust web applications with a minimal amount of code. It is popular, under active development, and thoroughly battle-tested by the largest websites in the world.

Complete source code for the book can be found in the [official Github repository](#).

In the next chapter we'll learn how to configure any computer for Django development with Docker.

Chapter 1: Docker

Properly configuring a local development environment remains a steep challenge despite all the other advances in modern programming. There are simply too many variables: different computers, operating systems, versions of Django, virtual environment options, and so on. When you add in the challenge of working in a team environment where everyone needs to have the same set up the problem only magnifies.

In recent years a solution has emerged: [Docker](#). Although only a few years old, Docker has quickly become the default choice for many developers working on production-level projects.

With Docker it's finally possible to faithfully and dependably reproduce a production environment locally, everything from the proper Python version to installing Django and running additional services like a production-level database. This means it no longer matter if you are on a Mac, Windows, or Linux computer. Everything is running within Docker itself.

Docker also makes collaboration in teams exponentially easier. Gone are the days of sharing long, out-of-date `README` files for adding a new developer to a group project. Instead with Docker you simply share two files—a `Dockerfile` and `docker-compose.yml` file—and the developer can have confidence their local development environment is exactly the same as the rest of the team.

Docker is not a perfect technology. It is still relatively new, complex under-the-hood, and under active development. But the promise that it aspires to—a consistent and shareable developer environment, that can be run either locally on any computer or deployed to any server—makes it a solid choice.

In this chapter we'll learn a little bit more about Docker itself and "Dockerize" our first Django project.

What is Docker?

Docker is a way to isolate an entire operating system via Linux containers which are a type of [virtualization](#). Virtualization has its roots at the beginning of computer science when large, expensive mainframe computers were the norm. How could multiple programmers use the same single machine? The answer was virtualization and specifically [virtual machines](#) which are complete copies of a computer system from the operating system on up.

If you rent space on a cloud provider like [Amazon Web Services \(AWS\)](#) they are typically not providing you with a dedicated piece of hardware. Instead you are sharing one physical server with other clients. But because each client has their virtual machine running on the server, it appears to the client as if they have their own server.

This technology is what makes it possible to quickly add or remove servers from a cloud provider. It's largely software behind the scenes, not actual hardware being changed.

What's the downside to a virtual machine? Size and speed. A typical guest operating system can easily take up 700MB of size. So if one physical server supports three virtual machines, that's at least 2.1GB of disk space taken up along with separate needs for CPU and memory resources.

Enter Docker. The key idea is that most computers rely on the same [Linux](#) operating system, so what if we virtualized from [the Linux layer up](#) instead? Wouldn't that provide a lightweight, faster way to duplicate much of the same functionality? The answer is yes. And in recent years [Linux containers](#) have become widely popular. For most applications—especially web applications—a virtual machine provides far more

resources than are needed and a container is more than sufficient.

This, fundamentally, is what Docker is: a way to implement Linux containers!

An analogy we can use here is that of homes and apartments. Virtual Machines are like homes: stand-alone buildings with their own infrastructure including plumbing and heating, as well as a kitchen, bathrooms, bedrooms, and so on. Docker containers are like apartments: they share common infrastructure like plumbing and heating, but come in various sizes that match the exact needs of an owner.

Containers vs. Virtual Environments

As a Python programmer you should already familiar with the concept of virtual environments, which are a way to isolate Python packages. Thanks to virtual environments, one computer can run multiple projects locally. For example, Project A might use Python 3.4 and Django 1.11 among other dependencies; whereas Project B uses Python 3.7 and Django 2.2. By configuring a dedicated virtual environment for each project we can manage these different software packages while not polluting our global environment.

Confusingly there are multiple popular tools right now to implement virtual environments: everything from `virtualenv` to `venv` to `Pipenv`, but fundamentally they all do the same thing.

The important distinction between virtual environments and Docker is that virtual environments *can only isolate Python packages*. They cannot isolate non-Python software like a PostgreSQL or MySQL database. And they still rely on a global, system-level installation of Python (in other words, on your computer). The virtual environment points to an existing Python installation; it does not contain Python itself.

Linux containers go a step further and isolate the entire operating system, not just

the Python parts. In other words, we will install Python itself within Docker as well as install and run a production-level database.

Docker itself is a complex topic and we won't dive that deep into it in this book, however understanding its background and key components is important. If you'd like to learn more about it, I recommend the [Dive into Docker video course](#).

Install Docker

Ok, enough theory. Let's start using Docker and Django together. The first step is to sign up for a free account on [Docker Hub](#) and then install the Docker desktop app on your local machine:

- [Docker for Mac](#)
- [Docker for Windows](#)
- [Docker for Linux](#)

This download might take some time to download as it is a big file! Feel free to stretch your legs at this point.

Once Docker is done installing we can confirm the correct version is running by typing the command `docker --version` on the command line. It should be at least version 18.

Command Line

```
$ docker --version  
Docker version 18.09.2, build 6247962
```

Docker is often used with an additional tool, [Docker Compose](#), to help automate commands. Docker Compose is included with Mac and Windows downloads but if you are on Linux you will need to add it manually. You can do this by running the command `sudo pip install docker-compose` after your Docker installation is complete.

To confirm Docker Compose is correctly installed run the command `docker-compose --version`.

Command Line

```
$ docker-compose --version  
docker-compose version 1.23.2, build 1110ad01
```

Docker Hello, World

Docker ships with its own “Hello, World” image that is a helpful first step to run. On the command line type `docker run hello-world`. This will download an official Docker image and then run it within a container. We’ll discuss both images and containers in a moment.

Command Line

```
$ docker run hello-world  
Unable to find image 'hello-world:latest' locally  
latest: Pulling from library/hello-world  
1b930d010525: Pull complete  
Digest: sha256:6540fc08ee6e6b7b63468dc3317e3303aae178cb8a45ed3123180328bcc1d20f  
Status: Downloaded newer image for hello-world:latest
```

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
(amd64)

3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID:

<https://hub.docker.com/>

For more examples and ideas, visit:

<https://docs.docker.com/get-started/>

The command `docker info` lets us inspect Docker. It will contain a lot of output but focus on the top lines which show we now have 1 container which is stopped and 1 image.

Command Line

```
$ docker info
```

Containers: 1

 Running: 0

 Paused: 0

 Stopped: 1

Images: 1

...

This means Docker is successfully installed and running.

Django Hello, World

Now we will create a Django “Hello, World” project that runs locally on our computer and then move it entirely within Docker so you can see how all the pieces fit together.

The first step is to choose a location for our code. This can be anywhere on your computer, but if you are on a Mac, an easy-to-find location is the `Desktop`. From the command line navigate to the `Desktop` and create a `code` directory for all the code examples in this book.

Command Line

```
$ cd ~/Desktop  
$ mkdir code && cd code
```

Then create a `hello` directory for this example and install Django using Pipenv which creates both a `Pipfile` and a `Pipfile.lock` file. Activate the virtual environment with the `shell` command.

Command Line

```
$ mkdir hello && cd hello  
$ pipenv install django==2.2.3  
$ pipenv shell  
(hello) $
```

If you need help installing Pipenv or Python 3 you can [find more details here](#).

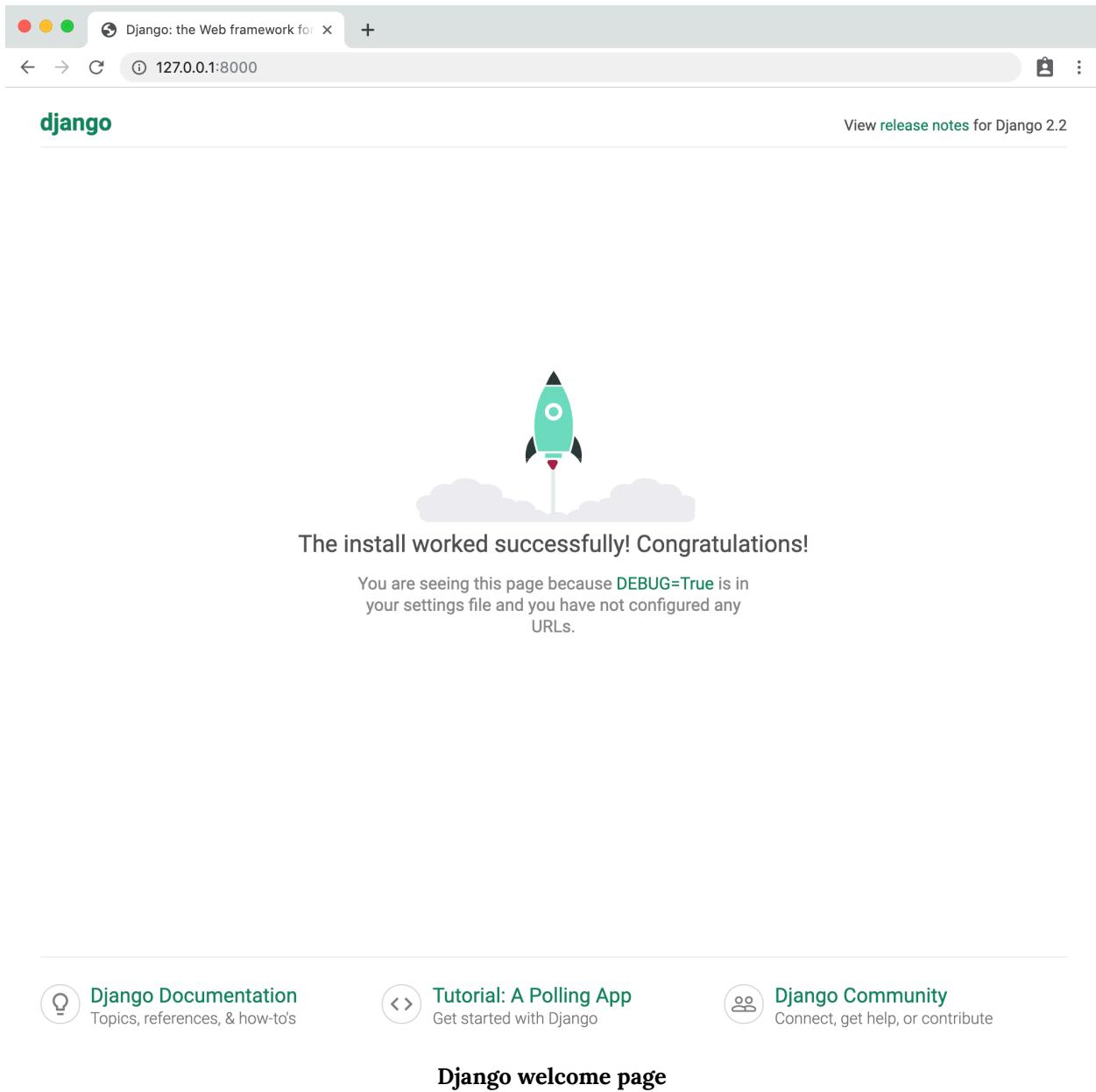
Now we can use the `startproject` command to create a new Django project called `hello_project`. Don’t forget include the period, `..`, at the end there! If you don’t then `startproject` will add an additional directory to the project. Finally use the `migrate`

command to initialize the database and start the local web server with the `runserver` command.

Command Line

```
(hello) $ django-admin startproject hello_project .
(hello) $ python manage.py migrate
(hello) $ python manage.py runserver
```

Assuming everything worked correctly you should now be able to navigate to see the Django Welcome page at <http://127.0.0.1:8000/> in your web browser.



Pages App

Now we will make a simple homepage by creating a dedicated `pages` app for it. Stop the local server by typing `Control+c` and then use the `startapp` command appending our desired `pages` name.

Command Line

```
(hello) $ python manage.py startapp pages
```

Django automatically installs a new `pages` directory and several files for us. But even though the app has been created our Bookstore project won't recognize it until we add it to the `INSTALLED_APPS` config within the `hello_project/settings.py` file.

Django loads apps from top to bottom so generally speaking it's a good practice to add new apps below built-in apps they might rely on such as `admin`, `auth`, and all the rest.

Note that while it is possible to simply type the name of the app, `pages`, you are better off typing the full `pages.apps.PagesConfig` which opens up more possibilities in configuring `apps`.

Code

```
# hello_project/settings.py

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'pages.apps.PagesConfig', # new
]
```

Now we can set the URL route for the `pages` app. Since we want our message to appear on the homepage we'll use the empty string `' '`. Don't forget to add the `include` import on the second line as well.

Code

```
# hello_project/urls.py

from django.contrib import admin
from django.urls import path, include # new

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('pages.urls')), # new
]
```

Rather than set up a template at this point we can just hardcode a message in our view layer at `pages/views.py` which will output the string “Hello, World!”.

Code

```
# pages/views.py

from django.http import HttpResponse

def HomePageView(request):
    return HttpResponse('Hello, World!')
```

What's next? Our last step is to create a `urls.py` file within the `pages` app and link it to `HomePageView`. Let's create the file. If you are on an Apple computer the `touch` command can be used from the command line to create new files. If on Windows or Linux, create new files within your text editor.

Command Line

```
(hello) $ touch pages/urls.py
```

Within your text editor import `path` on the top line, add the `homePageView`, and then set its route to again be the empty string of '''. Note that we also provide an optional name, `home`, for this route which is a best practice.

Code

```
# pages/urls.py

from django.urls import path

from .views import homePageView

urlpatterns = [
    path('', homePageView, name='home')
]
```

The full flow of our Django homepage is as follows:

- * when a user goes to the homepage they will first be routed to `hello_project/urls.py`
- * then routed to `pages/urls.py`
- * and finally directed to the `homePageView` which returns the string “Hello, World!”

Our work is done for a basic homepage. Start up the local server again.

Command Line

```
(hello) $ python manage.py runserver
```

If you refresh the web browser at <http://127.0.0.1:8000/> it will now output our desired message.



Hello World

Now it's time to switch to Docker. Stop the local server again with `Control+c` and exit our virtual environment since we no longer need it by typing `exit`.

Command Line

```
(hello) $ exit  
$
```

How do we know the virtual environment is no longer active? There will no longer be parentheses around the directory name on the command line prompt. Any normal Django commands you try to run at this point will fail. For example, try `python manage.py runserver` to see what happens.

Command Line

```
$ python manage.py runserver  
File "./manage.py", line 14  
    ) from exc  
      ^  
SyntaxError: invalid syntax
```

This means we're fully out of the virtual environment and ready for Docker.

Images, Containers, and the Docker Host

A Docker **image** is a snapshot in time of what a project contains. It is represented by a **Dockerfile** and is literally a list of instructions that must be built. A Docker **container** is a running instance of an image. To continue our apartment analogy from earlier, the image is the blueprint or set of plans for the apartment; the container is the actual, fully-built building.

The third core concept is the “Docker host” which is the underlying OS. It’s possible to have multiple containers running within a single Docker host. When we refer to code or processes running *within* Docker, that means they are running in the Docker host.

Let’s create our first **Dockerfile** to see all of this theory in action.

Command Line

```
$ touch Dockerfile
```

Within the **Dockerfile** add the following code which we’ll walk through line-by-line below.

Dockerfile

```
# Pull base image
FROM python:3.7-slim

# Set environment variables
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# Set work directory
WORKDIR /code
```

```
# Install dependencies  
COPY Pipfile Pipfile.lock /code/  
RUN pip install pipenv && pipenv install --system  
  
# Copy project  
COPY . /code/
```

Dockerfiles are read from top-to-bottom when an image is created. The first instruction **must** be the `FROM` command which lets us import a base image to use for our image. We're using the `slim` version of Python 3.7 because it is far smaller in size than the complete `python:3.7` image yet contains all the functionality we need. Images can become quite large in size if you don't monitor this!

Then we use the `ENV` command to set two environment variables:

- `PYTHONUNBUFFERED` ensures our console output looks familiar and is not buffered by Docker, which we don't want
- `PYTHONDONTWRITEBYTECODE` means Python won't try to write `.pyc` files which we also do not desire

Next we use `WORKDIR` to set a default work directory path called `code` which is where we will store our code. If we didn't do this then each time we wanted to execute commands within our container we'd have to type in a long path. Instead Docker will just assume we mean to execute all commands from this directory.

For our dependencies we are using `Pipenv` so we copy over both the `Pipfile` and `Pipfile.lock` files into a `/code/` directory in Docker.

It's worth taking a moment to explain why `Pipenv` creates a `Pipfile.lock`, too. The concept of lock files is not unique to Python or `Pipenv`; in fact it already present in

package managers for most modern programming languages: `Gemfile.lock` in Ruby, `yarn.lock` in JavaScript, `composer.lock` in PHP, and so on. `Pipenv` was the first popular project to incorporate them into Python packaging.

The benefit of a lock file is that this leads to a deterministic build: no matter how many times you install the software packages, you'll have the same result. Without a lock file that "locks down" the dependencies and their order, this is not necessarily the case. Which means that two team members who install the same list of software packages might have slightly different build installations.

When we're working with Docker where there is code both locally on our computer and also within Docker, the potential for `Pipfile.lock` conflicts arises when updating software packages. We'll explore this properly in the next chapter.

Moving along we use the `RUN` command to first install `Pipenv` and then `pipenv install` to install the software packages listed in our `Pipfile.lock`, currently just Django. It's important to add the `--system` flag as well since by default `Pipenv` will look for a virtual environment in which to install any package, but since we're within Docker now, technically there isn't any virtual environment. In a way, the Docker container is our virtual environment and more. So we must use the `--system` flag to ensure our packages are available throughout all of Docker for us.

As the final step we copy over the rest of our local code into the `/code/` directory within Docker. Why do we copy local code over twice, first the `Pipfile` and `Pipfile.lock` and then the rest? The reason is that images are created based on instructions top-down so we want things that change often-like our local code-to be last. That way we only have to regenerate that part of the image when a change happens, not reinstall everything each time there is a change. And since the software packages contained in our `Pipfile` and `Pipfile.lock` change infrequently, it makes sense to copy them over and install them earlier.

Our image instructions are now done so let's build the image using the command

`docker build .` The period, `.`, indicates the current directory is where to execute the command. There will be a lot of output here; I've only included the first two lines and the last three.

Command Line

```
$ docker build .
Sending build context to Docker daemon 155.1kB
Step 1/7 : FROM python:3.7-slim
...
Step 7/7 : COPY . /code/
--> d26473003a8d
Successfully built d26473003a8d
```

Moving on we now need to create a `docker-compose.yml` file to control how to run the container that will be built based upon our `Dockerfile` image.

Command Line

```
$ touch docker-compose.yml
```

It will contain the following code.

docker-compose.yml

```
version: '3.7'

services:
  web:
    build: .
    command: python /code/manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/code
    ports:
      - 8000:8000
```

On the top line we specify the **most recent version** of Docker Compose which is currently **3.7**. Don't be confused by the fact that Python is also on version **3.7** at the moment; there is no overlap between the two! It's just a coincidence.

Then we specify which **services** (or containers) we want to have running within our Docker host. It's possible to have multiple **services** running, but for now we just have one for **web**. We specify how to build the container by saying, Look in the current directory **.** for the **Dockerfile**. Then within the container run the **command** to start up the local server.

The **volumes** mount automatically syncs the Docker filesystem with our local computer's filesystem. This means that we don't have to rebuild the image each time we change a single file!

Lastly we specify the **ports** to expose within Docker which will be **8000**, which is the Django default.

If this is your first time using Docker, it is *highly likely* you are confused right now. But don't worry. We'll create multiple Docker images and containers over the course

of this book and with practice the flow will start to make more sense. You'll see we use very similar `Dockerfile` and `docker-compose.yml` files in each of our projects.

The final step is to run our Docker container using the command `docker-compose up`. This command will result in another long stream of output code on the command line.

Command Line

```
$ docker-compose up  
Creating network "hello_default" with the default driver  
Building web  
Step 1/7 : FROM python:3.7-slim  
...  
Creating hello_web_1 ... done  
Attaching to hello_web_1  
web_1  | Performing system checks...  
web_1  |  
web_1  | System check identified no issues (0 silenced).  
web_1  | July 13, 2019 - 16:47:03  
web_1  | Django version 2.2.3, using settings 'hello_project.settings'  
web_1  | Starting development server at http://0.0.0.0:8000/  
web_1  | Quit the server with CONTROL-C.
```

To confirm it actually worked, go back to <http://127.0.0.1:8000/> in your web browser. Refresh the page and the Django welcome page should still appear.

Django is now running purely within a Docker container. We are not working within a virtual environment locally. We did not execute the `runserver` command. All of our code now exists and our Django server is running within a self-contained Docker container. Success!

Stop the container with `control+c` and additionally type `docker-compose down`. Docker containers take up a lot of memory so it's a good idea to stop them in this way when

you're done using them. Containers are meant to be stateless which is why we use volumes to copy our code over locally where it can be saved.

Command Line

```
$ docker-compose down  
Removing hello_web_1 ... done  
Removing network hello_default
```

Git

Git is the version control system of choice these days and we'll use it in this book. First add a new Git file with `git init`, then check the `status` of changes, add updates, and include a commit message.

Command Line

```
$ git init  
$ git status  
$ git add .  
$ git commit -m 'ch1'
```

You can compare your code for this chapter with the [official repository](#) available on Github.

Conclusion

Docker is a self-contained environment that includes everything we need for local development: web services, databases, and more if we want. The general pattern will always be the same when using it with Django:

- create a virtual environment locally and install Django
- create a new project
- exit the virtual environment
- write a `Dockerfile` and then build the initial image
- write a `docker-compose.yml` file and run the container with `docker-compose up`

We'll build several more Django projects with Docker so this flow makes more sense, but that's really all there is to it. In the next chapter we'll create a new Django project using Docker and add PostgreSQL in a separate container as our database.

Chapter 2: PostgreSQL

One of the most immediate differences between working on a “toy app” in Django and a production-ready one is the database. Django ships with [SQLite](#) as the default choice for local development because it is small, fast, and file-based which makes it easy to use. No additional installation or configuration is required.

However this convenience comes at a cost. Generally speaking SQLite is not a good database choice for professional websites. So while it is fine to use SQLite locally while prototyping an idea, it is rare to actually use SQLite as the database on a production project.

Django ships with built-in support for [four databases](#): SQLite, PostgreSQL, MySQL, and Oracle. We’ll be using [PostgreSQL](#) in this book as it is the most popular choice for Django developers, however, the beauty of Django’s ORM is that even if we wanted to use MySQL or Oracle, the actual Django code we write will be almost identical. The Django ORM handles the translation from Python code to the databases for us which is quite amazing if you think about it.

The challenge of using these three databases is that each must be both installed and run locally if you want to faithfully mimic a production environment on your local computer. And we do want that! While Django handles the details of switching between databases for us there are inevitably small, hard-to-catch bugs that can crop up if you use SQLite for local development but a different database in production. Therefore a best practice is use the same database locally and in production.

In this chapter we’ll start a new Django project with a SQLite database and then switch over to both Docker and PostgreSQL.

Starting

On the command line make sure you've navigated back to the `code` folder on our desktop. You can do this two ways. Either type `cd ..` to move "up" a level so if you are currently in `Desktop/code/hello` you will move to `Desktop/code`. Or you can simply type `cd ~/Desktop/code/` which will take you directly to the desired directory. Then create a new directory called `postgresql` for this chapter's code.

Command Line

```
$ cd ..  
$ mkdir postgresql && cd postgresql
```

Now install Django, start the shell, and create a basic Django project called `postgresql-project`. Don't forget the period `.` at the end of the command!

Command Line

```
$ pipenv install django==2.2.3  
$ pipenv shell  
(postgresql) $ django-admin startproject postgresql_project .
```

So far so good. Now we can `migrate` our database to initialize it and use `runserver` to start the local server.

Normally I don't recommend running `migrate` on new projects until *after* a custom user model has been configured. Otherwise Django will bind the database to the built-in `User` model which is difficult to modify later on in the project. We'll cover this properly in Chapter 3 but since this chapter is primarily for demonstration purposes, using the default `User` model here is a one-time exception.

Command Line

```
(postgresql) $ python manage.py migrate  
(postgresql) $ python manage.py runserver
```

Confirm everything worked by navigating to <http://127.0.0.1:8000/> in your web browser. You may need to refresh the page but should see the familiar Django welcome page.

Stop the local server with `Control+c` and then use the `ls` command to list all files and directories.

Command Line

```
(postresql) $ ls  
Pipfile      Pipfile.lock      db.sqlite3      manage.py      postgresql_project
```

Docker

To switch over to Docker first `exit` our virtual environment and then create `Dockerfile` and `docker-compose.yml` files which will control our Docker image and container respectively.

Command Line

```
(postgresql) $ exit  
$ touch Dockerfile  
$ touch docker-compose.yml
```

The `Dockerfile` is the same as in Chapter 1.

Dockerfile

```
# Pull base image
FROM python:3.7-slim

# Set environment variables
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# Set work directory
WORKDIR /code

# Install dependencies
COPY Pipfile Pipfile.lock /code/
RUN pip install pipenv && pipenv install --system

# Copy project
COPY . /code/
```

Go ahead and build the initial image now using the `docker build .` command.

Command Line

```
$ docker build .  
Sending build context to Docker daemon 146.4kB  
Step 1/7 : FROM python:3.7-slim  
--> ba83e35afc23  
Step 2/7 : ENV PYTHONDONTWRITEBYTECODE 1  
--> Using cache  
--> accdccfdb2ff  
Step 3/7 : ENV PYTHONUNBUFFERED 1  
--> Using cache  
--> 6d16b0644a2a  
Step 4/7 : WORKDIR /code  
--> Using cache  
--> 1e879264a847  
Step 5/7 : COPY Pipfile Pipfile.lock /code/  
--> Using cache  
--> 72aa4b9e71c9  
Step 6/7 : RUN pip install pipenv && pipenv install --system  
--> Using cache  
--> 92f3b2b2044a  
Step 7/7 : COPY . /code/  
--> 38e9b8855d9d  
Successfully built 38e9b8855d9d
```

Your Docker image IDs might look slightly different than mine. For example the ID of the `python:3.7-slim` image is `ba83e35afc23` at this moment in time, but that image will be updated at some point in the future and the corresponding ID change as well. But these are largely minor changes so as long as you see `Successfully built` on the last line you are in good shape.

Did you notice that the `Dockerfile` built an image much faster this time around? That's because Docker looks locally on your computer first for a specific image. If it doesn't find an image locally it will then download it. And since many of these images were already on the computer from the previous chapter, Docker didn't need to download them all again!

Time now for the `docker-compose.yml` file which also matches what we saw previously in Chapter 1.

docker-compose.yml

```
version: '3.7'

services:
  web:
    build: .
    command: python /code/manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/code
    ports:
      - 8000:8000
```

Detached Mode

We'll start up our container now but this time in detached mode which requires either the `-d` or `--detached` flag (they do the same thing).

Command Line

```
$ docker-compose up -d
```

Detached mode runs [containers in the background](#), which means we can use a single command line tab without needing a separate one open as well. This saves us from switching back and forth between two command line tabs constantly. The downside is that if/when there is an error, the output won't always be visible. So if your screen does not match this book at some point, try typing `docker-compose logs` to see the current output and debug any issues.

You likely will see a “Warning: Image for service web was built because it did not already exist” message at the bottom of the command. Docker automatically created a new image for us within the container. As we'll see later in the book, adding the `--build` flag to force an image build is necessary when software packages are updated because, by default, Docker will look for a local cached copy of software and use that which improves performance.

To confirm things are working properly go back to <http://127.0.0.1:8000/> in your web browser. Refresh the page to see the Django welcome page again.

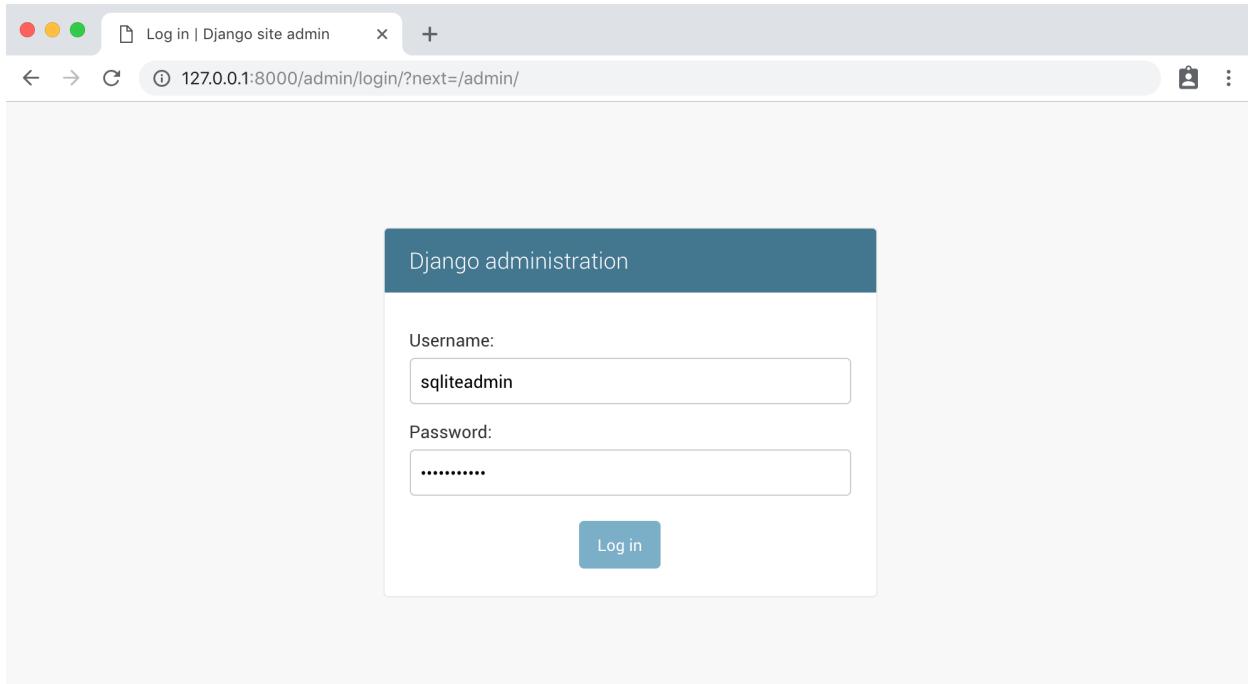
Since we're working *within Docker* now as opposed to locally we must preface traditional commands with `docker-compose exec web`. For example, to create a superuser account instead of typing `python manage.py createsuperuser` the updated command would now look like the line below.

Command Line

```
$ docker-compose exec web python manage.py createsuperuser
```

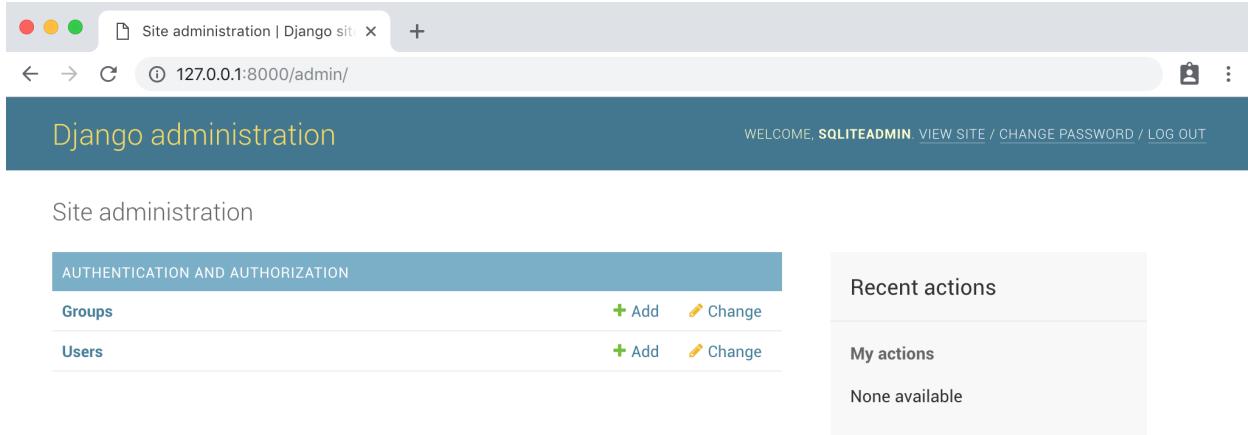
For the username choose `sqliteadmin`, `sqliteadmin@email.com` as the email address, and select the password of your choice. I often use `testpass123`.

Then navigate directly into the admin at <http://127.0.0.1:8000/admin> and log in.



Django admin login

You will be redirected to the admin homepage. Note in the upper right corner `sqliteadmin` is the username.



Django sqliteadmin

If you click on the `Users` button it takes us to the Users page where we can confirm only one user has been created.

The screenshot shows the Django administration interface for managing users. The title bar says "Select user to change | Django". The URL is "127.0.0.1:8000/admin/auth/user/". The main header says "Django administration" and "WELCOME, SQLITEADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT". Below it, the breadcrumb navigation shows "Home > Authentication and Authorization > Users". The main content area is titled "Select user to change" and contains a search bar and a table with one row. The table has columns: USERNAME, EMAIL ADDRESS, FIRST NAME, LAST NAME, and STAFF STATUS. The row for "sqliteadmin" has a checked checkbox in the first column, and a green checkmark in the STAFF STATUS column. To the right of the table is a "FILTER" sidebar with sections for "By staff status" (All, Yes, No), "By superuser status" (All, Yes, No), and "By active" (All, Yes, No). A "ADD USER +" button is at the top right of the content area.

	USERNAME	EMAIL ADDRESS	FIRST NAME	LAST NAME	STAFF STATUS
<input type="checkbox"/>	sqliteadmin	sqliteadmin@email.com			

Admin Users page

It's important to highlight another aspect of Docker at this point: so far we've been updating our database—currently represented by the `db.sqlite3` file—within Docker. That means the actual `db.sqlite3` file is changing each time. And thanks to the `volumes` mount in our `docker-compose.yml` config each file change has been copied over into a `db.sqlite3` file on our local computer too. You could quit Docker, start the `shell`, and see the exact same admin login at this point because the underlying SQLite database is the same.

PostgreSQL

Now it's time to switch over to PostgreSQL for our project which takes three additional steps:

- install a database adapter, `psycopg2`, so Python can talk to PostgreSQL
- update the `DATABASE` config in our `settings.py` file
- install and run PostgreSQL locally

Ready? Here we go. Stop the running Docker container with `docker-compose down`.

Command Line

```
$ docker-compose down
Stopping postgresql_web_1 ... done
Removing postgresql_web_1 ... done
Removing network postgresql_default
```

Then within our `docker-compose.yml` file add a new service called `db`. This means there will be two separate services, each a container, running within our Docker host: `web` for the Django local server and `db` for our PostgreSQL database.

The PostgreSQL version will be pinned to the latest version, `11`. If we had not specified a version number and instead used just `postgres` then the latest version of PostgreSQL would be downloaded even if at a later date that is Postgres 12 which will likely have different requirements.

Finally we add a `depends_on` line to our `web` service since it literally depends on the database to run. This means that `db` will be started up before `web`.

docker-compose.yml

```
version: '3.7'

services:
  web:
    build: .
    command: python /code/manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/code
    ports:
      - 8000:8000
    depends_on:
      - db
  db:
    image: postgres:11
```

Now run `docker-compose up -d` which will rebuild our image and spin up two containers, one running PostgreSQL within `db` and the other our Django `web` server.

Command Line

```
$ docker-compose up -d
Creating network "postgresql_default" with the default driver
...
Creating postgresql_db_1 ... done
Creating postgresql_web_1 ... done
```

It's important to note at this point that a production database like PostgreSQL is not file-based. It runs entirely within the `db` service and is ephemeral; when we execute

`docker-compose down` all data within it will be lost. This is in contrast to our code in the `web` container which has a `volumes` mount to sync local and Docker code.

In the next chapter we'll learn how to add a `volumes` mount for our `db` service to persist our database information.

Settings

With your text editor, open the `postgresql_project/settings.py` file and scroll down to the `DATABASES` config. The current setting is this:

Code

```
# postgresql_project/settings.py

DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

By default Django specifies `sqlite3` as the database engine, gives it the name `db.sqlite3`, and places it at `BASE_DIR` which means in our project-level directory.

Since directory structure is often a point of confusion “project-level” means the top directory of our project which contains `postgresql_project`, `manage.py`, `Pipfile`, `Pipfile.lock`, and the `db.sqlite3` file.

Command Line

```
(postgresql) $ ls
Dockerfile  Pipfile.lock  docker-compose.yml  postgresql_project
Pipfile  db.sqlite3  manage.py
```

To switch over to PostgreSQL we will update the [ENGINE](#) configuration. PostgreSQL requires a NAME, USER, PASSWORD, HOST, and PORT.

For convenience we'll set the first three to `postgres`, the HOST to `db` which is the name of our service set in `docker-compose.yml`, and the PORT to `5432` which is the default PostgreSQL [port](#).

Code

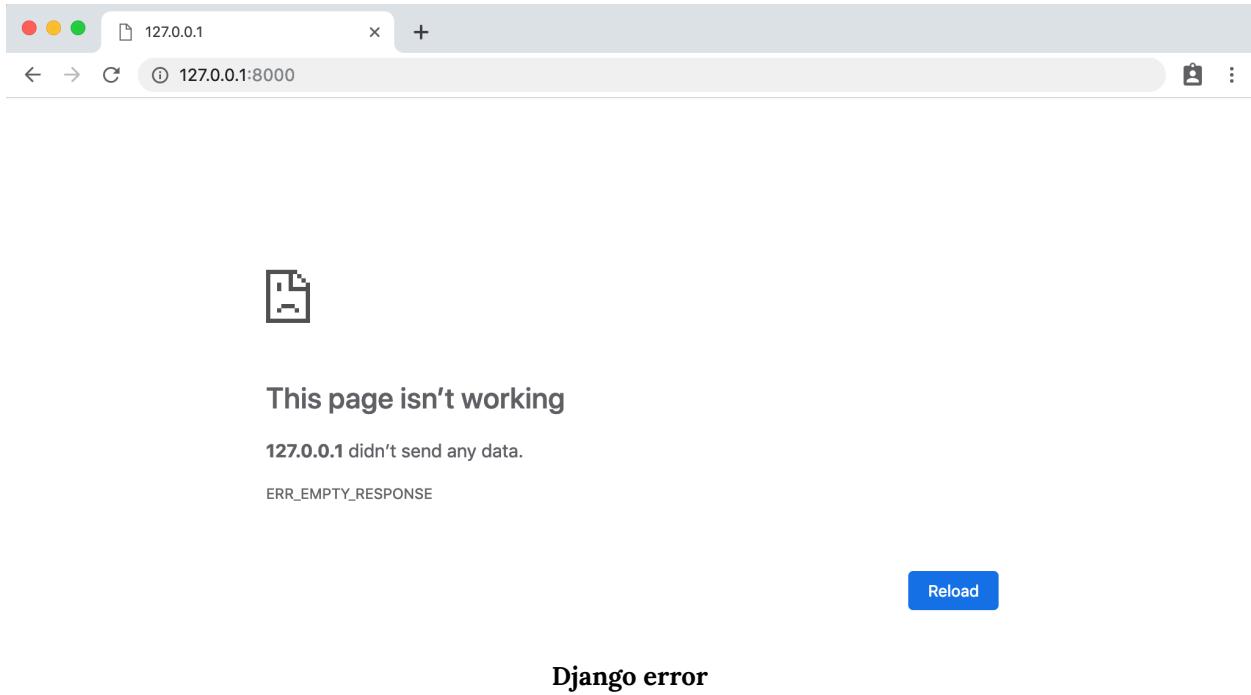
```
# postgresql_project/settings.py

DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'postgres',
        'USER': 'postgres',
        'PASSWORD': 'postgres',
        'HOST': 'db',
        'PORT': 5432
    }
}
```

You will see an error now if you refresh the web page.



What's happening? Since we're running Docker in detached mode with the `-d` flag it's not immediately clear. Time to check our logs.

Command Line

```
$ docker-compose logs  
...  
web_1 | django.core.exceptions.ImproperlyConfigured: Error loading psycopg2  
module: No module named 'psycopg2'
```

There will be a lot of output but at the bottom of the `web_1` section you'll see the above lines which tells us we haven't installed the `psycopg2` driver yet.

Pyscopg

PostgreSQL is a database that can be used by almost any programming language. But if you think about it, how does a programming language—and they all vary in some

way or another—connect to the database itself?

The answer is via a database adapter! And that's what [Psycopg](#) is, the most popular database adapter for Python. If you'd like to learn more about how Psycopg works here is a link to [a fuller description](#) on the official site.

We can install Psycopg with Pipenv. On the command line, enter the following command so it is installed within our Docker host.

Command Line

```
$ docker-compose exec web pipenv install psycopg2-binary==2.8.3
```

Why install within Docker rather than locally I hope you're asking? The short answer is that consistently installing new software packages within Docker and then rebuilding the image from scratch will save us from potential `Pipfile.lock` conflicts.

The `Pipfile.lock` generation depends heavily on the OS being used. We've specified our entire OS within Docker, including using `Python 3.7-slim`. But if you install `psycopg2` locally on your computer, which has a different environment, the resulting `Pipfile.lock` file will also be different. But then the `volumes` mount in our `docker-compose.yml` file, which automatically syncs the local and Docker filesystems, will cause the local `Pipfile.lock` to overwrite the version within Docker. So now our Docker container is trying to run an incorrect `Pipfile.lock` file. Ack!

One way to avoid these issues is to consistently install new software packages within Docker rather than locally.

If you now refresh the webpage you will...still see an error. Ok, let's check the logs.

Command Line

```
$ docker-compose logs
```

It's the same as before! Why does this happen? Docker automatically caches images unless something changes for performance reasons. We want it to automatically rebuild the image with our new `Pipfile` and `Pipfile.lock` but because the last line of our `Dockerfile` is `COPY . /code/` only the files will copy; the underlying image won't rebuild itself unless we force it too. This can be done by adding the `--build` flag.

So to review: whenever adding a new package first install it within Docker, stop the containers, force an image rebuild, and then start the containers up again. We'll use this flow repeatedly throughout the book.

Command Line

```
$ docker-compose down  
$ docker-compose up -d --build
```

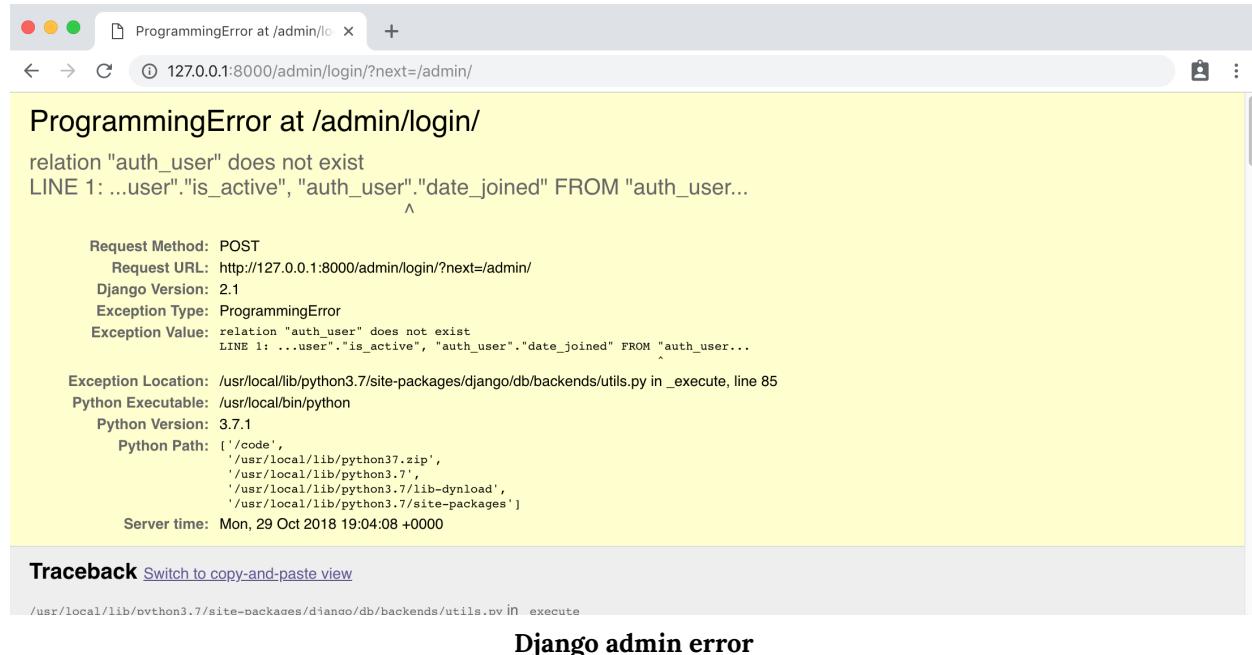
If you refresh the homepage again the Django welcome page at <http://127.0.0.1:8000/> now works! That's because Django has successfully connected to PostgreSQL via Docker.

Great, everything is working.

New Database

However since we are using PostgreSQL now, not SQLite, our database is empty. If you look at the current logs again by typing `docker-compose logs` you'll see complaints like "You have 15 unapplied migrations(s)".

To reinforce this point visit the Admin at <http://127.0.0.1:8000/admin/> and log in. Will our previous superuser account of `sqliteadmin` and `testpass123` work?



Nope! We see `ProgrammingError` at `/admin` which complains that `auth_user` doesn't even exist because we have not done a migration yet! Also, we don't have a superuser either on our PostgreSQL database.

To fix this situation we can both migrate and create a superuser within Docker that will access the PostgreSQL database.

Command Line

```
$ docker-compose exec web python manage.py migrate
$ docker-compose exec web python manage.py createsuperuser
```

What should we call our superuser? Let's use `postgresadmin` and for testing purposes set the email to `postgresadmin@email.com` and the password to `testpass123`.

In your web browser navigate to the admin page at <http://127.0.0.1:8000/admin/> and enter in the new superuser log in information.

The screenshot shows the Django administration interface at the URL `127.0.0.1:8000/admin/`. The top navigation bar includes links for Site administration, Log out, and Help. The main title is "Django administration". On the left, there's a sidebar titled "AUTHENTICATION AND AUTHORIZATION" with tabs for "Groups" and "Users". Under "Groups", there are "Add" and "Change" buttons. Under "Users", there are "Add" and "Change" buttons. To the right, there are two sections: "Recent actions" (empty) and "My actions" (also empty). At the bottom center, it says "Admin with postgresadmin".

In the upper right corner it shows that we are logged in with `postgresadmin` now not `sqliteadmin`. Also you can click on the `Users` tab on the homepage and visit the `Users` section to see our one and only user is the new superuser account.

The screenshot shows the Django administration interface for managing users. The top navigation bar includes links for 'Select user to change' and '127.0.0.1:8000/admin/auth/user/'. The main title is 'Django administration' with a welcome message for 'POSTGRESADMIN'. Below it, a breadcrumb trail shows 'Home > Authentication and Authorization > Users'. A search bar and a 'Search' button are at the top left. On the right, there's a 'ADD USER +' button. The central area displays a table with one user entry:

<input type="checkbox"/>	USERNAME	EMAIL ADDRESS	FIRST NAME	LAST NAME	STAFF STATUS
<input type="checkbox"/>	postgresadmin	postgresadmin@email.com			<input checked="" type="checkbox"/>

Below the table, it says '1 user'. To the right, a 'FILTER' sidebar provides options to filter by staff status (All, Yes, No), superuser status (All, Yes, No), and active status (All, Yes, No). At the bottom center, the text 'Admin users' is displayed.

Remember to stop our running container with `docker-compose down`.

Command Line

```
$ docker-compose down
```

Git

Let's save our changes again by initializing Git for this new project, adding our changes, and including a commit message.

Command Line

```
$ git init  
$ git status  
$ git add .  
$ git commit -m 'ch2'
```

The official source code for Chapter 2 is available on [Github](#).

Conclusion

The goal of this chapter was to demonstrate how Docker and PostgreSQL work together on a Django project. Switching between a SQLite database and a PostgreSQL is a mental leap for many developers initially.

The key point is that with Docker we don't need to be in a local virtual environment anymore. Docker is our virtual environment...and our database and more if desired. The Docker host essentially replaces our local operating system and within it we can run multiple containers, such as for our web app and for our database, which can all be isolated and run separately.

In the next chapter we will start our online Bookstore project. Let's begin!

Chapter 3: Bookstore Project

It is time to build the main project of this book, an online Bookstore. In this chapter we will start a new project, switch over to Docker, add a custom user model, and implement our first tests.

Let's start by creating a new Django project with Pipenv locally and then switch over to Docker. You're likely in the `postgresql` directory right now from Chapter 2 so on the command line type `cd ..` which will take you back to the desired `code` directory on the Desktop (assuming you're on a Mac). We'll create a `books` directory for our code, and then install `django`. We also know we'll be using PostgreSQL so we can install the `psycopg2` adapter now too. It is only *after* we have built our initial image that we start installing future software packages within Docker itself. Lastly use the `shell` command to enter the new virtual environment.

Command Line

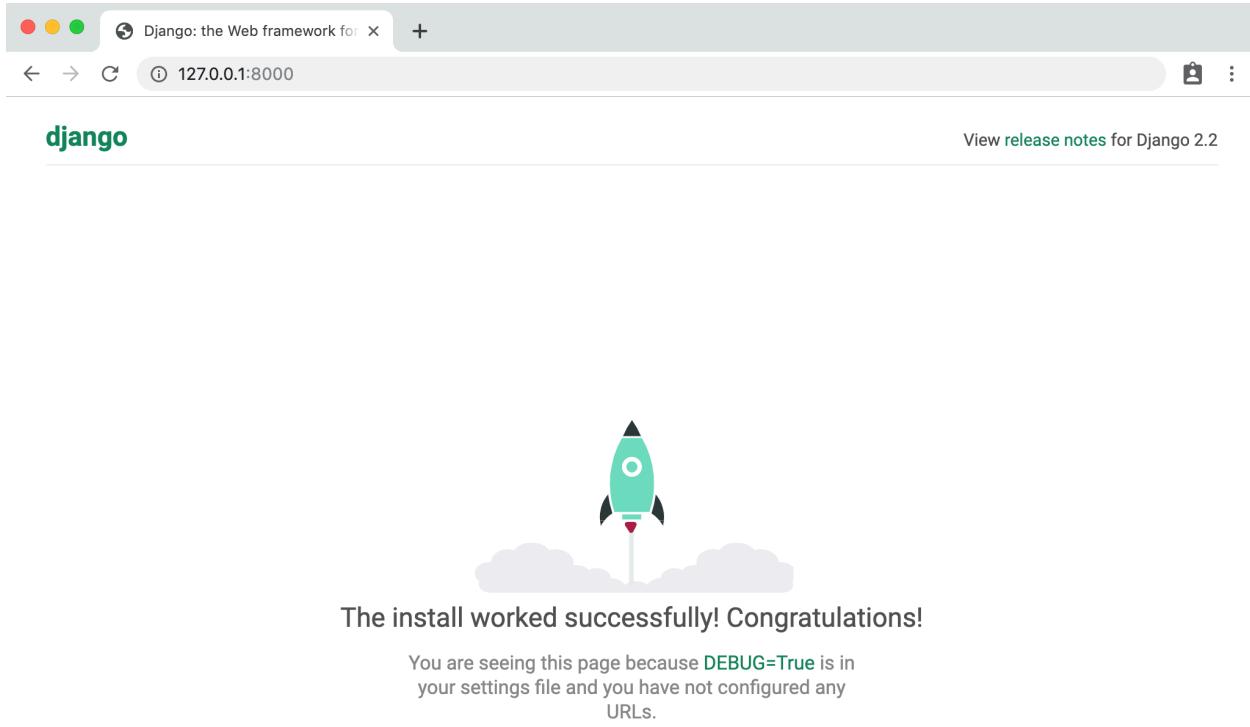
```
$ cd ..  
$ mkdir books && cd books  
$ pipenv install django==2.2.3 psycopg2-binary==2.8.3  
$ pipenv shell
```

We'll name our new Django project `bookstore_project`. Make sure you don't forget that period, `.`, at the end of the command or else Django will create an extra directory which we don't need. Then use `runserver` to start the local Django web server and confirm everything is working correctly.

Command Line

```
(books) $ django-admin startproject bookstore_project .  
(books) $ python manage.py runserver
```

In your web browser go to <http://127.0.0.1:8000/> and you should see the friendly Django welcome page.



[Django Documentation](#)
Topics, references, & how-to's



[Tutorial: A Polling App](#)
Get started with Django



[Django Community](#)
Connect, get help, or contribute

Django welcome page

On the command line you will likely see a warning about “unapplied migration(s)”. It’s safe to ignore this for now since we’re about to switch over to Docker and PostgreSQL.

Docker

We can now switch over to Docker in our project. Go ahead and stop the local server `Control+c` and also exit the virtual environment shell.

Command Line

```
(books) $ exit
```

```
$
```

Docker should already be installed and the desktop app running from the previous chapter. Per usual we need to create a `Dockerfile` and `docker-compose.yml` file.

Command Line

```
$ touch Dockerfile
```

```
$ touch docker-compose.yml
```

The `Dockerfile` will be the same as before.

Dockerfile

```
# Pull base image
FROM python:3.7-slim

# Set environment variables
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# Set work directory
WORKDIR /code

# Install dependencies
COPY Pipfile Pipfile.lock /code/
RUN pip install pipenv && pipenv install --system

# Copy project
COPY . /code/
```

But for the `docker-compose.yml` file we'll add an additional feature which is a dedicated volume for our database so it persists even when the `services` containers are stopped. Removing the volume itself is a separate process.

We can do this by specifying a path for `volumes` within the `db` container and then specifying a `volumes` outside of our services with the same name `postgres_data`. You can see the [Docker documentation on volumes](#) for a more technical explanation of how this all works if you're interested.

docker-compose.yml

```
version: '3.7'

services:
  web:
    build: .
    command: python /code/manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/code
    ports:
      - 8000:8000
    depends_on:
      - db
  db:
    image: postgres:11
    volumes:
      - postgres_data:/var/lib/postgresql/data/
volumes:
  postgres_data:
```

We can build our image and run the containers with one command.

Command Line

```
$ docker-compose up -d --build
```

If you see an error here like `Bindfor 0.0.0.0:8000 failed: port is already allocated` then then you did not fully stop the Docker container from Chapter 2. Try running `docker-compose down` in the directory where you previously ran it, probably `postgresql`.

Then attempt to build and run our new image and container again. If that approach still fails you can quit the Docker desktop application completely and then open it again.

Go to the web browser now at <http://127.0.0.1:8000/> and click refresh. It should be the same friendly Django welcome page albeit now running inside of Docker.

PostgreSQL

Even though we already installed `psycopg` and have PostgreSQL available in our `docker-compose.yml` file we still must direct Django to switch over to it instead of the default SQLite database. Do that now. The code is the same as in the previous chapter.

Code

```
# bookstore_project/settings.py

DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'postgres',
        'USER': 'postgres',
        'PASSWORD': 'postgres',
        'HOST': 'db',
        'PORT': 5432
    }
}
```

Refresh the web browser for the homepage to confirm everything still works correctly.

Custom User Model

Time to implement a custom user model which the official Django documentation “[highly recommends](#).” Why? Because you *will* need to make changes to the built-in `User` model at some point in your project’s life.

If you have not started with a custom user model from the very first `migrate` command you run, then you’re in for a world of hurt because `User` is tightly interwoven with the rest of Django internally. It is challenging to switch over to a custom user model mid-project.

A point of confusion for many people is that custom user models were only added in Django 1.5. Up until that point the recommended approach was to add a [OneToOneField](#), often called a Profile model, to `User`. You’ll often see this set up in older projects.

But these days using a custom user model is the more common approach. However as with many things Django-related, there are implementation choices: either extend [AbstractUser](#) which keeps the default `User` fields and permissions or extend [AbstractBaseUser](#) which is even more granular, and flexible, but requires more work.

We’ll stick with the simpler `AbstractUser` in this book as `AbstractBaseUser` can be added later if needed.

There are four steps for adding a custom user model to our project:

1. Create a `CustomUser` model
2. Update `settings.py`
3. Customize `UserCreationForm` and `UserChangeForm`
4. Add the custom user model to `admin.py`

The first step is to create a `CustomUser` model which will live within its own app. I like to name this app `users`. We could do this either locally within our virtual environment

shell, meaning we'd go `pipenv shell` and then run `python manage.py startapp users`. However for consistency we'll run the majority of our commands within Docker itself.

Command Line

```
$ docker-compose exec web python manage.py startapp users
```

Create a new `CustomUser` model which extends `AbstractUser`. That means we're essentially making a copy where `CustomUser` now has inherited all the functionality of `AbstractUser`, but we can override or add new functionality as needed. We're not making any changes yet so include the Python `pass` statement which acts as a placeholder for our future code.

Code

```
# users/models.py

from django.contrib.auth.models import AbstractUser
from django.db import models

class CustomUser(AbstractUser):
    pass
```

Now go in and update our `settings.py` file in the `INSTALLED_APPS` section to tell Django about our new `users` app. We also want to add a `AUTH_USER_MODEL` config at the bottom of the file which will cause our project to use `CustomUser` instead of the default `User` model.

Code

```
# bookstore_project/settings.py

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # Local
    'users.apps.UsersConfig', # new
]

...
AUTH_USER_MODEL = 'users.CustomUser' # new
```

Why do we do `users.apps.UsersConfig` here instead of just the older `users` approach? Both work but the longer form is a best practice as it takes advantage of additional configurations that can be added in [AppConfig](#).

Time to create a migrations file for the changes. We'll add the optional app name `users` to the command so that only changes to that app are included.

Command Line

```
$ docker-compose exec web python manage.py makemigrations users
Migrations for 'users':
  users/migrations/0001_initial.py
    - Create model CustomUser
```

Then run `migrate` to initialize the database for the very first time.

Command Line

```
$ docker-compose exec web python manage.py migrate
```

Custom User Forms

A user model can be both created and edited within the Django admin. So we'll need to update the built-in forms too to point to `CustomUser` instead of `User`.

Create a `users/forms.py` file.

Command Line

```
$ touch users/forms.py
```

In your text editor type in the following code to switch over to `CustomUser`.

Code

```
# users/forms.py

from django.contrib.auth import get_user_model
from django.contrib.auth.forms import UserCreationForm, UserChangeForm


class CustomUserCreationForm(UserCreationForm):

    class Meta(UserCreationForm):
        model = get_user_model()
        fields = ('email', 'username',)

class CustomUserChangeForm(UserChangeForm):

    class Meta(UserChangeForm):
        model = get_user_model()
        fields = ('email', 'username',)
```

At the very top we've imported `CustomUser` model via `get_user_model` which looks to our `AUTH_USER_MODEL` config in `settings.py`. This might feel a bit more circular than directly importing `CustomUser` here, but it enforces the idea of making one single reference to the custom user model rather than directly referring to it all over our project.

Next we import `UserCreationForm` and `UserChangeForm` which will both be extended.

Then create two new forms—`CustomUserCreationForm` and `CustomUserChangeForm`—that extend the base user forms imported above and specify swapping in our `CustomUser`

model and displaying the fields `email` and `username`. The `password` field is implicitly included by default and so does not need to be explicitly named here as well.

Custom User Admin

Finally we have to update our `users/admin.py` file. The admin is a common place to manipulate user data and there is tight coupling between the built-in `User` and the `admin`.

We'll extend the existing `UserAdmin` into `CustomUserAdmin` and tell Django to use our new forms, custom user model, and list only the `email` and `username` of a user. If we wanted to we could add more of the [existing User fields](#) to `list_display` such as `is_staff`.

Code

```
# users/admin.py

from django.contrib import admin
from django.contrib.auth import get_user_model
from django.contrib.auth.admin import UserAdmin

from .forms import CustomUserCreationForm, CustomUserChangeForm

CustomUser = get_user_model()

class CustomUserAdmin(UserAdmin):
    add_form = CustomUserCreationForm
    form = CustomUserChangeForm
    model = CustomUser
    list_display = ['email', 'username',]
```

```
admin.site.register(CustomUser, CustomUserAdmin)
```

Phew. A bit of code upfront but this saves a ton of heartache later on.

Superuser

A good way to confirm our custom user model is up and running properly is to create a superuser account so we can log into the admin. This command will access `CustomUserCreationForm` under the hood.

Command Line

```
$ docker-compose exec web python manage.py createsuperuser
```

I've used the username `wsv`, the email address `will@wsvincent.com`, and the password `testpass123`. You can use your own preferred variations here.

Now go to <http://127.0.0.1:8000/admin> and confirm that you can log in. You should see your superuser name in the upper right corner on the post-log in page.

The screenshot shows the Django admin homepage. At the top, there's a header bar with the title "Site administration | Django site" and a URL "127.0.0.1:8000/admin/". Below the header, the main content area has a blue header "Django administration" and a welcome message "WELCOME, wsv. [VIEW SITE](#) / [CHANGE PASSWORD](#) / [LOG OUT](#)". On the left, there are two main sections: "AUTHENTICATION AND AUTHORIZATION" (containing "Groups" with "Add" and "Change" buttons) and "USERS" (containing "Users" with "Add" and "Change" buttons). On the right, there are two panels: "Recent actions" (empty) and "My actions" (empty, with a note "None available").

Django admin homepage

You can also click on the `Users` section to see the email and username of your superuser account.

The screenshot shows the Django administration interface for the 'Users' section. At the top, there's a header bar with the title 'Select user to change' and the URL '127.0.0.1:8000/admin/users/customuser/'. Below the header, the main area is titled 'Django administration' and shows the path 'Home > Users > Users'. A sub-header 'WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT' is visible. The main content area is titled 'Select user to change' and contains a search bar with a magnifying glass icon and a 'Search' button. Below the search bar, there's an 'Action:' dropdown set to '-----' and a 'Go' button. To the right of the dropdown, it says '0 of 1 selected'. The user list table has columns for 'EMAIL ADDRESS' and 'USERNAME'. One user is listed: 'will@wsvincent.com' with 'wsv' in the 'USERNAME' column. A vertical filter sidebar on the right is titled 'FILTER' and contains sections for 'By staff status' (with 'All', 'Yes', and 'No' options), 'By superuser status' (with 'All', 'Yes', and 'No' options), and 'By active' (with 'All', 'Yes', and 'No' options). A 'ADD USER +' button is located at the top right of the main content area.

Django admin users page

Tests

Since we've added new functionality to our project we should test it. Whether you are a solo developer or working on a team, tests are important. In the words of Django co-founder Jacob Kaplan-Moss, “Code without tests is broken as designed.”

There are two main types of tests:

- *Unit tests* are small, fast, and isolated to a specific piece of functionality
- *Integration tests* are large, slow, and used for testing an entire application or a user flow like payment that covers multiple screens

You should write many unit tests and a small number of integration tests.

The Python programming language contains its own [unit testing framework](#) and Django's [automated testing framework](#) extends this with multiple additions into a web context. There is no excuse for not writing a lot of tests; they will save you time.

It's important to note that not everything needs to be tested. For example, any built-in Django features already contain tests in the source code. If we were using the default `User` model in our project we would not need to test it. But since we've created a `CustomUser` model we should.

Unit Tests

To write unit tests in Django we use [TestCase](#) which is, itself, an extension of Python's [TestCase](#). Our `users` app already contains a `tests.py` file which is automatically added when the `startapp` command is used. Currently it is blank. Let's fix that!

Each method must be prefaced with `test` in order to be run by the Django test suite. It is also a good idea to be overly descriptive with your unit test names since mature projects have hundreds if not thousands of tests!

Code

```
# users/tests.py

from django.contrib.auth import get_user_model
from django.test import TestCase

class CustomUserTests(TestCase):

    def test_create_user(self):
        User = get_user_model()
```

```
        user = User.objects.create_user(  
            username='will',  
            email='will@email.com',  
            password='testpass123'  
)  
  
    self.assertEqual(user.username, 'will')  
    self.assertEqual(user.email, 'will@email.com')  
    self.assertTrue(user.is_active)  
    self.assertFalse(user.is_staff)  
    self.assertFalse(user.is_superuser)  
  
  
def test_create_superuser(self):  
    User = get_user_model()  
    admin_user = User.objects.create_superuser(  
        username='superadmin',  
        email='superadmin@email.com',  
        password='testpass123'  
)  
    self.assertEqual(admin_user.username, 'superadmin')  
    self.assertEqual(admin_user.email, 'superadmin@email.com')  
    self.assertTrue(admin_user.is_active)  
    self.assertTrue(admin_user.is_staff)  
    self.assertTrue(admin_user.is_superuser)
```

At the top we have imported both `get_user_model` and `TestCase` before creating a `CustomUserTests` class. Within it are two separate tests. `test_create_user` confirms that a new user can be created. First we set our user model to the variable `User` and then create one via the manager method `create_user` which does the actual work of creating a new user with the proper permissions.

For `test_create_superuser` we follow a similar pattern but reference `create_superuser` instead of `create_user`. The difference between the two users is that a superuser should have both `is_staff` and `is_superuser` set to True.

To run our tests within Docker we'll prefix `docker-compose exec web` to the traditional command `python manage.py test`.

Command Line

```
$ docker-compose exec web python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

..
-----
Ran 2 tests in 0.268s
```

```
OK
```

```
Destroying test database for alias 'default'...
```

All the tests pass so we can proceed.

Git

We've accomplished quite a lot in this chapter so it is a good point to pause and commit our work by initializing a new Git repository, adding changes, and including a commit message.

Command Line

```
$ git init  
$ git status  
$ git add .  
$ git commit -m 'ch3'
```

You can compare with the [official source code for this chapter on Github](#).

Conclusion

Our Bookstore project is now running with Docker and PostgreSQL and we've configured a custom user model. Next up will be a `pages` app for our static pages.

Chapter 4: Pages App

Let's build a homepage for our new project. For now this will be a static page meaning it will not interact with the database in any way. Later on it will be a dynamic page displaying books for sale but...one thing at a time.

It's common to have multiple static pages in even a mature project such as an About page so let's create a dedicated pages app for them.

On the command line use the `startapp` command again to make a pages app.

Command Line

```
$ docker-compose exec web python manage.py startapp pages
```

Then add it to our `INSTALLED_APPS` setting. We'll also update `TEMPLATES` so that Django will look for a project-level `templates` folder. By default Django looks within each app for a `templates` folder, but organizing all templates in one space is easier to manage.

Code

```
# bookstore_project/settings.py

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
```

```
# Local
'users.apps.UsersConfig',
'pages.apps.PagesConfig', # new
]

TEMPLATES = [
{
    ...
'DIRS': [os.path.join(BASE_DIR, 'templates')], # new
    ...
}
]
```

Note that updating the `DIRS` setting means that Django will *also* look in this new folder; it will still look for any templates folders within an app.

Templates

Moving on it's time to create that new `templates` directory and put two files within it: `_base.html` and `home.html`. The first base level file will be inherited by all other files; `home.html` will be our homepage.

Command Line

```
$ mkdir templates  
$ touch templates/_base.html  
$ touch templates/home.html
```

Why call the base template `_base.html` with the underscore instead of `base.html`? This is optional, but some developers prefer to add an underscore `_` to denote a file that is intended to be inherited by other files and not displayed on its own.

In the base file we'll include the bare minimum needed and add `block` tags for both `title` and `content`. Block tags give higher-level templates the option to override just the content within the tags. For example, the homepage will have a title of "Home" but we want that to appear between html `<title></title>` tags. By using blocks on the content of the tag has to change. This saves typing and is arguably cleaner code.

Why use the name `content` for the main content of our project? This name could be anything—`main` or some other generic indicator—but using `content` is a common naming convention in the Django world. Can you use something else? Absolutely. Is `content` the most common one you'll see? Yes.

Code

```
<!-- templates/_base.html -->
<!DOCTYPE html>

<html>
<head>
    <meta charset="utf-8">
    <title>{% block title %}Bookstore{% endblock title %}</title>
</head>
<body>
    <div class="container">
        {% block content %}
        {% endblock content %}
    </div>
</body>
</html>
```

Now for the homepage which will simply say “Homepage” for now.

Code

```
<!-- templates/home.html -->
{% extends '_base.html' %}

{% block title %}Home{% endblock title %}

{% block content %}
<h1>Homepage</h1>
{% endblock content %}
```

URLs and Views

Every webpage in our Django project needs a `urls.py` and `views.py` file to go along with the template. For beginners the fact that order doesn't really matter here—we need all 3 files and really often a 4th, `models.py`, for the database—is confusing. Generally I prefer to start with the urls and work from there but there is no “right way” to build out this connected web of Django files.

Let's start with our project-level `urls.py` to set the proper path for webpages within the `pages` app. Since we want to create a homepage we add no additional prefix to the URL route which is designated by the empty string '''. We also import `include` on the second line to concisely add the `pages` app to our main `urls.py` file.

Code

```
# bookstore_project/urls.py

from django.contrib import admin
from django.urls import path, include # new

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('pages.urls')), # new
]
```

Next we create a `urls.py` file within the `pages` app.

Command Line

```
$ touch pages/urls.py
```

This file will import the `HomePageView` and set the path, again, to the empty string '''. Note that we provide an optional, but recommended, `named URL` of '`home`' at the end. This will come in handy shortly.

Code

```
# pages/urls.py

from django.urls import path

from .views import HomePageView

urlpatterns = [
    path('', HomePageView.as_view(), name='home'),
]
```

Finally we need a `views.py` file. We can leverage Django's built-in `TemplateView` so that the only tweak needed is to specify our desired template, `home.html`.

Code

```
# pages/views.py

from django.views.generic import TemplateView

class HomePageView(TemplateView):
    template_name = 'home.html'
```

We're almost done. If you navigate to the homepage now at `http://127.0.0.1:8000/` you'll actually see an error. But what's causing it? Since we're running the container in background detached mode—that `-d` flag—we must explicitly check the logs to see console output.

So type `docker-compose logs` which will turn up an error “`ModuleNotFoundError: No module named 'pages.urls'`”. What's happening is that Django does not automatically update the `settings.py` file for us based on a change. In a non-Docker world starting

and restarting the server does the trick. We must do the same here which means typing `docker-compose down` and then `docker-compose up -d` to load the new books app in properly.

Refresh the homepage now and it will work.



Tests

Time for tests. For our homepage we can use Django's `SimpleTestCase` which is a special subset of Django's `TestCase` that is designed for webpages that do not have a model included.

Testing can feel overwhelming at first, but it quickly becomes a bit boring. You'll use the same structure and techniques over and over again. In your text editor, update the existing `pages/tests.py` file. We'll start by testing the template.

Code

```
# pages/tests.py

from django.test import SimpleTestCase
from django.urls import reverse

class HomepageTests(SimpleTestCase):

    def test_homepage_status_code(self):
        response = self.client.get('/')
        self.assertEqual(response.status_code, 200)

    def test_homepage_url_name(self):
        response = self.client.get(reverse('home'))
        self.assertEqual(response.status_code, 200)
```

At the top we import `SimpleTestCase` as well as `reverse` which is useful for testing our URLs. Then we create a class called `HomepageTests` that extends `SimpleTestCase` and within it add a method for each unit test.

Note that we're adding `self` as the first argument of each unit test. This is a [Python convention](#) that is worth repeating.

It is best to be overly descriptive with your unit test names but be aware that each method must start with `test` to be run by the Django test suite.

The two tests here both check that the HTTP status code for the homepage equals 200 which means that it exists. It does not yet tell us anything specific about the contents of the page. For `test_homepageview_status_code` we're creating a variable called `response` that accesses the homepage `(/)` and then uses Python's `assertEqual` to check that the status code matches 200. A similar pattern exists for `test_homepage_url_name`

except that we are calling the URL name of `home` via the `reverse` method. Recall that we added this to the `pages/urls.py` file as a best practice. Even if we change the actual route of this page in the future, we can still refer to it by the same `home` URL name.

To run our tests we run the `python manage.py test` command albeit with the prefix `docker-compose exec web` so that it runs within Docker itself.

Command Line

```
$ docker-compose exec web python manage.py test
```

```
Creating test database for alias 'default'...
```

```
System check identified no issues (0 silenced).
```

```
..
```

```
Ran 4 tests in 0.277s
```

```
OK
```

```
Destroying test database for alias 'default'...
```

Why does it say 4 tests when we only created 2? Because we're testing the entire Django project and in the previous chapter under `users/tests.py` we added two tests for the custom user model. If we wanted to only run tests for the `pages` app we simply append that name onto the command so `docker-compose exec web python manage.py test pages`.

Testing Templates

So far we've tested that the homepage exists, but we should also confirm that it uses the correct template. `SimpleTestCase` comes with a method `assertTemplateUsed` just for this purpose! Let's use it.

Code

```
# pages/tests.py

from django.test import SimpleTestCase
from django.urls import reverse

class HomepageTests(SimpleTestCase):

    def test_homepage_status_code(self):
        response = self.client.get('/')
        self.assertEqual(response.status_code, 200)

    def test_homepage_url_name(self):
        response = self.client.get(reverse('home'))
        self.assertEqual(response.status_code, 200)

    def test_homepage_template(self): # new
        response = self.client.get('/')
        self.assertTemplateUsed(response, 'home.html')
```

We've created a `response` variable again and then checked that the template `home.html` is used. Let's run the tests again.

Command Line

```
$ docker-compose exec web python manage.py test pages
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

...
-----
Ran 3 tests in 0.023s
```

OK

```
Destroying test database for alias 'default'...
```

Did you notice something different in that command? We added the name of our app `pages` so that *only* the tests within that app were run. At this early state it's fine to run all the tests, but in larger projects if you know that you've only added tests within a specific app, it can save time to just run the updated/new tests and not the entire suite.

Testing HTML

Let's now confirm that our homepage has the correct HTML code and also does not have incorrect text. It's always good to test both that tests pass and that tests we expect to fail do, actually, fail!

Code

```
# pages/tests.py

from django.test import SimpleTestCase
from django.urls import reverse, resolve

from .views import HomePageView

class HomepageTests(SimpleTestCase):

    def test_homepage_status_code(self):
        response = self.client.get('/')
        self.assertEqual(response.status_code, 200)

    def test_homepage_url_name(self):
        response = self.client.get(reverse('home'))
        self.assertEqual(response.status_code, 200)

    def test_homepage_template(self):
        response = self.client.get('/')
        self.assertTemplateUsed(response, 'home.html')

    def test_homepage_contains_correct_html(self): # new
        response = self.client.get('/')
        self.assertContains(response, 'Homepage')
        self.assertNotContains(
            response, 'Hi there! I should not be on the page.')

    def test_homepage_does_not_contain_incorrect_html(self): # new
```

```
response = self.client.get('/')

self.assertNotContains(
    response, 'Hi there! I should not be on the page.')

---


```

Run the tests again.

Command Line

```
$ docker-compose exec web python manage.py test

Creating test database for alias 'default'...
System check identified no issues (0 silenced).

.....
-----
Ran 7 tests in 0.279s
```

```
OK
```

```
Destroying test database for alias 'default'...

---


```

setUp Method

Have you noticed that we seem to be repeating ourselves with these unit tests? For each one we are loading a `response` variable. That seems wasteful and prone to errors. It'd be better to stick to something more DRY (Don't Repeat Yourself).

Since the unit tests are executed top-to-bottom we can add a `setUp` method that will be run before every test. It will set `self.response` to our homepage so we no longer need to define a `response` variable for each test.

Code

```
# pages/tests.py

from django.test import SimpleTestCase
from django.urls import reverse

class HomepageTests(SimpleTestCase): # new

    def setUp(self):
        url = reverse('home')
        self.response = self.client.get(url)

    def test_homepage_status_code(self):
        self.assertEqual(self.response.status_code, 200)

    def test_homepage_template(self):
        self.assertTemplateUsed(self.response, 'home.html')

    def test_homepage_contains_correct_html(self):
        self.assertContains(self.response, 'Homepage')

    def test_homepage_does_not_contain_incorrect_html(self):
        self.assertNotContains(
            self.response, 'Hi there! I should not be on the page.')


---


```

Now run the tests again. Because `setUp` is a helper method and does not start with `test` it will not be considered a unit test in the final tally. So only 4 tests will run.

Command Line

```
$ docker-compose exec web python manage.py test pages
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

....
```

Ran 4 tests in 0.278s

OK

```
Destroying test database for alias 'default'...
```

Resolve

A final views check we can do is that our `HomePageView` “resolves” a given URL path. Django contains the utility function `resolve` for just this purpose. We will need to import it at the top of the file.

Our actual test, `test_homepage_url_resolves_homepageview`, checks that the name of the view used to resolve `/` matches `HomePageView`.

Code

```
# pages/tests.py

from django.test import SimpleTestCase
from django.urls import reverse, resolve # new

from .views import HomePageView

class HomepageTests(SimpleTestCase):

    def setUp(self):
        url = reverse('home')
        self.response = self.client.get(url)

    def test_homepage_status_code(self):
        self.assertEqual(self.response.status_code, 200)

    def test_homepage_template(self):
        self.assertTemplateUsed(self.response, 'home.html')

    def test_homepage_contains_correct_html(self):
        self.assertContains(self.response, 'Homepage')

    def test_homepage_does_not_contain_incorrect_html(self):
        self.assertNotContains(
            self.response, 'Hi there! I should not be on the page.')

    def test_homepage_url_resolves_homepageview(self): # new
        view = resolve('/')
```

```
self.assertEqual(  
    view.func.__name__,  
    HomePageView.as_view().__name__  
)
```

Phew. That's our last test. Let's confirm that everything passes.

Command Line

```
$ docker-compose exec web python manage.py test  
Creating test database for alias 'default'...  
System check identified no issues (0 silenced).  
.....  
-----  
Ran 7 tests in 0.282s
```

OK

```
Destroying test database for alias 'default'...
```

Git

Time to add our new changes to source control with Git.

Command Line

```
$ git status  
$ git add .  
$ git commit -m 'ch4'
```

You can compare with the [official source code on Github](#) for this chapter.

Conclusion

We have configured our templates and added the first page to our project, a static homepage. We also added tests which should *always* be included with new code changes. Some developers prefer a method called Test-Driven Development where they write the tests first and then the code. Personally I prefer to write the tests *immediately after* which is what we'll do here.

Both approaches work, the key thing is to be rigorous with your testing. Django projects quickly grow in size where it's impossible to remember all the working pieces in your head. And if you are working on a team, it is a nightmare to work on an untested codebase. Who knows what will break?

In the next chapter we'll add user registration to our project: log in, log out, and sign up.

Chapter 5: User Registration

User registration is a core feature in any dynamic website. And it will be in our Bookstore project, too. In this chapter we will implement log in, log out, and sign up functionality. The first two are relatively straightforward since Django provides us with the necessary views and urls for them, however sign up is more challenging since there is no built-in solution.

Auth App

Let's begin by implementing log in and log out using Django's own [auth](#) app. Django provides us with the necessary views and urls which means we only need to update a template for things to work. This saves us a lot of time as developers and it ensures that we don't make a mistake since the underlying code has already been tested and used by millions of developers.

However this simplicity comes at the cost of feeling "magical" to Django newcomers. We covered some of these steps previously in my book, [Django for Beginners](#), but we did not slow down and look at the underlying source code. The intention for a beginner was to broadly explain and demonstrate "how" to implement user registration properly, but this came at the cost of truly diving into "why" we used the code we did.

Since this is a more advanced book, we delve deeper to understand the underlying source code better. The approach here can also be used to explore any other built-in Django functionality on your own.

The first thing we need to do is make sure the `auth` app is included in our `INSTALLED_APPS` setting. We have added our own apps here previously, but have you ever taken a close look at the built-in apps Django adds automatically for us? Most likely the answer is no. Let's do that now!

Code

```
# bookstore_project/settings.py

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth', # Yoohoo!!!!
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # Local
    'users.apps.UsersConfig',
    'pages.apps.PagesConfig',
]
```

There are, in fact, 6 apps already there that Django provides for us which power the site. The first is `admin` and the second is `auth`. This is how we know the `auth` app is already present in our Django project.

When we earlier ran the `migrate` command for the first time all of these apps were linked together in the initial database. And remember that we used the `AUTH_USER_MODEL` setting to tell Django to use our custom user model, not the default `User` model here. This is why we had to wait until that configuration was complete before running `migrate` for the first time.

Auth URLs and Views

To use Django's built-in `auth` app we must explicitly add it to our `bookstore_project/urls.py` file. The easiest approach is to use `accounts/` as the prefix since that is commonly used in the Django community. Make the one line change below. Note that as our `urls.py` file grows in length, adding comments for each type of URL-admin, user management, local apps, etc.-helps with readability.

Code

```
# bookstore_project/urls.py

from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    # Django admin
    path('admin/', admin.site.urls),

    # User management
    path('accounts/', include('django.contrib.auth.urls')), # new

    # Local apps
    path('', include('pages.urls')),
]
```

What's included in the `auth` app? A lot it turns out. First off, there are a number of associated urls.

Code

```
accounts/login/ [name='login']
accounts/logout/ [name='logout']
accounts/password_change/ [name='password_change']
accounts/password_change/done/ [name='password_change_done']
accounts/password_reset/ [name='password_reset']
accounts/password_reset/done/ [name='password_reset_done']
accounts/reset/<uidb64>/<token>/ [name='password_reset_confirm']
accounts/reset/done/ [name='password_reset_complete']
```

How did I know that? Two ways. The first is the [official auth docs](#) tell us so! But a second, deeper approach is to look at the Django source code which is [available on Github](#). If we navigate or search around we'll find our way to the [auth app itself](#). And within that we can find the `urls.py` file [at this link](#) which shows the complete source code.

It takes practice to understand the Django source code, but it is well worth the time.

Homepage

What's next? Let's update our existing homepage so that it will notify us if a user is already logged in or not which currently can only happen via the admin.

Here is the new code for the `templates/home.html` file. It uses the Django templating engine's [if/else](#) tags for basic logic.

Code

```
<!-- templates/home.html -->
{% extends '_base.html' %}

{% block title %}Home{% endblock title %}

{% block content %}

<h1>Homepage</h1>

{% if user.is_authenticated %}

    Hi {{ user.email }}!

{% else %}

    <p>You are not logged in</p>
    <a href="{% url 'login' %}">Log In</a>

{% endif %}

{% endblock content %}
```

If the user is logged in (authenticated), we display a greeting that says “Hi” and includes their email address. These are both **variables** which we can use with Django’s template engine via double opening {{ and closing }} brackets.

The default `User` contains numerous fields including `is_authenticated` and `email` which are referenced here.

And the `logout` and `login` are URL names. The `url` template tag means if we specify the URL name the link will automatically refer to that URL path. For example, in the previous chapter we set the name of our homepage URL to `home` so a link to the homepage would take the format of `{% url 'home' %}`. More on this shortly.

If you look at the homepage now at `http://127.0.0.1:8000/` it will likely show the email address of your superuser account since we used it previously to log in.



Homepage

Hi will@wsvincent.com!

Homepage with greeting

In the admin over at <http://127.0.0.1:8000/admin/> if you click on the “Log out” button in the upper right corner we can log out of the admin and by extension the Django project.

The screenshot shows the Django admin interface. The top navigation bar includes links for 'Home', '127.0.0.1:8000', and 'Logout'. The main header 'Django administration' is displayed. On the right, there's a welcome message 'WELCOME, wsv.' followed by links for 'VIEW SITE / CHANGE PASSWORD / LOG OUT'. The 'LOG OUT' link is specifically highlighted with a red circle and an arrow pointing to it from the text below. The left sidebar lists 'AUTHENTICATION AND AUTHORIZATION' (Groups, Text) and 'USERS' (Users). The right sidebar shows 'Recent actions' and 'My actions' (None available).

Admin logout link

Return to the homepage at <http://127.0.0.1:8000/> and refresh the page.

NoReverseMatch at /

Reverse for 'login' not found. 'login' is not a valid view function or pattern name.

Request Method: GET
Request URL: http://127.0.0.1:8000/
Django Version: 2.1.5
Exception Type: NoReverseMatch
Exception Value: Reverse for 'login' not found. 'login' is not a valid view function or pattern name.
Exception Location: /usr/local/lib/python3.7/site-packages/django/urls/resolvers.py in _reverse_with_prefix, line 622
Python Executable: /usr/local/bin/python
Python Version: 3.7.2
Python Path: ['/code', '/usr/local/lib/python37.zip', '/usr/local/lib/python3.7', '/usr/local/lib/python3.7/lib-dynload', '/usr/local/lib/python3.7/site-packages']
Server time: Thu, 14 Mar 2019 17:33:28 +0000

Error during template rendering

In template /code/templates/_base.html, error at line 0

Reverse for 'login' not found. 'login' is not a valid view function or pattern name.

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title>{% block title %}Bookstore{% endblock title %}</title>
6 </head>
7 <body>
8   <div class="container">
9     {% block content %}
10    {% endblock content %}
```

Homepage with error

What is this!?

Redirects

Fortunately Django is quite descriptive in its error messages. The problem is `Reverse for 'login' not found`. Django doesn't know *where* to redirect the user after they click on the "login" link in our template.

We must use the `LOGIN_REDIRECT_URL` config to set this. We'll specify the named URL, which is `home` for our homepage, so that users are redirected to the homepage.

At the bottom of the `bookstore_project/settings.py` file add this one line.

Code

```
# bookstore_project/settings.py  
LOGIN_REDIRECT_URL = 'home'
```

Refresh the webpage and you will see it notices the change and provides us with the generic “You are not logged in” greeting.



Homepage

You are not logged in

[Log In](#)

Homepage logged out

Django Source Code

You might have been able to piece together these steps on your own from reading [the official docs](#). But the deeper-and better-approach is to learn how to read the Django source code on your own.

One question is, how was the `user` and its related variables magically available in our template? The answer is that Django has a concept called the `template context` which means each template is loaded with data from the corresponding `views.py` file. We can use `user` within template tags to access User attributes. In other words, Django just gives this to us automatically.

So to check if a user is logged in or not, we access `user` and then can use the boolean `is_authenticated` attribute. If a user is logged in, it will return `True` and we can do things like display the user's email. Or if no user is logged in, the result will be `False`.

Moving on we have the URL name `login`. Where did that come from? The answer, of course, is from Django itself! Let's unpack the code snippet `{% url 'login' %}` piece by piece.

First up we're using the [url template tag](#) which takes as its first argument a [named URL pattern](#). That's the optional `name` section we add as a best practice to all our URL paths. Therefore there must be a '`login`' name attached to the URL used by Django for log outs, right!

There are two ways we could have known this. In other words, if I hadn't just told you that we wanted to use `{% url 'login' %}`, how could you have figured it out?

First look at the [official documentation](#). Personally I often use the search feature so I would have typed in something like "logout" and then clicked around until I found a description of log out. The one we want is actually called [authentication views](#) and lists the corresponding URL patterns for us.

Code

```
accounts/login/ [name='login']
accounts/logout/ [name='logout']
accounts/password_change/ [name='password_change']
accounts/password_change/done/ [name='password_change_done']
accounts/password_reset/ [name='password_reset']
accounts/password_reset/done/ [name='password_reset_done']
accounts/reset/<uidb64>/<token>/ [name='password_reset_confirm']
accounts/reset/done/ [name='password_reset_complete']
```

This tells us at the path `accounts/login/` is where "login" is located and its name is '`login`'. A little confusing at first, but here is the info we need.

Going a step deeper to phase two, we can investigate the underlying Django source code to see "logout" in action. If you perform a search [over on Github](#) you'll eventually

find the [auth app itself](#). Ok, now let's start by investigating the `urls.py` file. [Here is the link](#) to the complete code:

Code

```
# django/contrib/auth/urls.py

from django.contrib.auth import views
from django.urls import path

urlpatterns = [
    path('login/', views.LoginView.as_view(), name='login'),
    path('logout/', views.LogoutView.as_view(), name='logout'),

    path('password_change/', views.PasswordChangeView.as_view(),
         name='password_change'),
    path('password_change/done/', views.PasswordChangeDoneView.as_view(),
         name='password_change_done'),
    path('password_reset/', views.PasswordResetView.as_view(),
         name='password_reset'),
    path('password_reset/done/', views.PasswordResetDoneView.as_view(),
         name='password_reset_done'),
    path('reset/<uidb64>/<token>/', views.PasswordResetConfirmView.as_view(),
         name='password_reset_confirm'),
    path('reset/done/', views.PasswordResetCompleteView.as_view(),
         name='password_reset_complete'),
]
```

Here is the underlying code Django uses itself for the `auth` app. I hope you can see that the “`logout`” route is not magic. It's right there in plain site, it uses the view `LogoutView` and has the URL name '`logout`'. Not magic at all! Just a little challenging to find the first time.

This three-step process is a great way to learn: either remember the Django shortcut, look it up in the docs, or on occasion dive into the source code and truly understand where all this goodness comes from.

Log Out

Now let's add a log out option to our homepage since only a superuser will have access to the admin. How do we do this?

If you look at the auth views above we can see that logout uses `LogoutView`, which we could explore in the source code, and has a URL name of `logout`. That means we can refer to it with a template tag as just `logout`.

One last thing...a logout also requires a redirect, just like login. So rather than show you the error message page take a look at `LOGOUT_REDIRECT_URL` which can be added to the bottom of our `bookstore_project/settings.py` file.

Code

```
# bookstore_project/settings.py
LOGIN_REDIRECT_URL = 'home'
LOGOUT_REDIRECT_URL = 'home' # new
```

Then add the logout link to `templates/home.html`.

Code

```
<!-- templates/home.html -->
{% extends '_base.html' %}

{% block title %}Home{% endblock title %}

{% block content %}

<h1>Homepage</h1>

{% if user.is_authenticated %}

    Hi {{ user.email }}!

    <p><a href="{% url 'logout' %}">Log Out</a></p>

{% else %}

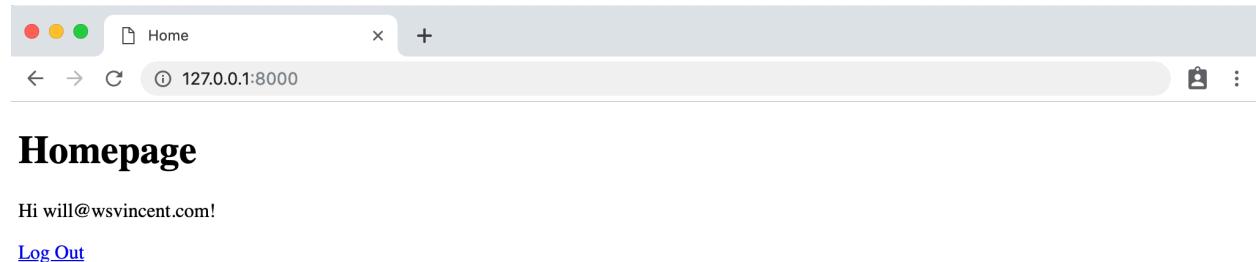
    <p>You are not logged in</p>

    <a href="{% url 'login' %}">Log In</a>

{% endif %}

{% endblock content %}
```

Go back into the admin at <http://127.0.0.1:8000/admin/> to log in. Then navigate to the homepage <http://127.0.0.1:8000/> which now has the “Log out” link.



The screenshot shows a web browser window with the following details:

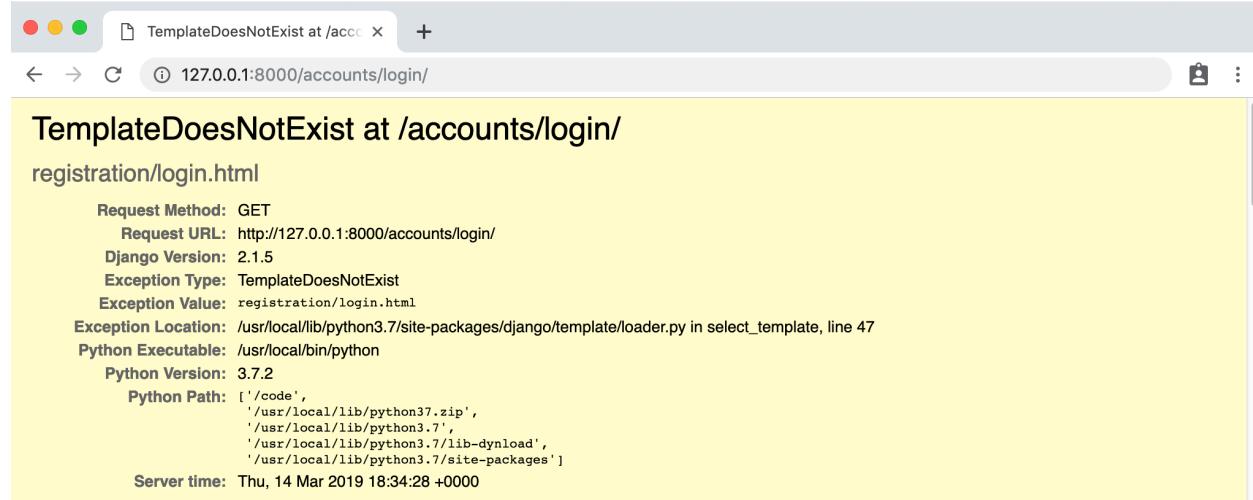
- Address bar: Home - 127.0.0.1:8000
- Content area:
 - Header: **Homepage**
 - Welcome message: Hi will@wsvincent.com!
 - Link: [Log Out](#)

Homepage with logout link

If you click on it you will be logged out and see our homepage with a log in link.

Log In

Click on the “Log In” link now. It results in an error!



The screenshot shows a web browser window with the title "TemplateDoesNotExist at /accounts/login/". The URL in the address bar is "127.0.0.1:8000/accounts/login/". The main content area displays the following error message and details:

```
TemplateDoesNotExist at /accounts/login/
registration/login.html

Request Method: GET
Request URL: http://127.0.0.1:8000/accounts/login/
Django Version: 2.1.5
Exception Type: TemplateDoesNotExist
Exception Value: registration/login.html
Exception Location: /usr/local/lib/python3.7/site-packages/django/template/loader.py in select_template, line 47
Python Executable: /usr/local/bin/python
Python Version: 3.7.2
Python Path: ['/code',
              '/usr/local/lib/python37.zip',
              '/usr/local/lib/python3.7',
              '/usr/local/lib/python3.7/lib-dynload',
              '/usr/local/lib/python3.7/site-packages']
Server time: Thu, 14 Mar 2019 18:34:28 +0000
```

Log in template not exist error

Django is throwing a `TemplateDoesNotExist` error at us. Specifically, it seems to expect a log in template at `registration/login.html`. In addition to Django telling us this, we can look in the [documentation](#) and see that the desired `template_name` has that location.

But let's really be sure and check the source code so we can remove any perceived magic here. After all, it's just Django.

Back in the `auth/views.py` file we can see on line 45 for `LoginView` that the `template_name` is '`registration/login.html`'. So if we wanted to change the default location we could, but it would mean overriding `LoginView` which seems like overkill. Let's just use what Django gives us here.

Create a new `registration` folder within the existing `templates` directory and then add our `login.html` file there, too.

Command Line

```
$ mkdir templates/registration  
$ touch templates/registration/login.html
```

The actual code is as follows. We extend our base template, add a title, and then specify that we want to use a form that will “post” or send the data.

Code

```
<!-- templates/registration/login.html -->  
{% extends '_base.html' %}  
  
{% block title %}Log In{% endblock title %}  
  
{% block content %}  
<h2>Log In</h2>  
<form method="post">  
  {% csrf_token %}  
  {{ form.as_p }}  
  <button type="submit">Log In</button>  
</form>  
{% endblock content %}
```

You should **always** add **CSRF protection** on any submittable form. Otherwise a malicious website can change the link and attack the site and the user. Django has CSRF middleware to handle this for us; all we need to do is add `{% csrf_token %}` tags at the start of the form.

Next we can control the look of the form contents. For now we’ll use `as_p()` so that each form field is displayed within a paragraph `p` tag.

With that explanation out of the way, let's check if our new template is working correctly. Go to <http://127.0.0.1:8000/accounts/login/>.



And there is our page! Lovely. You can navigate back to the homepage and confirm that the “Log In” link works too if you like. As a final step, go ahead and log in. It should redirect you back to the homepage and provide a greeting to you.

Sign Up

Implementing a sign up page for user registration is completely up to us. We'll go through the standard steps for any new page:

- add a URL path in `users/urls.py`
- update `bookstore_project/urls.py` to point to the `users` app
- create a `users/views.py` file
- create a `signup.html` template
- update `_base.html` to display the sign up page

A common question is: what's the right order for implementing these steps? Honestly it doesn't matter since we need *all* of them for the sign up page to work properly. Generally I like to start with `urls`, then switch to `views`, and finally `templates` but it's a matter of personal preference.

To start create a `urls.py` file within the `users` app. Up to this point it only contains our `CustomUser` in the `models.py` file; we haven't configured any routes or views.

Command Line

```
$ touch users/urls.py
```

The URL path for the sign up page will take a view called `SignupPageView` (which we'll create next), at the route `signup/`, and have a `name` of `signup` which we can later use to refer to the page with a `url` template tag. The existing url names for `login` and `signup` are written within the built-in Django app file `django/contrib/auth/urls.py` we saw above.

Code

```
# users/urls.py

from django.urls import path

from .views import SignupPageView

urlpatterns = [
    path('signup/', SignupPageView.as_view(), name='signup'),
]
```

Next update the `bookstore_project/urls.py` file to include the `users` app. We can create any route we like but it's common to use the same `accounts/` one used by the default `auth` app. Note that it's important to include the path for `users.urls` below: URL paths are loaded top-to-bottom so this ensures that any `auth` URL paths will be loaded first.

Code

```
# bookstore_project/urls.py

from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    # Django admin
    path('admin/', admin.site.urls),

    # User management
    path('accounts/', include('django.contrib.auth.urls')),

    # Local apps
    path('accounts/', include('users.urls')), # new
    path('', include('pages.urls')),
]
```

Now we can create our view which will be called `SignupPageView`. It references the `CustomUserCreationForm`, has a `success_url` that points to the `login` page meaning after the form is submitted the user will be redirected there. And the `template_name` will be `signup.html`.

Code

```
# users/views.py

from django.urls import reverse_lazy
from django.views import generic

from .forms import CustomUserCreationForm

class SignupPageView(generic.CreateView):
    form_class = CustomUserCreationForm
    success_url = reverse_lazy('login')
    template_name = 'signup.html'
```

Finally we have our template. Create a `signup.html` file within our existing `templates` directory.

Command Line

```
$ touch templates/signup.html
```

The code is basically identical to the log in page.

Code

```
<!-- templates/signup.html -->
{% extends '_base.html' %}

{% block title %}Sign Up{% endblock title %}

{% block content %}

<h2>Sign Up</h2>

<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Sign Up</button>
</form>

{% endblock content %}
```

As a final step we can add a line for “Sign Up” to our `home.html` template right below the link for “Log In”. This is a one-line change.

Code

```
<!-- templates/home.html -->
{% extends '_base.html' %}

{% block title %}Home{% endblock title %}

{% block content %}

<h1>Homepage</h1>

{% if user.is_authenticated %}
    Hi {{ user.email }}!
<p><a href="{% url 'logout' %}">Log Out</a></p>
```

```
{% else %}

<p>You are not logged in</p>

<a href="{% url 'login' %}">Log In</a>
<a href="{% url 'signup' %}">Sign Up</a>

{% endif %}

{% endblock content %}
```

All done! Reload the homepage to see our work.



Homepage

You are not logged in

[Log In](#) [Sign Up](#)

Homepage with Signup

The “Sign Up” link will redirect us to <http://127.0.0.1:8000/accounts/signup/>.



Sign Up

Email address:

Username: Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password:

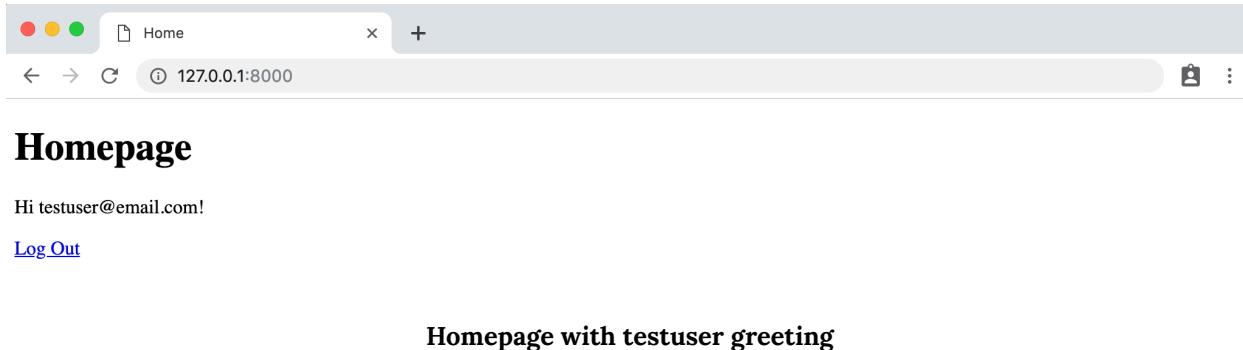
- Your password can't be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can't be a commonly used password.
- Your password can't be entirely numeric.

Password confirmation: Enter the same password as before, for verification.

Signup page

Create a new user with the email address `testuser@email.com`, username of `testuser`, and `testpass123` for the password. Upon submission it will redirect us to the Log In

page. Attempt to log in with this new account.



Tests

For tests we do *not* need to test log in and log out features since those are built into Django and already have tests. We *do* need to test our sign up functionality though!

Let's start by creating a `setUp` method that loads our page. Then we'll populate `test_signup_template` with tests for the status code, template used, and both included and excluded text similarly to how we did it in the last chapter for the homepage.

In your text editor, update the `users/tests.py` file with these changes.

Code

```
# users/tests.py

from django.contrib.auth import get_user_model
from django.test import TestCase
from django.urls import reverse # new

class CustomUserTests(TestCase):
    ...
    
```

```
class SignupPageTests(TestCase): # new

    def setUp(self):
        url = reverse('signup')
        self.response = self.client.get(url)

    def test_signup_template(self):
        self.assertEqual(self.response.status_code, 200)
        self.assertTemplateUsed(self.response, 'signup.html')
        self.assertContains(self.response, 'Sign Up')
        self.assertNotContains(
            self.response, 'Hi there! I should not be on the page.')


---


```

Then run our tests.

Command Line

```
$ docker-compose exec web python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

.....
-----
Ran 8 tests in 0.329s
```

OK

```
Destroying test database for alias 'default'...
```

Next we can test that our `CustomUserCreationForm` is being used and that the page resolves to `SignupPageView`.

Code

```
# users/tests.py

from django.contrib.auth import get_user_model
from django.test import TestCase
from django.urls import reverse, resolve # new

from .forms import CustomUserCreationForm # new
from .views import SignupPageView # new

class CustomUserTests(TestCase):
    ...

class SignupPageTests(TestCase):

    def setUp(self):
        url = reverse('signup')
        self.response = self.client.get(url)

    def test_signup_template(self):
        self.assertEqual(self.response.status_code, 200)
        self.assertTemplateUsed(self.response, 'signup.html')
        self.assertContains(self.response, 'Sign Up')
        self.assertNotContains(
            self.response, 'Hi there! I should not be on the page.')

    def test_signup_form(self): # new
        form = self.response.context.get('form')
        self.assertIsInstance(form, CustomUserCreationForm)
```

```
self.assertContains(self.response, 'csrfmiddlewaretoken')

def test_signup_view(self): # new
    view = resolve('/accounts/signup/')
    self.assertEqual(
        view.func.__name__,
        SignupPageView.as_view().__name__
    )
```

Run our tests again.

Command Line

```
$ docker-compose exec web python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

.....
-----
```

Ran 10 tests in 0.328s

OK

```
Destroying test database for alias 'default'...
```

All done.

setUpTestData()

Django 1.8 introduced a [major update to TestCase](#) that added the ability to run tests both within a whole class and for each individual test. In particular, [setUpTestData\(\)](#) allows the creation of initial data at the class level that can be applied to the entire

`TestCase`. This results in much faster tests than using `setUp()`, however, care must be taken not to modify any objects created in `setUpTestData()` in your test methods.

We will use `setUp()` in this book, but be aware that if your test suite seems sluggish, `setUpTestData()` is a potential optimization to look into.

Git

As ever make sure to save our work by adding changes into Git.

Command Line

```
$ git status  
$ git add .  
$ git commit -m 'ch5'
```

The official source code is [located on Github](#) if you want to compare your code.

Conclusion

Our Bookstore project is not the most beautiful site in the world, but it is very functional at this point. In the next chapter we'll configure our static assets and add Bootstrap for improved styling.

Chapter 6: Static Assets

Static assets like CSS, JavaScript, and images are a core component of any website and Django provides us with a large degree of flexibility around their configuration and storage. In this chapter we'll configure our initial static assets and add [Bootstrap](#) to our project for improved styling.

staticfiles app

Django relies on the [staticfiles app](#) to manage static files from across our entire project, make them accessible for rapid local development on the file system, and also combine them into a single location that can be served in a better performing manner in production. This process and the distinction between local and production static files confuses many Django newcomers.

To start we'll update the [staticfiles app](#) configuration in `settings.py`.

STATIC_URL

The first static file setting, `STATIC_URL`, is already included for us in the `bookstore-project/settings.py` file.

Code

```
# bookstore_project/settings.py  
STATIC_URL = '/static/'
```

This sets the URL that we can use to reference static files. Note that it is important to include a trailing slash / at the end of the directory name.

STATICFILES_DIRS

Next up is `STATICFILES_DIRS` which defines the location of static files in *local development*. In our project these will all live within a top-level `static` directory.

Code

```
# bookstore_project/settings.py  
STATIC_URL = '/static/'  
STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static'),] # new
```

It's often the case that there will be multiple directories with static files within a project so Python brackets [], which denote a `list`, are typically added here to accommodate future additions.

STATIC_ROOT

`STATIC_ROOT` is the location of static files for *production* so it must be set to a different name, typically `staticfiles`. When it comes time to deploy a Django project, the `collectstatic` command will automatically compile all available static files throughout the entire project into a single directory. This is far faster than having static files sprinkled across the project as is the case in local development.

Code

```
# bookstore_project/settings.py

STATIC_URL = '/static/'

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

STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles') # new
```

STATICFILES_FINDERS

The last setting is `STATICFILES_FINDERS` which tells Django how to look for static file directories. It is implicitly set for us and although this is an optional step, I prefer to make it explicit in all projects.

Code

```
# bookstore_project/settings.py

STATICFILES_FINDERS = [
    "django.contrib.staticfiles.finders.FileSystemFinder",
    "django.contrib.staticfiles.finders.AppDirectoriesFinder",
]
```

The `FileSystemFinder` looks within the `STATICFILES_DIRS` setting, which we set to `static`, for any static files. Then the `AppDirectoriesFinder` looks for any directories named `static` located within an app, as opposed to located at a project-level `static` directory. This setting is read top-to-bottom meaning if a file called `static/img.jpg` is first found by `FileSystemFinder` it will be in place of an `img.jpg` file located within, say, the `pages` app at `pages/static/img.jpg`.

Our final group of settings therefore should look as follows:

Code

```
# bookstore_project/settings.py

STATIC_URL = '/static/'

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

STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')

STATICFILES_FINDERS = [
    "django.contrib.staticfiles.finders.FileSystemFinder",
    "django.contrib.staticfiles.finders.AppDirectoriesFinder",
]
```

Static Directory

Let's now add some static files and incorporate them into our project. Even though we're referring to a `static` directory for our files it's up to us to create it so do that now along with new subdirectories for CSS, JavaScript, and images.

Command Line

```
$ mkdir static
$ mkdir static/css
$ mkdir static/js
$ mkdir static/images
```

Next create a `base.css` file.

Command Line

```
$ touch static/css/base.css
```

We'll keep things basic and have our `h1` headline be red. The point is to show how CSS can be added to our project, not to delve too deeply into CSS itself.

Code

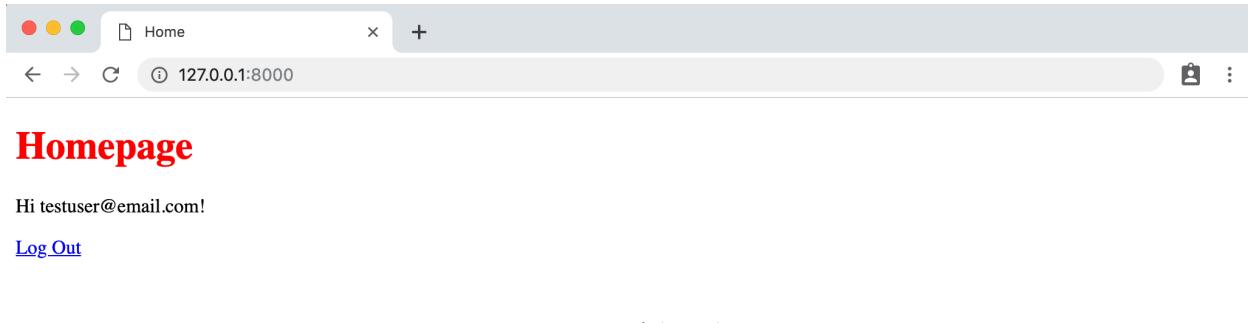
```
/* static/css/base.css */  
  
h1 {  
    color: red;  
}
```

If you refresh the homepage now you'll see that nothing has changed. That's because static assets must be explicitly loaded into the templates. First load all static files at the top of the page with `{% load static %}` and then include a link to the `base.css` file. The `static` template tag uses `STATIC_URL`, which we set to `/static/`, so rather than needing to write out `static/css/base.css` we can simply refer to `css/base.css`.

Code

```
<!-- templates/_base.html -->  
  
{% load static %}  
  
<!DOCTYPE html>  
  
<html>  
  
<head>  
    <meta charset="utf-8">  
    <title>{% block title %}Bookstore{% endblock %}</title>  
    <!-- CSS -->  
    <link rel="stylesheet" href="{% static 'css/base.css' %}">  
</head>  
...
```

Refresh the homepage to see our work. There's our CSS in action!



A screenshot of a web browser window. The title bar says "Home". Below it, the address bar shows "127.0.0.1:8000". The main content area has a red background. At the top left, there is a red "Homepage" heading. Below it, the text "Hi testuser@email.com!" is displayed in white. At the bottom left, there is a blue "Log Out" link. In the center of the page, the text "Homepage with red text" is displayed in black.

If instead you see an error screen saying Invalid block tag on line 7: 'static'. Did you forget to register or load this tag? then you forgot to include the line `{% load static %}` at the top of the file. I do this all the time myself.

Images

How about an image? You can download the book cover for *Django for Professionals* at [this link](#). Save it into the directory `books/static/images` as `djangoforprofessionals.jpg`.

To display it on the homepage, update `templates/home.html`. Add both the `{% load static %}` tags at the top and on the next-to-last line the `` link for the file.

Code

```
<!-- templates/home.html -->

{% extends '_base.html' %}

{% load static %}

{% block title %}Home{% endblock title %}

{% block content %}

<h1>Homepage</h1>


```

```
{% if user.is_authenticated %}

<p>Hi {{ user.email }}!</p>

<p><a href="{% url 'logout' %}">Log Out</a></p>

{% else %}

<p>You are not logged in</p>

<p><a href="{% url 'login' %}">Log In</a> | <a href="{% url 'signup' %}">Sign Up</a></p>

{% endif %}

{% endblock content %}
```

Refreshing the homepage you'll see the raw file is quite large! Let's control that with some additional CSS.

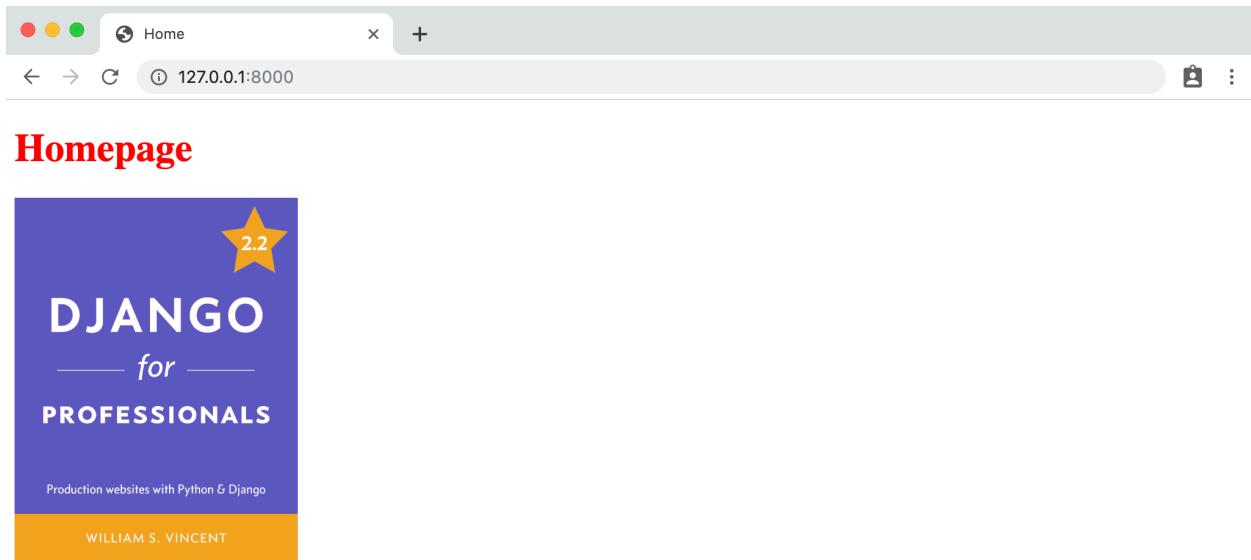
Code

```
/* static/css/base.css */

h1 {
    color: red;
}

.bookcover {
    height: 300px;
    width: auto;
}
```

Now update the homepage and the book cover image fits nicely.



Homepage with Book Cover

JavaScript

To add JavaScript we'll go through a similar process. Create a new file called `base.js`.

Command Line

```
$ touch static/js/base.js
```

Often I put a tracking code of some kind here, such as for Google Analytics, but for demonstration purposes we'll add a `console.log` statement so we can confirm the JavaScript loaded correctly.

Code

```
// static/js/base.js  
console.log('JavaScript here!')
```

Now add it to our `_base.html` template. JavaScript should be added at the bottom of the file so it is loaded last, after the HTML, CSS, and other assets that appear first on the screen when rendered in the web browser. This gives the appearance of the complete webpage loading faster.

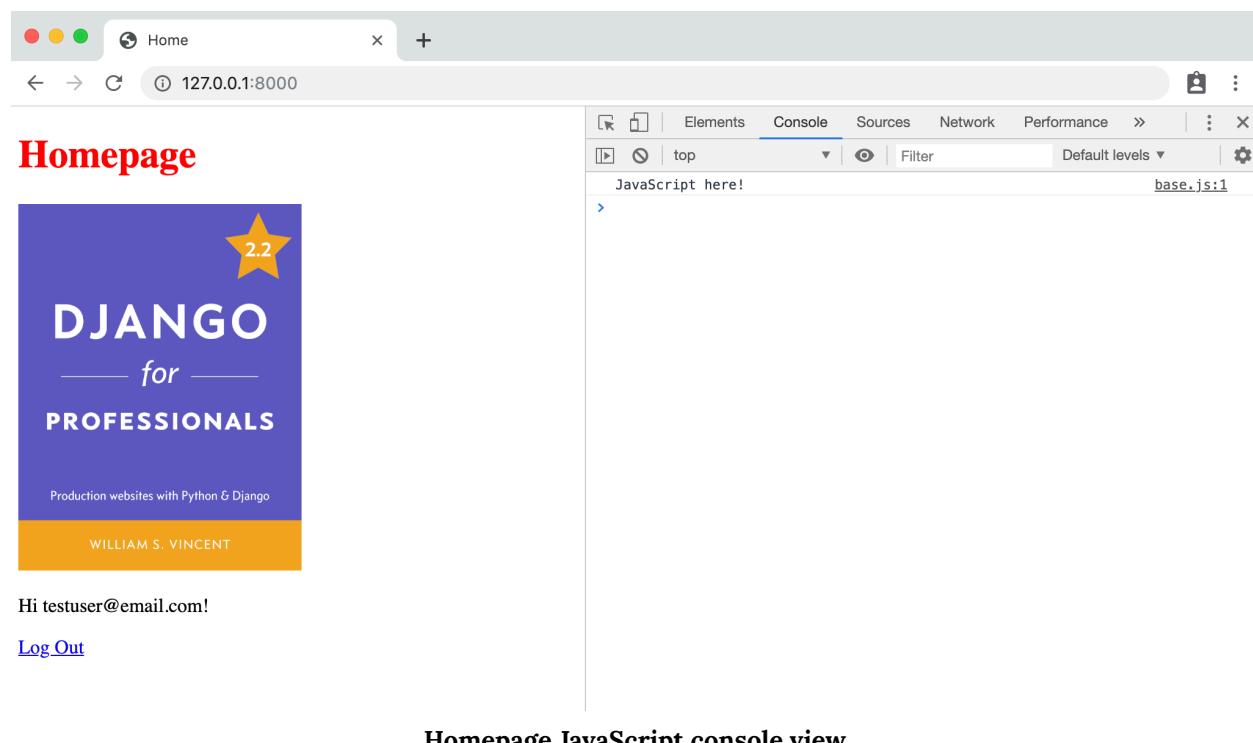
Code

```
<!-- templates/_base.html -->  
{% load static %}  
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="utf-8">  
    <title>{% block title %}Bookstore{% endblock title %}</title>  
    <!-- CSS -->  
    <link rel="stylesheet" href="{% static 'css/base.css' %}">  
</head>  
<body>  
    <div class="container">  
        {% block content %}  
        {% endblock content %}  
    </div>  
    <!-- JavaScript -->  
    {% block javascript %}  
    <script src="{% static 'js/base.js' %}"></script>  
    {% endblock javascript %}
```

```
</body>  
</html>
```

In your web browser, make the JavaScript console available. This involves opening up Developer Tools and making sure you're on the “Console” section. On Chrome which is being used for the images in this book, go to `View` in the top menu, then `Developer` → `Developer Tools` which will open a sidebar. Make sure `Console` is selected from the options.

If you refresh the page, you should see the following:



collectstatic

Imagine we wanted to deploy our website right away. Among other steps, we'd need to run `collectstatic` to create a single, production-ready directory of all the static files in our project.

Command Line

```
$ docker-compose exec web python manage.py collectstatic
```

```
122 static files copied to '/code/staticfiles'.
```

If you look within your text editor, there is now a `staticfiles` directory with four subdirectories: `admin`, `css`, `images`, and `js`. The first one is the static assets of the Django `admin` app and the other three we specified. That's why there are 122 files copied over.

Bootstrap

Writing custom CSS for your website is a worthy goal and something I advise all software developers, even back-end ones, to try at some point. But practically speaking there is a reason front-end frameworks like [Bootstrap](#) exist: they save you a ton of time when starting a new project. Unless you have a dedicated designer to collaborate with, stick with a framework for the early iterations of your website.

In this section we'll add Bootstrap to our project alongside our existing `base.css` file. Typing all this out by hand would take a while and be error prone so this is a rare case where I advise simply copy/pasting from the [official source code](#).

Note that order matters here for both the CSS and JavaScript. The file will be loaded top-to-bottom so our `base.css` file comes *after* the Bootstrap CSS so our `h1` style overrides the Bootstrap default. At the bottom of the file, it's similarly important to load jQuery first, then PopperJS, and only then the Bootstrap JavaScript file.

Finally observe that a navigation header has been added to the project with basic logic so if a user is logged in, only the “Log Out” link is visible while a logged out user will see both “Log In” and “Sign Up” links.

Code

```
<!-- templates/_base.html -->

{% load static %}

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>{% block title %}Bookstore{% endblock title %}</title>

<meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/\4.3.1/css/bootstrap.min.css" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous">

<link rel="stylesheet" href="{% static 'css/base.css' %}">

</head>

<body>

<header>

<!-- Fixed navbar -->

<div class="d-flex flex-column flex-md-row align-items-center p-3 px-md-4 mb-3 bg-white border-bottom shadow-sm">

<a href="{% url 'home' %}" class="navbar-brand my-0 mr-md-auto font-weight-normal">Bookstore</h5>

<nav class="my-2 my-md-0 mr-md-3">

<a class="p-2 text-dark" href="#">About</a>

{% if user.is_authenticated %}

<a class="p-2 text-dark" href="{% url 'logout' %}">Log Out</a>

{% else %}

<a class="p-2 text-dark" href="{% url 'login' %}">Log In</a>

<a class="btn btn-outline-primary"
```

```
    href="#">Sign Up
```

```
{% endif %}
```

```
</nav>
```

```
</header>
```

```
<div class="container">
```

```
{% block content %}
```

```
{% endblock content %}
```

```
</div>
```

```
<!-- JavaScript -->
```

```
<!-- jQuery first, then Popper.js, then Bootstrap JS -->
```

```
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
```

```
integrity="sha384-q8i/X+965Dz00rT7abK41JStQIAqVgRVzbzo5smXKp4YfRvH+8\
```

```
abTE1Pi6jizo" crossorigin="anonymous"></script>
```

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/\
```

```
umd/popper.min.js" integrity="sha384-U02eT0CpHqdSJQ6hJty5KVphtPhzWj9W01\
```

```
clHTMGa3JDZwrnQq4sF86dIHNDz0W1" crossorigin="anonymous"></script>
```

```
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/\
```

```
bootstrap.min.js" integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6V\
```

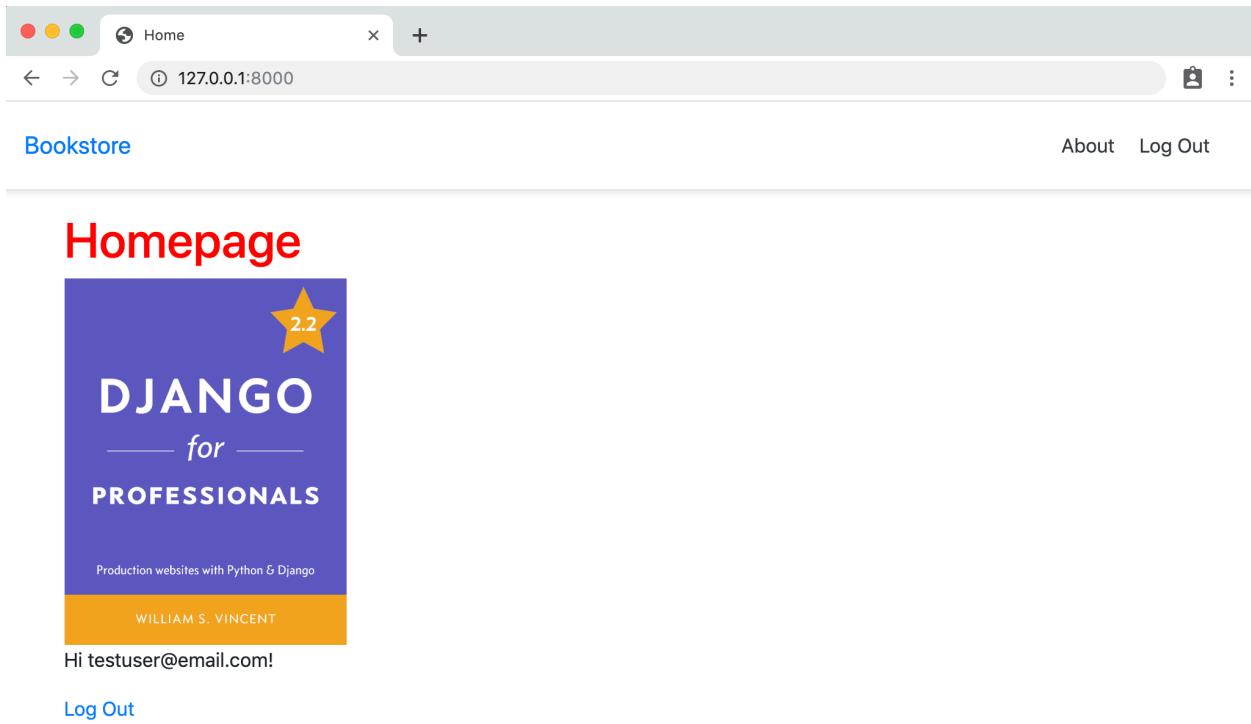
```
rjIEaFF/nJGzIxFDsf4x0xIM+B07jRM" crossorigin="anonymous"></script>
```

```
</body>
```

```
</html>
```

It's best not to attempt to type this code. Instead copy and paste it from [the official repo](#). Then make one change: on line 18 change the href for the about page to # not `{% url 'about' %}`. We'll add the about page URL route in the next section.

If you refresh the homepage after making these changes it should look as follows:



Homepage with Bootstrap

About Page

Did you notice the navbar link for an About page? Trouble is the page and the link don't exist yet. But because we already have a handy `pages` app it's quite quick to make one.

Since this will be a static page we don't need a database model involved. However we will need a template, view, and url. Let's start with the template called `about.html`.

Command Line

```
$ touch templates/about.html
```

The page will literally just say "About Page" for now while inheriting from `_base.html`.

Code

```
<!-- templates/about.html -->

{% extends '_base.html' %}

{% block title %}About{% endblock title %}

{% block content %}

<h1>About Page</h1>

{% endblock content %}
```

The view can rely on Django's built-in `TemplateView` just like our homepage.

Code

```
# pages/views.py

from django.views.generic import TemplateView

class HomePageView(TemplateView):
    template_name = 'home.html'

class AboutPageView(TemplateView): # new
    template_name = 'about.html'
```

And the URL path will be pretty similar as well. Set it to `about/`, import the appropriate view, and provide a URL name of `about`.

Code

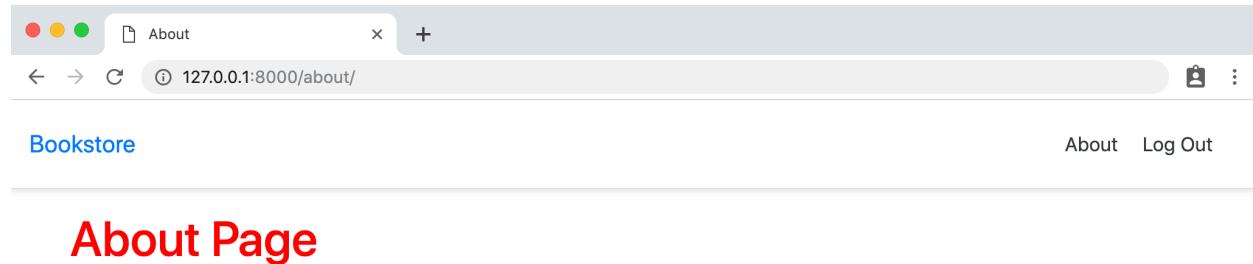
```
# pages/urls.py

from django.urls import path

from .views import HomePageView, AboutPageView # new

urlpatterns = [
    path('about/', AboutPageView.as_view(), name='about'), # new
    path('', HomePageView.as_view(), name='home'),
]
```

If you go now to <http://127.0.0.1:8000/about/> you can see the About page.



About Page

As a final step, update the link in the navbar to the page. Because we provided a name in the URL path of `about` that's what we'll use.

On line 18 of `_base.html` change the line with the About page link to the following:

Code

```
<!-- templates/_base.html -->  
<nav class="my-2 my-md-0 mr-md-3">  
  <a class="p-2 text-dark" href="{% url 'about' %}">About</a>  
  {% if user.is_authenticated %}
```

Django Crispy Forms

One last update concerns our forms. The popular 3rd party package [django-crispy-forms](#) provides a host of welcome upgrades.

We'll follow the usual pattern to install it which is: install within Docker, stop our Docker container and then rebuild it.

Command Line

```
$ docker-compose exec web pipenv install django-crispy-forms==1.7.1  
$ docker-compose down  
$ docker-compose up -d --build
```

Since we're using a volume for our PostgreSQL database it persists even when we stop, rebuild, and start our Docker containers. An alternate approach would be to use [fixtures](#) to quickly load data.

Now add `crispy_forms` to the `INSTALLED_APPS` setting. A nice additional feature is to specify `bootstrap4` under `CRISPY_TEMPLATE_PACK` which will provide pre-styled forms for us.

Code

```
# bookstore_project/settings.py

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # Third-party
    'crispy_forms', # new

    # Local
    'users.apps.UsersConfig',
    'pages.apps.PagesConfig',
]

# django-crispy-forms
CRISPY_TEMPLATE_PACK = 'bootstrap4' # new
```

To use Crispy Forms we load `crispy_forms_tags` at the top of a template and add `{{ form|crispy }}` to replace `{{ form.as_p}}` for displaying form fields. We will take this time to also add Bootstrap styling to the Submit button.

Start with `templates/signup.html`. Make the updates below.

Code

```
<!-- templates/signup.html -->  
{% extends '_base.html' %}  
{% load crispy_forms_tags %}  
  
{% block title %}Sign Up{% endblock title %}  
  
{% block content %}  
<h2>Sign Up</h2>  
<form method="post">  
    {% csrf_token %}  
    {{ form|crispy }}  
    <button class="btn btn-success" type="submit">Sign Up</button>  
</form>  
{% endblock content %}
```

The screenshot shows a web browser window with the title "Sign Up" and the URL "127.0.0.1:8000/accounts/signup/". The page has a header with "Bookstore" on the left, "About" and "Log In" on the right, and a "Sign Up" button highlighted with a blue border. Below the header is a large "Sign Up" heading. The form consists of several input fields: "Email address", "Username*", "Password*", "Password confirmation*", and a password strength indicator. A note below the username field specifies character requirements. A "Sign Up" button is at the bottom.

Email address

Username*

Required. 150 characters or fewer. Letters, digits and @./+/-/_ only.

Password*

- Your password can't be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can't be a commonly used password.
- Your password can't be entirely numeric.

Password confirmation*

Enter the same password as before, for verification.

Sign Up

Sign Up Page with Crispy Forms

Update `login.html` as well with `crispy_forms_tags` at the top and `{{ form|crispy }}` in the form.

Code

```
<!-- templates/registration/login.html -->
{% extends '_base.html' %}

{% load crispy_forms_tags %}

{% block title %}Log In{% endblock title %}

{% block content %}

<h2>Log In</h2>

<form method="post">

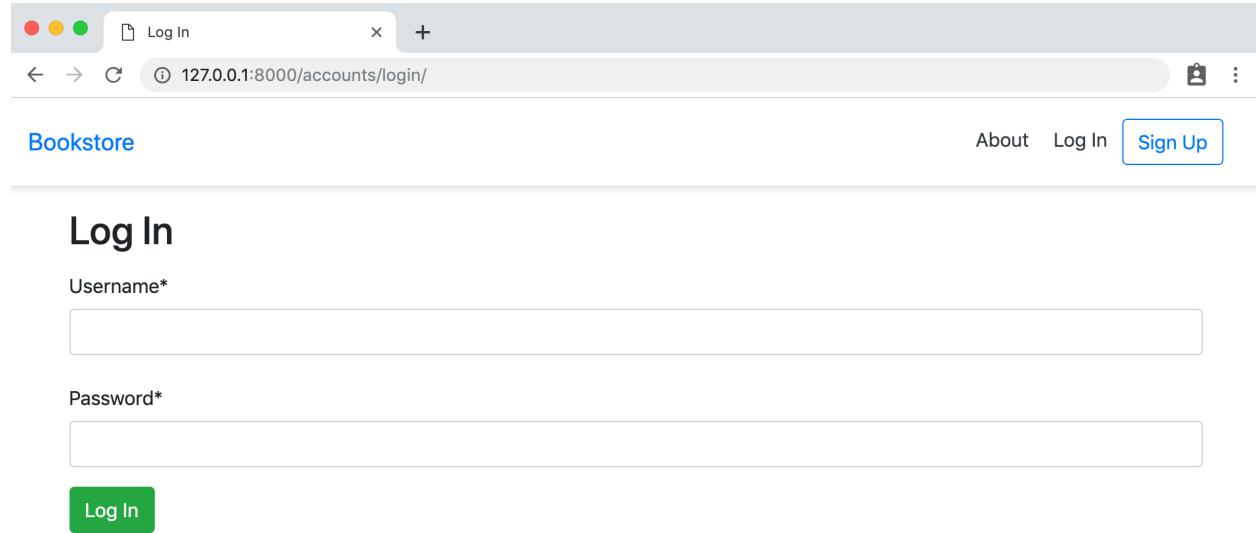
    {% csrf_token %}

    {{ form|crispy }}

    <button class="btn btn-success" type="submit">Log In</button>

</form>

{% endblock content %}
```



The screenshot shows a web browser window with the following details:

- Address Bar:** Shows the URL `127.0.0.1:8000/accounts/login/`.
- Header:** The page has a header with "Bookstore" on the left, "About", "Log In", and "Sign Up" on the right. The "Sign Up" button is highlighted with a blue border.
- Content:** The main content area is titled "Log In". It contains two input fields: one for "Username*" and one for "Password*". Both fields have placeholder text and are enclosed in a light gray box.
- Buttons:** A green "Log In" button is located at the bottom of the form.

Log In Page with Crispy Forms

Tests

Time for tests. As a recap we have added an About page to our site and the ability for an admin to upload images. When we add an image-upload form later on we'll properly test that but for now it's enough to add basic tests for our About page. They will be very similar to those we did for our homepage.

Code

```
# pages/tests.py

from django.test import SimpleTestCase
from django.urls import reverse, resolve

from .views import HomePageView, AboutPageView # new

class HomepageTests(SimpleTestCase):
    ...

class AboutPageTests(SimpleTestCase): # new

    def setUp(self):
        url = reverse('about')
        self.response = self.client.get(url)

    def test_aboutpage_status_code(self):
        self.assertEqual(self.response.status_code, 200)

    def test_aboutpage_template(self):
```

```
    self.assertTemplateUsed(self.response, 'about.html')

def test_aboutpage_contains_correct_html(self):
    self.assertContains(self.response, 'About Page')

def test_aboutpage_does_not_contain_incorrect_html(self):
    self.assertNotContains(
        self.response, 'Hi there! I should not be on the page.')

def test_aboutpage_url_resolves_aboutpageview(self):
    view = resolve('/about/')
    self.assertEqual(
        view.func.__name__,
        AboutPageView.as_view().__name__
    )
```

Run the tests.

Command Line

```
$ docker-compose exec web python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
....
```

```
Ran 15 tests in 0.433s
```

OK

```
Destroying test database for alias 'default'...
```

Git

Check the `status` of our changes in this chapter, add them all, and then provide a commit message.

Command Line

```
$ git status  
$ git add .  
$ git commit -m 'ch6'
```

As always you can compare your code with the [official code on Github](#) if there are any issues.

Conclusion

Static assets are a core part of every website and in Django we have to take a number of additional steps so they are compiled and hosted efficiently in production. Later on in the book we'll learn how to use a dedicated content delivery network (CDN) for hosting and displaying our project's static files.

Chapter 7: Advanced User Registration

At this point we have the standard Django user registration implemented. But often that's just the starting point on professional projects. What about customizing things a bit? For example, Django's default username/email/password pattern is somewhat dated these days. It's far more common to simply require email/password for sign up and log in. And really every part of the authentication flow—the forms, emails, pages—can be customized if so desired.

Another major factor in many projects is the need for social authentication, that is handling sign up and log in via a third-party service like Google, Facebook, and so on.

We could implement our own solutions here from scratch but there are some definite risks: user registration is a complex area with many moving parts and one area where we really do not want to make a security mistake.

For this reason, many professional Django developers rely on the popular third-party [django-allauth](#). Adding any third party package should come with a degree of caution since you *are* adding another dependency to your technical stack. It's important to make sure any package is both up-to-date and well tested. Fortunately [django-allauth](#) is both.

At the cost of a little bit of *magic* it addresses all of these concerns and makes customization much, much easier.

django-allauth

Start by installing `django-allauth`. Because we're using `Pipenv` we want to avoid conflicts with the `Pipfile.lock` so we'll install it within Docker first, then stop Docker,

and rebuild our image with the `--build` flag which prevents the default image caching and ensures that our entire image is built from scratch.

Command Line

```
$ docker-compose exec web pipenv install django-allauth==0.38.0
$ docker-compose down
$ docker-compose up -d --build
```

Our website will still function the same as before since we haven't explicitly told Django about this new `django-allauth` package. To do that we need to update the `INSTALLED_APPS` config within our `settings.py` file adding Django's built-in, but optional, [sites framework](#), as well as `allauth` and its account feature `allauth.account`.

Django's sites framework is a powerful feature that allows one Django project to control multiple sites. Given we only have one site in our project, we'll set the `SITE_ID` to `1`. If we added a second site it would have an ID of `2`, a third site would have an ID of `3`, and so on.

Code

```
# bookstore_project/settings.py

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.sites', # new

    # Third-party
```

```
'crispy_forms',
'allauth', # new
'allauth.account', # new

# Local
'users.apps.UsersConfig',
'pages.apps.PagesConfig',
]

# django-allauth config
SITE_ID = 1 # new
```

AUTHENTICATION_BACKENDS

The `settings.py` file created by Django for any new project contains a number of explicit settings—those that we see in the file already—as well as a longer additional list of implicit settings that exist but aren’t visible. This can be confusing at first. The complete list of settings configurations [is available here](#).

An example is the `AUTHENTICATION_BACKENDS` setting. Under the hood Django sets this to `'django.contrib.auth.backends.ModelBackend'` which is used when Django attempts to authenticate a user.

We could add the following line to `settings.py` and the current behavior would remain unchanged:

Code

```
AUTHENTICATION_BACKENDS = (
    'django.contrib.auth.backends.ModelBackend',
)
```

However for `django-allauth` we need to add its specific authentication options, too, which will allow us to switch over to using login via e-mail in a moment. So at the bottom of your `settings.py` file add the following section:

Code

```
# bookstore_project/settings.py

# django-allauth config
SITE_ID = 1

AUTHENTICATION_BACKENDS = (
    'django.contrib.auth.backends.ModelBackend',
    'allauth.account.auth_backends.AuthenticationBackend', # new
)
```

EMAIL_BACKEND

Another configuration implicitly set is `EMAIL_BACKEND`. By default Django will look for a configured `SMTP server` to send emails.

`dango-allauth` will send such an email upon a successful user registration, which we can and will customize later, but since we don't yet have a SMTP server properly configured, it will result in an error.

The solution, for now, is to have Django output any emails to the command line console instead. Thus we can override the default, implicit config by using `console` instead of `smtp`. Add this at the bottom of the `settings.py` file.

Code

```
# bookstore_project/settings.py

# django-allauth config

SITE_ID = 1

AUTHENTICATION_BACKENDS = (
    'django.contrib.auth.backends.ModelBackend',
    'allauth.account.auth_backends.AuthenticationBackend',
)

EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend' # new
```

ACCOUNT_LOGOUT_REDIRECT

There's one more subtle change to make to our configurations at this time. If you look at the [configurations page](#) again you'll see there is a setting for `ACCOUNT_LOGOUT_REDIRECT` that defaults to the path of the homepage at `/`.

In our current `settings.py` file we have the following two lines for redirects which point to the homepage via its URL name of `'home'`.

Code

```
# bookstore_project/settings.py

LOGIN_REDIRECT_URL = 'home'

LOGOUT_REDIRECT_URL = 'home'
```

The issue is that `django-allauth`'s `ACCOUNT_LOGOUT_REDIRECT` actually overrides the built-in `LOGOUT_REDIRECT_URL`, however, since they both point to the homepage this change may not be apparent. To future-proof our application since maybe we don't want to always redirect to the homepage on logout, we should be explicit here with the logout redirect.

We can also move the two redirect lines under our `django-allauth config` section. This is what the entire `django-allauth config` section should look like at this time.

Code

```
# bookstore_project/settings.py

# django-allauth config

LOGIN_REDIRECT_URL = 'home'

ACCOUNT_LOGOUT_REDIRECT = 'home' # new

SITE_ID = 1

AUTHENTICATION_BACKENDS = (
    'django.contrib.auth.backends.ModelBackend',
    'allauth.account.auth_backends.AuthenticationBackend',
)

EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

Given that we have made many changes to our `bookstore_project/settings.py` file let's now run `migrate` to update our database.

Command Line

```
$ docker-compose exec web python manage.py migrate
Operations to perform:
  Apply all migrations: account, admin, auth, contenttypes, sess
  ions, sites, users
Running migrations:
  Applying account.0001_initial... OK
  Applying account.0002_email_max_length... OK
  Applying sites.0001_initial... OK
  Applying sites.0002_alter_domain_unique... OK
```

URLs

We also need to swap out the built-in auth app URLs for django-allauth's own allauth app. We'll still use the same accounts/ URL path, however, since we'll be using django-allauth's templates and routes for sign up we can delete the URL path for our users app, too.

Code

```
# bookstore_project/urls.py

from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    # Django admin
    path('admin/', admin.site.urls),
    # User management
    path('accounts/', include('django.contrib.auth.urls')),
    path('users/', include('users.urls')),
]
```

```
path('accounts/', include('allauth.urls')), # new  
  
# Local apps  
path('', include('pages.urls')),  
]
```

At this point we could further delete `users/urls.py` and `users/views.py` which were both created solely for our hand-written sign up page and are no longer being used.

Templates

Django's `auth` app looks for templates within a `templates/registration` directory, but `djano-allauth` prefers they be located within a `templates/account` directory. So we will create a new `templates/account` directory and then copy over our existing `login.html` and `signup.html` templates into it.

Command Line

```
$ mkdir templates/account  
$ mv templates/registration/login.html templates/account/login.html  
$ mv templates/signup.html templates/account/signup.html
```

It's easy to add an `s` onto `account` here by accident, but don't or you'll get an error. The correct directory is `templates/account/`.

We can delete the `templates/registration` directory at this point since it is no longer needed.

Command Line

```
$ rm -r templates/registration
```

`rm` means remove and `r` means do it recursively, which is necessary whenever you are dealing with a directory. If you'd like further information on this command you can type `man rm` to read the manual.

The last step is to update the URL links within both `templates/_base.html` and `templates/home.html` to use django-allauth's URL names rather than Django's. We do this by adding an `account_` prefix so Django's '`logout`' will now be '`account_logout`', '`login`' will be '`account_login`', and `signup` will be `account_signup`.

Code

```
<!-- templates/_base.html -->  
...  
<nav class="my-2 my-md-0 mr-md-3">  
  <a class="p-2 text-dark" href="{% url 'about' %}">About</a>  
  {% if user.is_authenticated %}  
    <a class="p-2 text-dark" href="{% url 'account_logout' %}">Log Out</a>  
  {% else %}  
    <a class="p-2 text-dark" href="{% url 'account_login' %}">Log In</a>  
    <a class="btn btn-outline-primary"  
       href="{% url 'account_signup' %}">Sign Up</a>  
  {% endif %}  
</nav>  
...
```

Code

```
<!-- templates/home.html -->
{% extends '_base.html' %}
{% load static %}

{% block title %}Home{% endblock title %}

{% block content %}
<h1>Homepage</h1>

{% if user.is_authenticated %}
<p>Hi {{ user.email }}!</p>
<p><a href="{% url 'account_logout' %}">Log Out</a></p>
{% else %}
<p>You are not logged in</p>
<p><a href="{% url 'account_login' %}">Log In</a> | 
<a href="{% url 'account_signup' %}">Sign Up</a></p>
{% endif %}
{% endblock content %}
```

And we're done!

Log In

If you refresh the homepage at <http://127.0.0.1:8000> and then click on the “Log in” link you’ll see an updated page.

The screenshot shows a web browser window with the title "Log In". The address bar displays "127.0.0.1:8000/accounts/login/". The page content is titled "Log In". It contains two input fields: "Username*" and "Password*". Below these fields is a checkbox labeled "Remember Me". At the bottom is a green "Log In" button. The top navigation bar includes links for "About", "Log In", and "Sign Up".

Log In Page

Note the new “Remember Me” box option. This is the first of many [configurations](#) that `django-allauth` provides. The default `None` asks the user if they want their session to be remembered so they don’t have to log in again. It can also be set to `False` to not remember or `True` to always remember. We’ll choose `True` which is how a traditional Django log in page would work.

Under our `# django-allauth config` section of the `bookstore_project/settings.py` file add a new line for this.

Code

```
# bookstore_project/settings.py  
# django-allauth config  
...  
ACCOUNT_SESSION_REMEMBER = True # new
```

Refresh the “Log In” page and the box is gone!

The screenshot shows a web browser window with the address bar displaying "127.0.0.1:8000/accounts/login/". The page title is "Log In". The navigation bar includes links for "About", "Log In", and "Sign Up". The main content area contains fields for "Username*" and "Password*", both with placeholder text. A green "Log In" button is at the bottom.

Log In Page No Box

If you try out the log in form with your superuser account it will redirect back to the homepage with a welcome message. Click on the “Log Out” link.

The screenshot shows a web browser window with the address bar displaying "127.0.0.1:8000/accounts/logout/". The page title is "Sign Out". The main content area displays a "Messages:" section with a success message: "Successfully signed in as wsv." It also shows a "Menu:" section with links for "Change E-mail" and "Sign Out".

Sign Out

Are you sure you want to sign out?

[Sign Out](#)

Log Out Page

Rather than directly log us out django-allauth has an intermediary “Log Out” page which we can customize to match the rest of our project.

Log Out

Update the default Log Out template by creating a `templates/account/logout.html` file to override it.

Command Line

```
$ touch templates/account/logout.html
```

Like our other templates it will extend `_base.html` and include Bootstrap styling on the submitted button.

Code

```
<!-- templates/account/logout.html -->

{% extends '_base.html' %}

{% load crispy_forms_tags %}

{% block title %}Log out{% endblock %}

{% block content %}

<div class="container">

<h1>Log Out</h1>

<p>Are you sure you want to sign out?</p>

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

  {% csrf_token %}

  {{ form|crispy }}

  <button class="btn btn-danger" type="submit">Log Out</button>

</form>

</div>

{% endblock content %}
```

Go ahead and refresh the page.

The screenshot shows a web browser window with the following details:

- Address Bar:** Shows the URL `127.0.0.1:8000/accounts/logout/`.
- Page Title:** The page title is "Bookstore".
- Header:** Includes links for "About" and "Log Out".
- Main Content:** A large red header says "Sign Out". Below it, a message asks "Are you sure you want to sign out?". A prominent red button labeled "Log Out" is at the bottom.
- Page Footer:** A caption "Custom Log Out Page" is centered at the bottom of the page.

Sign Up

At the top of our website, in the nav bar, click on link for “Sign Up” which has Bootstrap and `django-crispy-forms` styling.

The screenshot shows a web browser window with the following details:

- Address Bar:** Shows the URL `127.0.0.1:8000/accounts/signup/`.
- Page Title:** "Sign Up" (part of the address bar).
- Page Content:**
 - Bookstore Header:** "Bookstore" on the left, "About" and "Log In" on the right.
 - Sign Up Form:** A form with four input fields:
 - Username***: An input field labeled "Username".
 - E-mail (optional)**: An input field labeled "E-mail address".
 - Password***: An input field labeled "Password".
 - Password (again)***: An input field labeled "Password (again)".
 - Sign Up Button:** A green button labeled "Sign Up".

Sign Up Page

An optional customization we can make via `django-allauth` is to only ask for a password once. Since we'll configure password change and reset options later, there's less of a risk that a user who types in the password incorrectly will be locked out of their account.

This change is, if you look again at the [django-allauth configuration options](#), a one-liner.

Code

```
# bookstore_project/settings.py  
# django-allauth config  
...  
ACCOUNT_SIGNUP_PASSWORD_ENTER_TWICE = False # new
```

Refresh the page and the form will update itself to remove the additional password line.

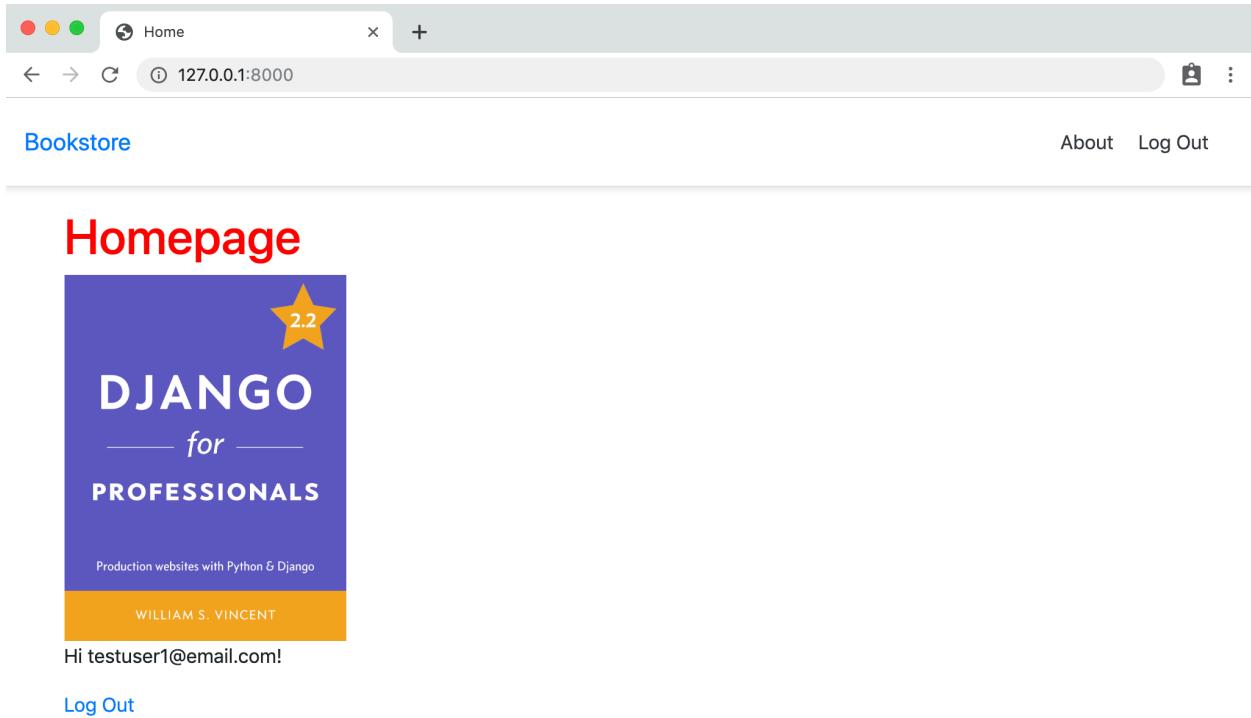
The screenshot shows a web browser window with the following details:

- Header:** The browser title bar displays "Sign Up". The address bar shows the URL "127.0.0.1:8000/accounts/signup/".
- Page Header:** The page has a header with "Bookstore" (blue), "About" (grey), "Log In" (grey), and a "Sign Up" button (blue outline).
- Content:** The main content area is titled "Sign Up". It contains three input fields:
 - "Username*" with a placeholder "Username" in a grey box.
 - "E-mail (optional)" with a placeholder "E-mail address" in a grey box.
 - "Password*" with a placeholder "Password" in a grey box.
- Buttons:** At the bottom left is a green "Sign Up" button.

Sign Up with Single Password

Now create a new user to confirm everything works. We can call the user `testuser1`, use `testuser1@email.com` as email, and `testpass123` as the password.

Upon submit it will redirect you to the homepage.



testuser Homepage

Remember how we configured email to output to the console? django-allauth automatically sends an email upon registration which we can view by typing `docker-compose logs`.

Command Line

```
$ docker-compose logs  
...  
web_1 | Content-Type: text/plain; charset="utf-8"  
web_1 | MIME-Version: 1.0  
web_1 | Content-Transfer-Encoding: 7bit  
web_1 | Subject: [example.com] Please Confirm Your E-mail Address  
web_1 | From: webmaster@localhost  
web_1 | To: testuser@email.com  
web_1 | Date: Sat, 13 Jul 2019 14:04:15 -0000
```

```
web_1 | Message-ID: <155266195771.15.17095643701553564393@cdab877c4af3>
web_1 |
web_1 | Hello from example.com!
web_1 |
web_1 | You're receiving this e-mail because user testuser has given yours as
an e-mail address to connect their account.
web_1 |
web_1 | To confirm this is correct, go to http://127.0.0.1:8000/accounts/confirm-email
web_1 | MQ:1h4oIn:GYETeK5dRClGjcgA8NbuOoyvafA/
web_1 |
web_1 | Thank you from example.com!
web_1 | example.com
web_1 | -----
...
-----
```

There it is. Later on we'll customize this message and configure a proper email service to send it to actual users.

Admin

Log in to the admin with your superuser account at <http://127.0.0.1:8000/admin/> and we can see it, too, has changed now that django-allauth is involved.

The screenshot shows the Django administration interface at the URL `127.0.0.1:8000/admin/`. The top navigation bar includes links for Site administration, Django site, and a user icon. The main content area is titled "Django administration" and "WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT". On the left, there are four main sections: "ACCOUNTS" (Email addresses, Groups), "AUTHENTICATION AND AUTHORIZATION" (Groups), "SITES" (Sites), and "USERS" (Users). Each section has "Add" and "Change" buttons. To the right, there are two boxes: "Recent actions" (empty) and "My actions" (None available).

Admin Homepage

There are two new sections: `Accounts` and `Sites` courtesy of our recent work. If you click on the `Users` section we see our traditional view that shows the three current user accounts.

The screenshot shows the Django administration interface for the 'CustomUser' model. The title bar says 'Select user to change | Django'. The URL is '127.0.0.1:8000/admin/users/customuser/'. The main header says 'Django administration' and 'WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT'. Below it, 'Home > Users > Users' is shown. A search bar and a 'Search' button are at the top left. On the right, there's a 'ADD USER +' button. A 'FILTER' sidebar on the right lists filters for 'By staff status' (All, Yes, No), 'By superuser status' (All, Yes, No), and 'By active' (All, Yes, No). The main content area shows a table with columns 'EMAIL ADDRESS' and 'USERNAME'. It lists three users:

EMAIL ADDRESS	USERNAME
testuser@email.com	testuser
testuser1@email.com	testuser1
will@wsvincent.com	wsv

Below the table, it says '3 users'.

Admin Users

Go back to the homepage and click on the section for `Sites` to see what the Django sites framework provides. We'll update both the Domain Name and the Display Name in a later chapter on configuring email.

The screenshot shows the Django administration interface for managing sites. The title bar says "Select site to change" and the URL is "127.0.0.1:8000/admin/sites/site/". The main header includes "Django administration", "WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT", and navigation links "Home > Sites > Sites". Below this is a search bar with a magnifying glass icon and a "Search" button. A "SELECT" dropdown menu is open, showing options like "-----", "All", and "Selected". The table lists one site entry:

<input type="checkbox"/>	DOMAIN NAME	DISPLAY NAME
<input type="checkbox"/>	example.com	example.com

Below the table, it says "1 site".

Admin Sites

Email Only Login

It's time to really use django-allauth's [extensive list of configurations](#) by switching over to using just email for login, not username. This requires a few changes. First we'll make `username` not required, but set `email` instead to required. Then we'll require `email` to be unique and the authentication method of choice.

Code

```
# bookstore_project/settings.py

# django-allauth config

...
ACCOUNT_USERNAME_REQUIRED = False # new
ACCOUNT_AUTHENTICATION_METHOD = 'email' # new
ACCOUNT_EMAIL_REQUIRED = True # new
ACCOUNT_UNIQUE_EMAIL = True # new
```

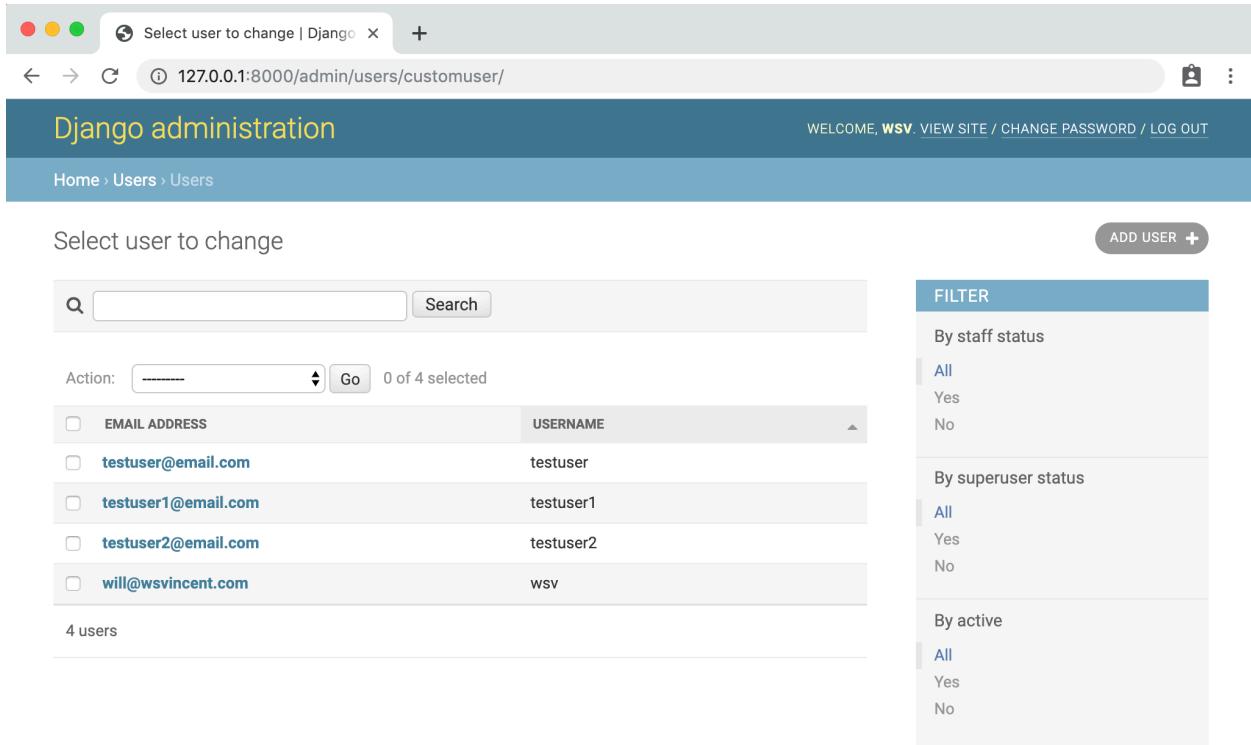
Navigate back to the homepage and click on “Log Out” since you’ll be logged in with your superuser account. Then click on the navbar link for “Sign Up” and create an account for `testuser2@email.com` with `testpass123` as the password.

The screenshot shows a web browser window with the following details:

- Header:** Shows three colored dots (red, yellow, green) on the left, followed by a "Sign Up" button, a close button ("x"), and a plus sign ("+"). Below the header is a URL bar with the address `127.0.0.1:8000/accounts/signup/`.
- Navigation:** Below the header are standard browser navigation buttons for back, forward, and refresh.
- Page Title:** The title bar says "Bookstore".
- User Links:** A "Sign Up" button is highlighted with a blue border, while "About" and "Log In" are in regular black text.
- Form Fields:** The "Sign Up" form has two fields:
 - E-mail***: The input field contains `testuser2@email.com`.
 - Password***: The input field contains a series of dots (...).
- Buttons:** A green "Sign Up" button is located at the bottom of the form.

Sign Up Email Only

After being redirected to the homepage upon success, now go into the admin at `http://127.0.0.1:8000/admin/` to inspect what actually happened. Log in with your superuser account and navigate to the `Users` section.



The screenshot shows the Django administration interface for the 'CustomUser' model. The title bar says 'Select user to change | Django'. The URL is '127.0.0.1:8000/admin/users/customuser/'. The main area is titled 'Django administration' with a 'WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT' message. The breadcrumb navigation shows 'Home > Users > Users'. Below this, a search bar and a 'Search' button are present. A 'Select user to change' dropdown menu is open, showing four users: 'testuser' (email: testuser@email.com), 'testuser1' (email: testuser1@email.com), 'testuser2' (email: testuser2@email.com), and 'wsv' (email: will@wsvincent.com). Each user has a checkbox next to their email address. The right sidebar contains filter options for 'By staff status' (All, Yes, No), 'By superuser status' (All, Yes, No), and 'By active' (All, Yes, No).

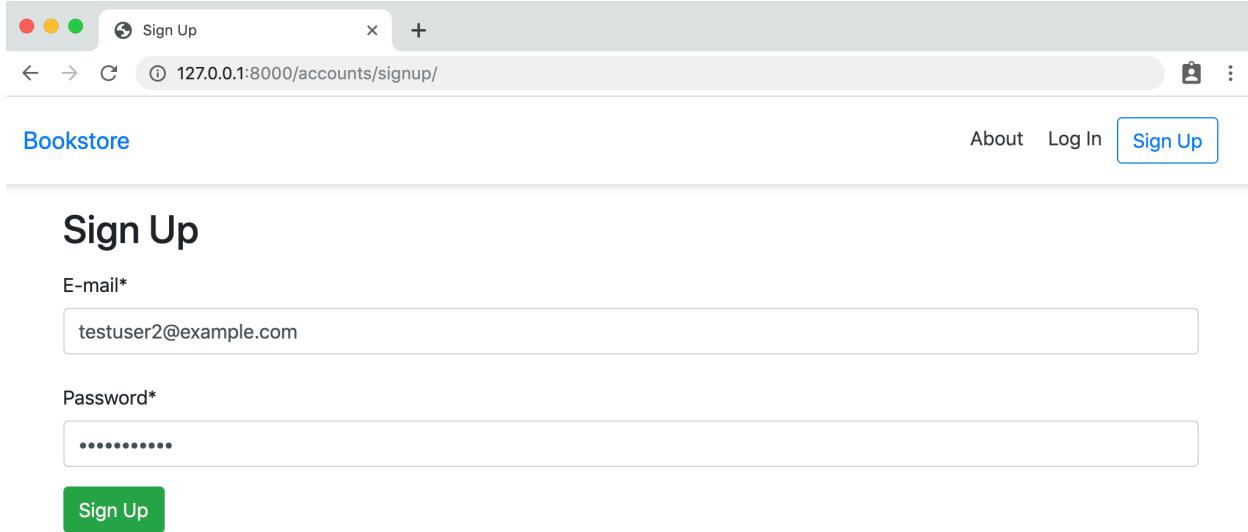
Admin Users

We can see that `django-allauth` automatically populated a username for us based on the email part before the @. This is because our underlying `CustomUser` model still has a `username` field. We didn't delete it.

While this approach may seem a little hackish, but in fact it works just fine. Fully removing the `username` from the custom user model requires the use of `AbstractBaseUser`, which is an additional, optional step some developers take. It requires far more coding and understanding so it is not recommended unless you really know your way around Django's authentication system!

There is, however, an edge case here that we should confirm which is: what happens if we have `testuser2@email.com` and then a sign up for `testuser2@example.com`? Wouldn't that result in a username of `testuser2` for both which would cause a conflict? Let's try it out!

Log out of the admin, go to the Sign Up Page again and create an account for testuser2@example.com.



The screenshot shows a web browser window with the title "Sign Up". The address bar displays "127.0.0.1:8000/accounts/signup/". The page content includes a "Bookstore" logo, navigation links for "About" and "Log In", and a prominent blue "Sign Up" button. The main form has two input fields: "E-mail*" containing "testuser2@example.com" and "Password*" containing "*****". A green "Sign Up" button is at the bottom of the form.

Sign Up Form

Now log back into the admin and go to our `Users` section.

The screenshot shows the Django administration interface for managing users. The top navigation bar includes the title "Select user to change | Django", the URL "127.0.0.1:8000/admin/users/customuser/", and links for "WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT". Below the header, the "Django administration" logo is visible, along with the breadcrumb trail "Home > Users > Users". A search bar and a "Search" button are present. On the right, there is an "ADD USER" button. The main content area displays a table of users with columns for "EMAIL ADDRESS" and "USERNAME". The table lists five users: testuser@email.com (username testuser), testuser1@email.com (username testuser1), testuser2@email.com (username testuser2), testuser2@example.com (username testuser249), and will@wsvincent.com (username wsv). A "FILTER" sidebar on the right allows filtering by staff status (All, Yes, No), superuser status (All, Yes, No), and active status (All, Yes, No). The footer of the screenshot area contains the text "5 users".

Admin Users

django-allauth automatically adds a two-digit string to the username. In this case it is 49 so testuser2 becomes testuser249. This two-digit string will be randomly generated for us.

Tests

Time for tests. Like any good third-party package django-allauth comes with its own tests so we don't need to re-test its core functionality, just confirm that our project works as expected.

If you run our current test suite there are 3 errors related to `SignupPageTests` since we're using `django-allauth` now for this rather than our own views, forms, and urls.

Command Line

```
$ docker-compose exec web python manage.py test  
...  
Ran 15 tests in 0.363s  
  
FAILED (errors=3)
```

Let's update the tests. The first issue is that `signup` is no longer the correct URL name, instead we're using `account_signup` which is the name `django-allauth` provides. How did I know that? I looked at [the source code](#) and found the URL name.

Another change is the location of the `signup.html` template which is now located at `account/signup.html`.

We're also not using `CustomUserCreationForm` anymore, but instead, that provided by `django-allauth` so we can remove that test. Remove as well the imports for `CustomUserCreationForm` and `SignupPageView` at the top of the file.

Code

```
# users/tests.py  
  
from django.contrib.auth import get_user_model  
from django.test import TestCase  
from django.urls import reverse, resolve  
  
  
  
class CustomUserTests(TestCase):  
    ...  
  
  
  
class SignupTests(TestCase): # new
```

```
username = 'newuser'

email = 'newuser@email.com'

def setUp(self):
    url = reverse('account_signup')
    self.response = self.client.get(url)

def test_signup_template(self):
    self.assertEqual(self.response.status_code, 200)
    self.assertTemplateUsed(self.response, 'account/signup.html')
    self.assertContains(self.response, 'Sign Up')
    self.assertNotContains(
        self.response, 'Hi there! I should not be on the page.')

def test_signup_form(self):
    new_user = get_user_model().objects.create_user(
        self.username, self.email)
    self.assertEqual(get_user_model().objects.all().count(), 1)
    self.assertEqual(get_user_model().objects.all()
                    [0].username, self.username)
    self.assertEqual(get_user_model().objects.all()
                    [0].email, self.email)
```

Run the tests again.

Command Line

```
$ docker-compose exec web python manage.py test  
Creating test database for alias 'default'...  
System check identified no issues (0 silenced).  
.....  
-----  
Ran 14 tests in 0.410s
```

OK

```
Destroying test database for alias 'default'...
```

Social

If you want to add social authentication it's just a few settings. I have a [complete tutorial online](#) for integrating Github. The process is similar for Google, Facebook, and all the rest django-allauth supports. [Here is the complete list of providers.](#)

Git

As always commit the code changes with Git.

Command Line

```
$ git status  
$ git add .  
$ git commit -m 'ch7'
```

And if there are any issues, compare with the [official source code on Github](#).

Conclusion

We now have a user registration flow that works and can be quickly extended into social authentication if needed. In the next chapter we'll add environment variables to our project for greater security and flexibility.

Chapter 8: Environment Variables

[Environment variables](#) are variables that can be loaded into the operating environment of a project at run time as opposed to hard coded into the codebase itself. They are considered an integral part of the popular [Twelve-Factor App Design](#) methodology and a Django best practice because they allow a greater level of security and simpler local/production configurations.

Why greater security? Because we can store truly secret information—database credentials, API keys, and so on—separate from the actual code base. This is a good idea because using a version control system, like `git`, means that it only takes one bad commit for credentials to be added in there forever. Which means that *anyone* with access to the codebase has full control over the project. This is very, very dangerous. It's much better to limit who has access to the application and environment variables provide an elegant way to do so.

A secondary advantage is that environment variables greatly simplify having different environments for both local and production code. As we will see, there are a number of setting configurations that Django uses by default intended to make local development easier, but which must be changed once the same project is ready for production.

In a non-Docker environment the current best practice is to use [django-environ](#), however, since we're using Docker it's possible to add environment variables directly via our `docker-compose.yml` file which is what we'll do.

.env files

Note that it is also possible to use separate `.env` files to store the environment variables and reference them in a `docker-compose.yml` file. A file that begins with a period `.` is known as a [hidden file](#) and frequently used for configurations. It's not really hidden; the file is still there in the directory. However if you type `ls`, the default listing of files command to see the contents of a directory, hidden files will not appear. But they are still there and accessible if you add the flag `ls -la`.

The advantage of a `.env` file is that it can be removed from Git via a separate `.gitignore` file. However in practice chaining together multiple `.env` files becomes quite complicated and while it might make sense on a larger project with many developers and many levels of access, we will stick to the more straightforward approach of plugging environment variables directly into a `docker-compose.yml` file in this book.

SECRET_KEY

For our first environment variable let's start with the `SECRET_KEY` configuration in the `bookstore_project/settings.py` file. This key is a randomly generated string used for [cryptographic signing](#) and created whenever the `startproject` command is run.

There is a two-step process for adding environment variables: first we add the values to our `docker-compose.yml` file and then we replace the hardcoded `bookstore_project/settings.py` value with the environment variable.

Within the `docker-compose.yml` file start by adding a section called `environment` under `web` services. We will place all our environment variables here. Then add a line called `SECRET_KEY` that will = the desired value. This can be a little confusing because the `=` symbol can be included in secret keys! To make the structure crystal clear: if our

secret key were dog then the line would be `SECRET_KEY=dog`. If the secret key were `dog=abc` then the line would be `SECRET_KEY=dog=abc`. That's it!

Here is what my file looks like with the secret key generated for the project. Swap in your own secret key in place of it. Note that in the `bookstore_project/settings.py` file the secret key will be surrounded by quotes '' to signify it as a string. **Do not** include the quotes when copying the secret key over into `docker-compose.yml`!

docker-compose.yml

```
version: '3.7'

services:
  web:
    build: .
    command: python /code/manage.py runserver 0.0.0.0:8000
    environment:
      - SECRET_KEY=p_o3vp1rg5)t^lxm9-43%0)s-=1qpeq%o7gfq+e4#*!t+_ev82
    volumes:
      - .:/code
    ports:
      - 8000:8000
    depends_on:
      - db

  db:
    image: postgres:11
    volumes:
      - postgres_data:/var/lib/postgresql/data/

volumes:
  postgres_data:
```

Step two swap out the hard coded secret key value in `bookstore_project/settings.py` for a link to the environment variable. If you look at the `bookstore_project/settings.py` file the very first line import's `os` from Python. Using `os.environ` allows us to reference environment variables which are supplied via `docker-compose.yml`.

Here's what your updated file should look like:

Code

```
# bookstore_project/settings.py  
SECRET_KEY = os.environ.get('SECRET_KEY')
```

It can be confusing when both an environment variable and the setting itself have the same name so to solidify the structure here, we could have called this environment variable `NEW_SECRET_KEY` in our `docker-compose.yml` file in which case the `bookstore_project/settings.py` line would have been `SECRET_KEY = os.environ.get('NEW_SECRET_KEY')`. However it is common to have the environment variable name match that of the setting it replaces.

The final step is to stop and re-start our Docker containers since they are designed to be stateless so when the state has changed—and environment variables are part of the state!—we need to quickly reload the containers to incorporate any new environment variables that have been set.

Command Line

```
$ docker-compose down  
$ docker-compose up -d
```

All set. You should be able to navigate to the webpage again, refresh it, and everything still works as before. If the environment variable hadn't loaded you'd see an error since a `SECRET_KEY` is required for any Django project. If that's the case run `docker-compose logs` from the command line to diagnose the issue.

DEBUG

Next up is `DEBUG` which is a boolean setting. By default Django sets this to `True` to help with debugging in local development, however, when it comes time to deploy a website in production this should be set to `False`.

In Chapter 17: Security we will learn how to create a `docker-compose-prod.yml` file with production-only configurations that sets this to `False`. Since we're using variables our `bookstore_project/settings.py` file can remain the same and we only need to change the docker compose reference. But for now, since we're still in local development mode, let's set `DEBUG` to `True`. We could also set this to `1` and `FALSE` to `0` which is a choice you'll see some developers make.

Update the `docker-compose.yml` file with a new environment variable for `DEBUG`.

`docker-compose.yml`

```
version: '3.7'

services:
  web:
    build: .
    command: python /code/manage.py runserver 0.0.0.0:8000
    environment:
      - SECRET_KEY=p_o3vp1rg5)t^lxm9-43%0)s-=1qpeq%o7gfq+e4#*!t+_ev82
      - DEBUG=1
    volumes:
      - .:/code
    ports:
      - 8000:8000
    depends_on:
```

```
- db

db:
  image: postgres:11
  volumes:
    - postgres_data:/var/lib/postgresql/data/

volumes:
  postgres_data:
```

Then update the `DEBUG` configuration within `bookstore_project/settings.py` to reference the environment variable now. Note the addition of Python's built-in `int` function and a default of `0`.

Code

```
# bookstore_project/settings.py

DEBUG = int(os.environ.get('DEBUG', default=0))
```

Remember to stop and start the Docker containers to load in the environment variables.

Command Line

```
$ docker-compose down
$ docker-compose up -d
```

Databases

It's possible and recommended to have multiple levels of users and permissions in your PostgreSQL database. But given this is a book on Django covering the topic properly is well beyond our scope however using environment variables for such secret information is a good ideas as well.

Git

Make sure to commit the code changes with Git.

Command Line

```
$ git status  
$ git add .  
$ git commit -m 'ch8'
```

If any issues crop up, compare your files against the [official source code on Github](#).

Conclusion

Adding environment variables is a necessary step for any truly professional Django project. While a bit scary at first they are conceptually quite straightforward. In the next chapter we'll fully configure our email settings and add password reset functionality.

Chapter 9: Email

In this chapter we will fully configure email and add password change and password reset functionality. Currently emails are not actually sent to users. They are simply outputted to our command line console. We'll change that by signing up for a third-party email service, obtaining API keys, and updating our `settings.py` file. Django takes care of the rest.

So far all of our work—custom user model, pages app, static assets, authentication with `django-allauth`, and environment variables—could apply to almost *any* new project. After this chapter we will start building out the `Bookstore` site itself as opposed to foundational steps.

Custom Confirmation Emails

Let's sign up for a new user account to review the current user registration flow. Then we'll customize it. Make sure you are logged out and then navigate to the Sign Up page. I've chosen to use `testuser3@email.com` and `testpass123` as the password.

The screenshot shows a web browser window with the following details:

- Address Bar:** Shows "Sign Up" and the URL "127.0.0.1:8000/accounts/signup/".
- Page Title:** "Bookstore".
- User Navigation:** "About", "Log In", and a "Sign Up" button.
- Form Fields:**
 - E-mail***: Input field containing "testuser3@email.com".
 - Password***: Input field containing "*****".
- Action Button:** A green "Sign Up" button.

testuser3 Sign Up

Upon submission we are redirected to the homepage with a custom greeting and an email is sent to us within the command line console. You can see this by checking the logs with `docker-compose logs`.

To customize this email we first need to find the existing templates. Navigate over to the [django-allauth source code on Github](#) and perform a search with a portion of the generated text. That leads to the discovery that there are in fact two files used: one for the subject line, `email_confirmation_subject.txt`, and one for the email body called `email_confirmation_message.txt`.

To update both we'll override them by recreating the same structure of `django-allauth` which means making our own `email` directory within `templates/account` and then adding our own versions of the files there.

Command Line

```
$ mkdir templates/account/email  
$ touch templates/account/email/email_confirmation_subject.txt  
$ touch templates/account/email/email_confirmation_message.txt
```

Let's start with the subject line since it's the shorter of the two. Here is the default text from django-allauth.

email_confirmation_subject.txt

```
{% load i18n %}  
{% autoescape off %}  
{% blocktrans %}Please Confirm Your E-mail Address{% endblocktrans %}  
{% endautoescape %}
```

The first line, `{% load i18n %}`, is to support Django's [internationalization](#) functionality, the ability to support multiple languages. Then comes the Django template tag for [autoescape](#). By default it is "on" and protects against security issues like cross site scripting. But since we can trust the content of the text here, it is turned off.

Finally we come to our text itself which is wrapped in [blocktrans](#) template tags to support translations. Let's change the text to demonstrate that we can.

email_confirmation_subject.txt

```
{% load i18n %}  
{% autoescape off %}  
{% blocktrans %}Confirm Your Sign Up{% endblocktrans %}  
{% endautoescape %}
```

Now turn to the email confirmation message itself. Here is [the current default](#):

email_confirmation_message.txt

```
{% load account %}{% user_display user as user_display %}{% load i18n %}  
{% autoescape off %}{% blocktrans with site_name=current_site.name\  
site_domain=current_site.domain %}Hello from {{ site_name }}!  

```

You're receiving this e-mail because user {{ user_display }} has given yours\
as an e-mail address to connect their account.

To confirm this is correct, go to {{ activate_url }}
{% endblocktrans %}{% endautoescape %}
{% blocktrans with site_name=current_site.name site_domain=current_site.\
domain %}Thank you from {{ site_name }}!
{{ site_domain }}{% endblocktrans %}

You probably noticed that the default email sent referred to our site as example.com
which is displayed here as {{ site_name }}. Where does that come from? The answer is
the sites section of the Django admin, which is used by django-allauth. So head to the
admin at <http://127.0.0.1:8000/admin/> and click on the Sites link on the homepage.

The screenshot shows the Django administration interface for managing sites. The title bar says "Select site to change | Django". The URL is "127.0.0.1:8000/admin/sites/site/". The main heading is "Django administration" with a welcome message "WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT". Below it, the breadcrumb navigation is "Home > Sites > Sites". A search bar and a "Search" button are present. A "Select site to change" dropdown menu is open, showing one item: "example.com" under "DOMAIN NAME" and "example.com" under "DISPLAY NAME". There is also a "ADD SITE" button. The status bar at the bottom indicates "1 site".

Admin Sites

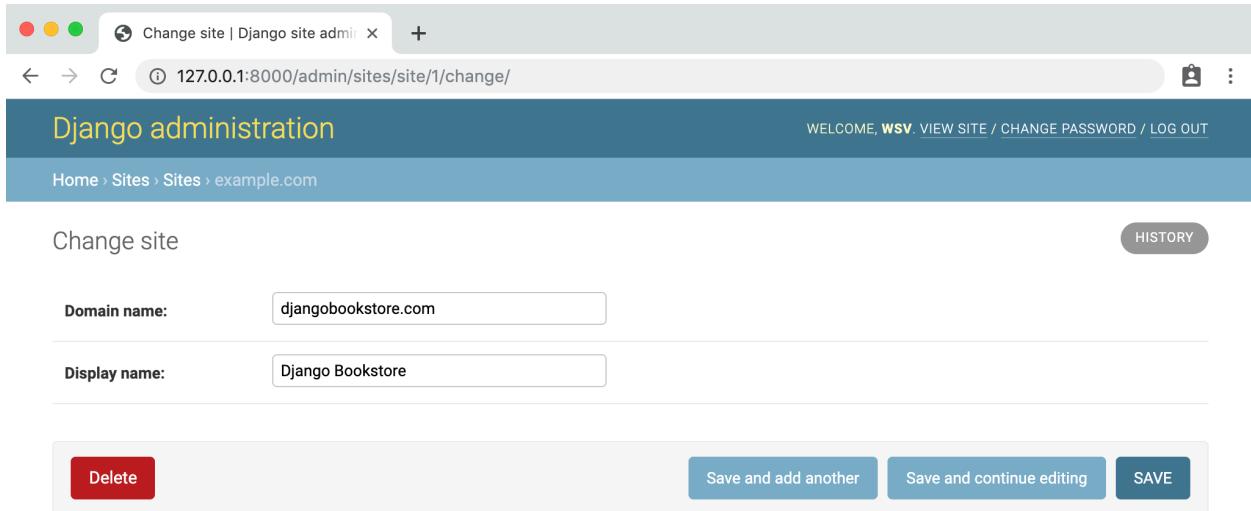
There is a “Domain Name” and a “Display Name” here. Click on `example.com` under “Domain Name” so we can edit it.

The screenshot shows the "Change site" page for the site "example.com". The title bar says "Change site | Django site admin". The URL is "127.0.0.1:8000/admin/sites/site/1/change/". The main heading is "Django administration" with a welcome message "WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT". Below it, the breadcrumb navigation is "Home > Sites > Sites > example.com". A "HISTORY" button is visible. The form fields show "Domain name: example.com" and "Display name: example.com". At the bottom, there are four buttons: "Delete", "Save and add another", "Save and continue editing", and a large "SAVE" button.

Admin Change Site

The **Domain Name** is the full domain name for a site, for example it might be `djangobookstore.com`, while the **Display Name** is a human-readable name for the site such as `Django Bookstore`.

Make these updates and click the “Save” button in the lower right corner when done.



The screenshot shows the Django administration interface for changing a site. The URL is 127.0.0.1:8000/admin/sites/site/1/change/. The page title is "Django administration". The top navigation bar includes "WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT". Below the title, it says "Home > Sites > Sites > example.com". The main content area is titled "Change site". It has two input fields: "Domain name" with "djangobookstore.com" and "Display name" with "Django Bookstore". At the bottom are four buttons: a red "Delete" button, and three blue "Save" buttons labeled "Save and add another", "Save and continue editing", and a larger "SAVE" button.

Admin Sites - DjangoBookstore.com

Ok, back to our email. Let's customize it a bit by changing the greeting from "Hello" to "Hi". Note that backslashes \ are included for formatting but are not necessary in the raw code. In other words, you can remove them from the code below—and other code examples—as needed.

email_confirmation_message.txt

```
{% load account %}{% user_display user as user_display %}{% load i18n %}\n\n{%- autoescape off %}{% blocktrans with site_name=current_site.name\nsite_domain=current_site.domain %}Hi from {{ site_name }}!
```

You're receiving this e-mail because user {{ user_display }} has given yours as an e-mail address to connect their account.

To confirm this is correct, go to {{ activate_url }}

```
{% endblocktrans %}{% endautoescape %}\n\n{%- blocktrans with site_name=current_site.name site_domain=current_site.\n.domain %}Thank you from {{ site_name }}!
```

```
{% site_domain %}{% endblocktrans %}
```

One final item to change. Did you notice the email was from `webmaster@localhost`? That's a default setting we can also update via `DEFAULT_FROM_EMAIL`. Let's do that now by adding the following line at the bottom of the `bookstore_project/settings.py` file.

Code

```
# bookstore_project/settings.py  
DEFAULT_FROM_EMAIL = 'admin@jangobookstore.com'
```

Make sure you are logged out of the site and go to the Sign Up page again to create a new user. I've used `testuser4@email.com` for convenience. After being redirected to the homepage check the command line to see the message by typing `docker-compose logs`.

Command Line

```
...  
web_1 | Content-Transfer-Encoding: 7bit  
web_1 | Subject: [Django Bookstore] Confirm Your Sign Up  
web_1 | From: admin@jangobookstore.com  
web_1 | To: testuser4@email.com  
web_1 | Date: Sat, 13 Jul 2019 18:34:50 -0000  
web_1 | Message-ID: <156312929025.27.2332096239397833769@87d045aff8f7>  
web_1 |  
web_1 | Hi from Django Bookstore!  
web_1 |  
web_1 | You're receiving this e-mail because user testuser4 has given yours\  
as an e-mail address to connect their account.  
web_1 |
```

```
web_1 | To confirm this is correct, go to http://127.0.0.1:8000/accounts/\
confirm-email/NA:1hmjKk:6MiDB5XoLW3HAhePuZ5WucR0Fiw/
web_1 |
web_1 | Thank you from Django Bookstore!
web_1 | djangobookstore.com
```

And there it is with the new `From` setting, the new message, and the new domain `djangobookstore.com` that sent the email.

Email Confirmation Page

Click on the unique URL link in the email which redirects to the email confirm page.

The screenshot shows a web browser window with the following details:

- Address Bar:** Confirm E-mail Address
- URL:** 127.0.0.1:8000/accounts/confirm-email/NA:1hmjKk:6MiDB5XoLW3HAhePuZ5WucR0Fiw/
- Content Area:**
 - Messages:**
 - You have signed out.
 - Confirmation e-mail sent to testuser4@email.com.
 - Successfully signed in as testuser4.
 - Menu:**
 - Change E-mail
 - Sign Out

Confirm E-mail Address

Please confirm that testuser4@email.com is an e-mail address for user testuser4.

[Confirm](#)

Confirm Email Page

Not very attractive. Let's update it to match the look of the rest of our site. Searching again in the [django-allauth source code on Github](#) reveals the name and location of this file is `templates/account/email_confirm.html`. So let's create our own template.

Command Line

```
$ touch templates/account/email_confirm.html
```

And then update it to extend `_base.html` and use Bootstrap for the button.

Code

```
<!-- templates/account/email_confirm.html -->

{% extends '_base.html' %}

{% load i18n %}
{% load account %}

{% block head_title %}{% trans "Confirm E-mail Address" %}{% endblock %}

{% block content %}

<h1>{% trans "Confirm E-mail Address" %}</h1>

{% if confirmation %}

{% user_display confirmation.email_address.user as user_display %}

<p>{% blocktrans with confirmation.email_address.email as email %}Please confirm  
that <a href="mailto:{{ email }}">{{ email }}</a> is an e-mail address for user  
{{ user_display }}.{% endblocktrans %}</p>

<form method="post" action="{% url 'account_confirm_email' confirmation.key %}">
{% csrf_token %}

<button class="btn btn-primary" type="submit">{% trans 'Confirm' %}</button>
```

```
</form>

{%
  else %}

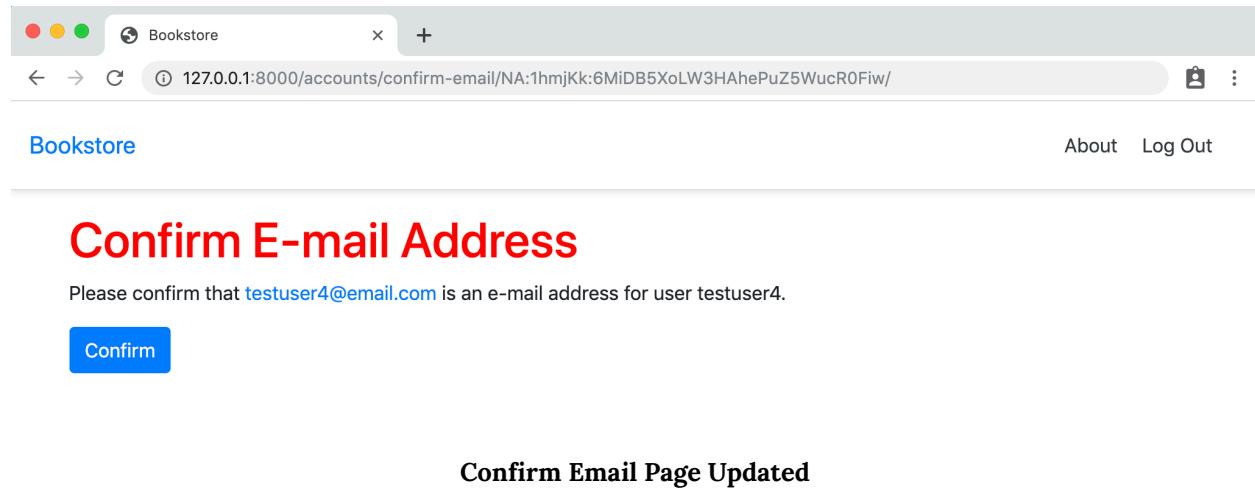
  {% url 'account_email' as email_url %}

  <p>{% blocktrans %}This e-mail confirmation link expired or is invalid. Please
  <a href="{{ email_url }}>issue a new e-mail confirmation request</a>.
  {% endblocktrans %}</p>

{%
  endif %}

{%
  endblock %}
```

Refresh the page to see our update.



The screenshot shows a web browser window with the following details:

- Header:** Bookstore
- Address Bar:** 127.0.0.1:8000/accounts/confirm-email/NA:1hmjKk:6MiDB5XoLW3HAhePuZ5WucR0Fiw/
- Page Content:**
 - Title:** Bookstore
 - Links:** About Log Out
 - Section:** Confirm E-mail Address
 - Text:** Please confirm that [testuser4@email.com](#) is an e-mail address for user testuser4.
 - Button:** Confirm
- Message:** Confirm Email Page Updated

Password Reset and Password Change

Django and `django-allauth` also come with support for additional user account features such as the ability to reset a forgotten password and change your existing password if already logged in.

The locations of the default password reset and password change pages are as follows:

- <http://127.0.0.1:8000/accounts/password/reset/>
- <http://127.0.0.1:8000/accounts/password/change/>

If you go through the flow of each you can find the corresponding templates and email messages in the `django-allauth` source code.

Email Service

The emails we have configured so far are generally referred to as “Transactional Emails” as they occur based on a user action of some kind. This is in contrast to “Marketing Emails” such as, say, a monthly newsletter.

There are many transactional email providers available including SendGrid, MailGun, Amazon’s Simple Email Service. Django is agnostic about which provider is used; the steps are similar for all and many have a free tier available.

After signing up for an account with your email service of choice you’ll often have a choice between using `SMTP` or a Web API. `SMTP` is easier to configure, but a web API is more configurable and robust. Start with `SMTP` and work your way from there: email configurations can be quite complex in their own right.

After obtaining a username and password with an email provider, a few settings tweaks will allow Django to use them to send emails.

The first step would be to update the `EMAIL_BACKEND` config which should be near the bottom of the `bookstore_project/settings.py` file since we previously updated it.

Code

```
# bookstore_project/settings.py  
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend' # new
```

And then to configure `EMAIL_HOST`, `EMAIL_HOST_USER`, `EMAIL_HOST_PASSWORD`, `EMAIL_PORT`, and `EMAIL_USE_TLS` based on the instructions from your email provider as environment variables.

In the official source code the `EMAIL_BACKEND` will remain `console`, but the previous steps are how to add an email service. If you find yourself frustrated properly configuring email, well, you're not alone! Django does at least make it far, far easier than implementing without the benefits of a batteries-included framework.

Git

To commit this chapter's code updates make sure to check the status of changes, add them all, and include a commit message.

Command Line

```
$ git status  
$ git add .  
$ git commit -m 'ch9'
```

If you have any issues compare your code against the [official source code on Github](#).

Conclusion

Configuring email properly is largely a one-time pain. But it is a necessary part of any production website. Now that SendGrid is working additional email functionality can be added as needed and it will “just work.”

This concludes the foundational chapters for our Bookstore project. In the next chapter we’ll finally start building out the Bookstore itself.

Chapter 10: Books App

In this chapter we will build a Books app for our project that displays all available books and has an individual page for each. We'll also explore different URL approaches starting with using an `id`, then switching to a slug, and finally using a UUID.

To start we must create this new app which we'll call `books`.

Command Line

```
$ docker-compose exec web python manage.py startapp books
```

And to ensure Django knows about our new app, open your text editor and add the new app to `INSTALLED_APPS` in our `settings.py` file:

Code

```
# bookstore_project/settings.py

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.sites',

    # Third-party
    'allauth',
    'allauth.account',
```



```
def __str__(self):  
    return self.title
```

At the top we're importing the Django class `models` and then creating a `Book` model that subclasses it which means we automatically have access to everything within `django.db.models.Models` and can add additional fields and methods as desired.

For `title` and `author` we're limiting the length to 200 characters and for `price` using a `DecimalField` which is a good choice when dealing with currency.

Below we've specified a `__str__` method to control how the object is outputted in the Admin and Django shell.

Now that our new database model is created we need to create a new migration record for it.

Command Line

```
$ docker-compose exec web python manage.py makemigrations books  
Migrations for 'books':  
    books/migrations/0001_initial.py  
        - Create model Book
```

And then apply the migration to our database.

Command Line

```
$ docker-compose exec web python manage.py migrate books
Operations to perform:
  Apply all migrations: account, admin, auth, books, contenttypes, sessions,
  sites, users
Running migrations:
  Applying books.0001_initial... OK
```

Adding the name of the app `books` to each command is optional but a good habit as it keeps both the migrations file and the `migrate` command focused on just that app. If we'd left the app name off then all changes would be included in the migrations file and database migrate which can be harder to debug later on.

Our database is configured. Let's add some data to the admin.

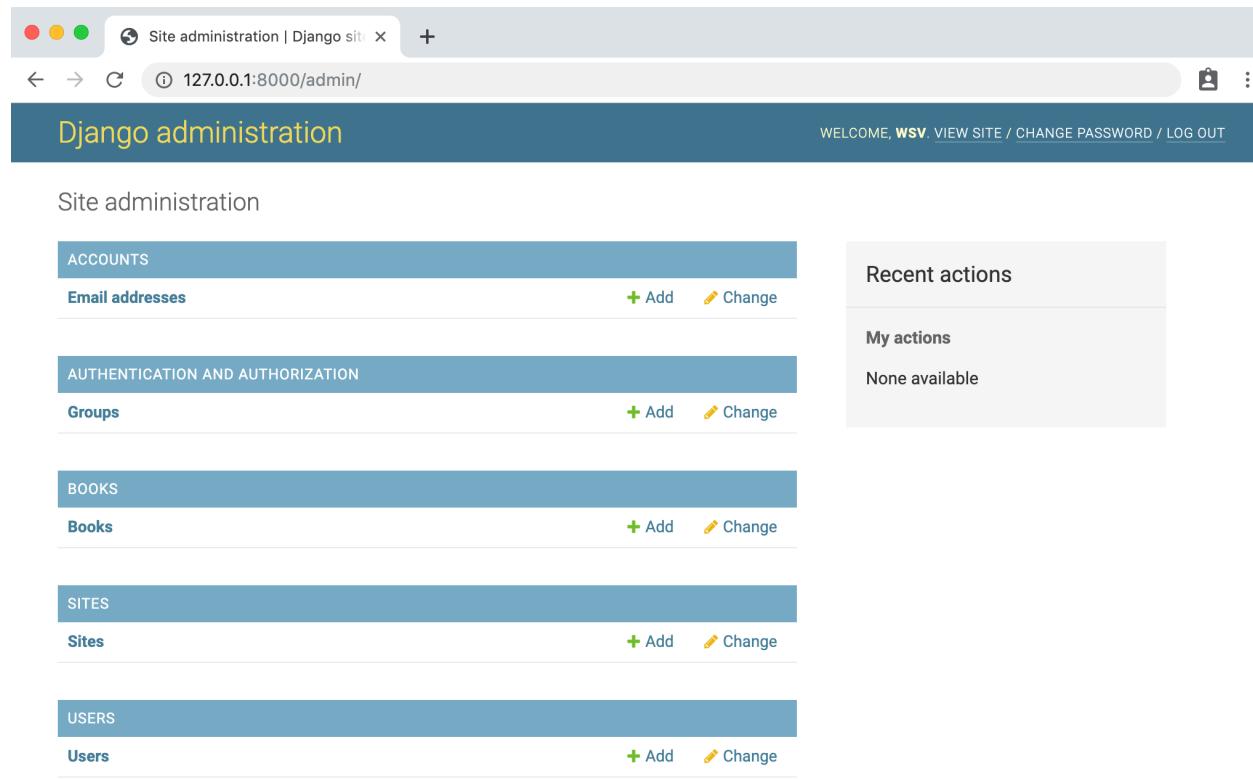
Admin

We need a way to access our data for which the Django admin is perfectly suited. Don't forget to `books/admin.py` or else the app won't appear! I forget this step almost every time even after using Django for years.

Code

```
# books/admin.py  
from django.contrib import admin  
from .models import Book  
  
admin.site.register(Book)
```

If you look into the admin at <http://127.0.0.1:8000/admin/> the Books app is now there.



The screenshot shows the Django administration interface at <http://127.0.0.1:8000/admin/>. The top navigation bar includes links for Site administration | Django site, a search bar, and user information (wsv). Below the header, the title "Django administration" is displayed, along with "WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT". The main content area is titled "Site administration" and lists several apps: ACCOUNTS, AUTHENTICATION AND AUTHORIZATION, BOOKS, SITES, and USERS. Under each app, there are links for "Email addresses", "Groups", "Books", "Sites", and "Users" respectively, each with "+ Add" and "Change" buttons. To the right, there are sections for "Recent actions" (empty) and "My actions" (empty).

Admin Homepage

Let's add a book entry for *Django for Professionals*. Click on the + Add button next to Books to create a new entry. The title is "Django for Professionals", the author is "William S. Vincent", and the price is \$39.00. There's no need to include the dollar sign \$ in the amount as we'll add that in our eventual template.

The screenshot shows the Django admin interface for adding a new book. The title is "Django for Professionals", the author is "William S. Vincent", and the price is "39.00". The "SAVE" button is highlighted.

Admin - Django for Professionals book

After clicking on the “Save” button we’re redirected to the main Books page which only shows the title.

The screenshot shows the Django admin interface for the Books list. A success message indicates that the book "Django for Professionals" was added successfully. The list shows one item: "Django for Professionals".

Admin Books Page

Let's update the `books/admin.py` file to specify which fields we also want displayed.

Code

```
# books/admin.py

from django.contrib import admin

from .models import Book


class BookAdmin(admin.ModelAdmin):

    list_display = ("title", "author", "price",)

admin.site.register(Book, BookAdmin)
```

Then refresh the page.

The screenshot shows the Django administration interface. The top bar includes the Django logo, a search field, and navigation links. The main title is 'Django administration'. The top right has a 'WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT' message. Below the title, the breadcrumb navigation shows 'Home > Books > Books'. A large heading 'Select book to change' is centered above a table. The table has columns: Action, TITLE, AUTHOR, and PRICE. One row is visible, showing a checkbox next to 'Django for Professionals', the author 'William S. Vincent', and the price '39.00'. At the bottom left of the table, it says '1 book'. On the right side of the table, there is a 'ADD BOOK +' button. The entire interface is styled with a light blue header and a white body.

Action:	TITLE	AUTHOR	PRICE
<input type="checkbox"/>	Django for Professionals	William S. Vincent	39.00

Admin Books List Page

Now that our database model is complete we need to create the necessary views, URLs, and templates so we can display the information on our web application. Where to start is always a question and a confusing one at that for developers.

Personally I often start with the URLs, then the Views, and the Templates.

URLs

We need to update two `urls.py` files. The first is `bookstore_project/urls.py` to notify it of the proper path for our new books app.

Code

```
# bookstore_project/urls.py

from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    # Django admin
    path('admin/', admin.site.urls),

    # User management
    path('accounts/', include('allauth.urls')),

    # Local apps
    path('', include('pages.urls')),
    path('books/', include('books.urls')), # new
]
```

Now create our `books` app URLs paths. We must create this file first.

Command Line

```
$ touch books/urls.py
```

We'll use empty string '' so combined with the fact that all `books` app URLs will start at `books/` that will also be the route for our main list view of each book. The view it references, `BookListView`, has yet to be created.

Code

```
# books/urls.py

from django.urls import path

from .views import BookListView

urlpatterns = [
    path('', BookListView.as_view(), name='book_list'),
]
```

Views

Moving on time for that `BookListView` we just referenced in our URLs file. This will rely on the built-in `ListView`, a Generic Class-Based View provided for common use cases like this. All we must do is specify the proper model and template to be used.

Code

```
# books/views.py

from django.views.generic import ListView

from .models import Book

class BookListView(ListView):
    model = Book
    template_name = 'books/book_list.html'
```

Note the template `book_list.html` does not exist yet.

Templates

It is optional to create an app specific folder within templates but it can help especially as number grows in size so we'll create one called `books`.

Command Line

```
$ mkdir templates/books/  
$ touch templates/books/book_list.html
```

Code

```
<!-- templates/books/book_list.html -->  
  
{% extends '_base.html' %}  
  
{% block title %}Books{% endblock title %}  
  
{% block content %}  
  {% for book in object_list %}  
    <div>  
      <h2><a href="">{{ book.title }}</a></h2>  
    </div>  
  {% endfor %}  
{% endblock content %}
```

At the top we note that this template extends `_base.html` and then wraps our desired code with `content` blocks. We use the Django Templating Language to set up a simple `for` loop for each book. Note that `object_list` comes from `ListView` and contains all the objects in our view.

The final step is to spin up and then down our containers to reload the Django `settings.py` file. Otherwise it won't realize we've made a change and so there will

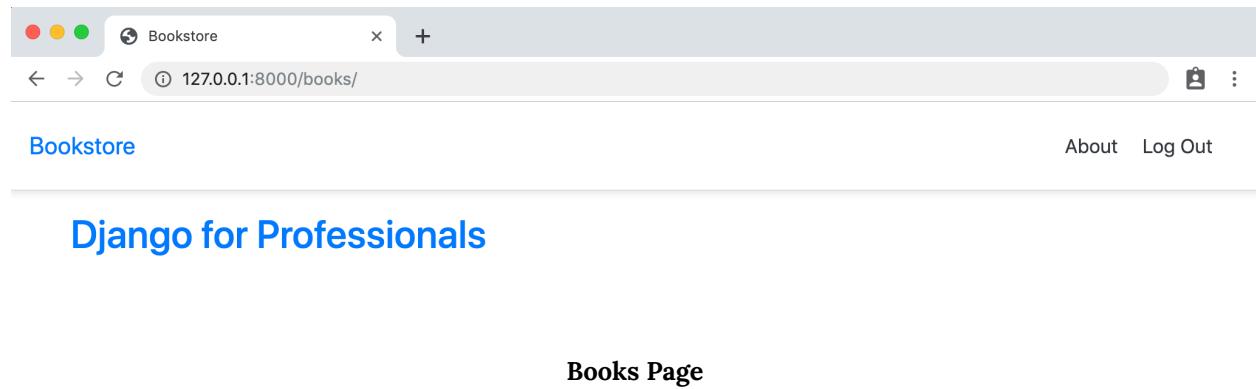
be an error page and in the logs a message about “ModuleNotFoundError: No module named ‘books.urls’”.

Spin down and then up again our containers.

Command Line

```
$ docker-compose down  
$ docker-compose up -d
```

If you go to <http://127.0.0.1:8000/books/> now the books page will work.



object_list

ListView relies on `object_list`, as we just saw, but this is far from descriptive. A better approach is to rename it to a [friendlier](#) name using `context_object_name`.

Update `books/views.py` as follows.

Code

```
# books/views.py

from django.views.generic import ListView, DetailView

from .models import Book


class BookListView(ListView):
    model = Book
    context_object_name = 'book_list' # new
    template_name = 'books/book_list.html'
```

And then swap out `object_list` in our template for `book_list`.

Code

```
<!-- templates/books/book_list.html -->

{% extends '_base.html' %}

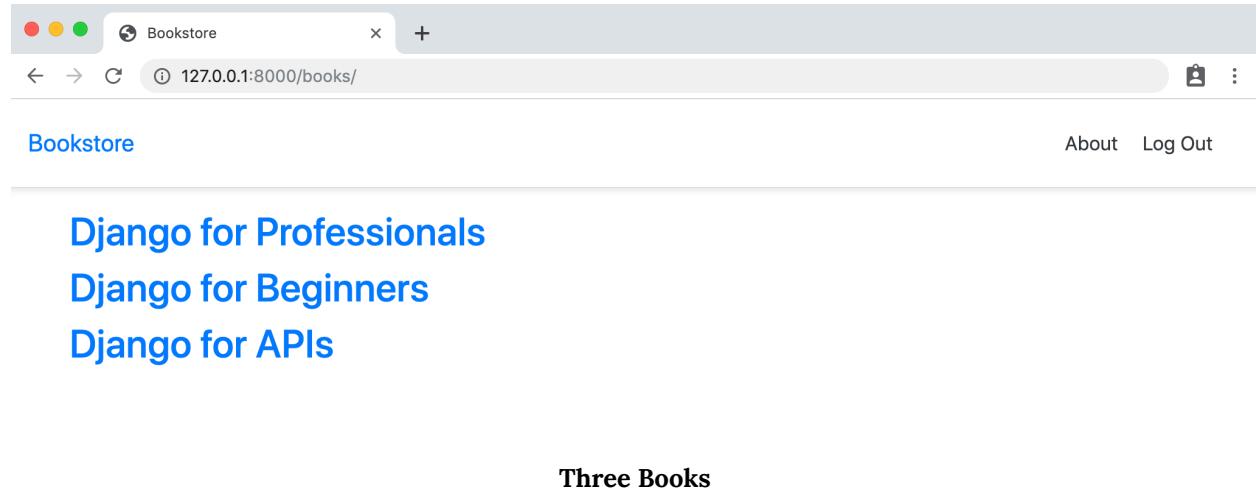
{% block title %}Books{% endblock title %}

{% block content %}
    {% for book in book_list %}
        <div>
            <h2><a href="">{{ book.title }}</a></h2>
        </div>
    {% endfor %}
{% endblock content %}
```

Refresh the page and it will still work as before! This technique is especially helpful

on larger projects where multiple developers are working on a project. It's hard for a front-end engineer to guess correctly what `object_list` means!

To prove the list view works for multiple items add two more books to the site via the admin. I've added my two other Django books—*Django for Beginners* and *Django for APIs*—which both have “William S. Vincent” as the author and “39.00” as the price.



Individual Book Page

Now we can add individual pages for each book by using another Generic Class-Based View called [DetailView](#).

Our process is similar to the Books page and starts with the URL importing `BookDetailView` on the second line and then setting the path to be the primary key of each book which will be represented as an integer `<int:pk>`.

Code

```
# books/urls.py

from django.urls import path

from .views import BookListView, BookDetailView # new

urlpatterns = [
    path('', BookListView.as_view(), name='book_list'),
    path('<int:pk>', BookDetailView.as_view(), name='book_detail'), # new
]
```

Django automatically adds an [auto-incrementing primary key](#) to our database models. So while we only declared the fields `title`, `author`, and `body` on our `Post` model, under-the-hood Django also added another field called `id`, which is our primary key. We can access it as either `id` or `pk`.

The `pk` for our first book is 1. For the second one it will 2. And so on. Therefore when we go to the individual entry page for our first post, we can expect that its URL route will be `books/1`.

Now on to the `books/views.py` file where we'll import `DetailView` and create a `BookDetailView` class that also specifies `model` and `template_name` fields.

Code

```
# books/views.py

from django.views.generic import ListView, DetailView # new

from .models import Book


class BookListView(ListView):
    model = Book
    context_object_name = 'book_list'
    template_name = 'books/book_list.html'

class BookDetailView(DetailView): # new
    model = Book
    template_name = 'books/book_detail.html'
```

And finally the template which we must first create.

Command Line

```
$ touch templates/books/book_detail.html
```

Then have it display all the current fields. We can also showcase the title in the `title` tags so that it appears in the web browser tab.

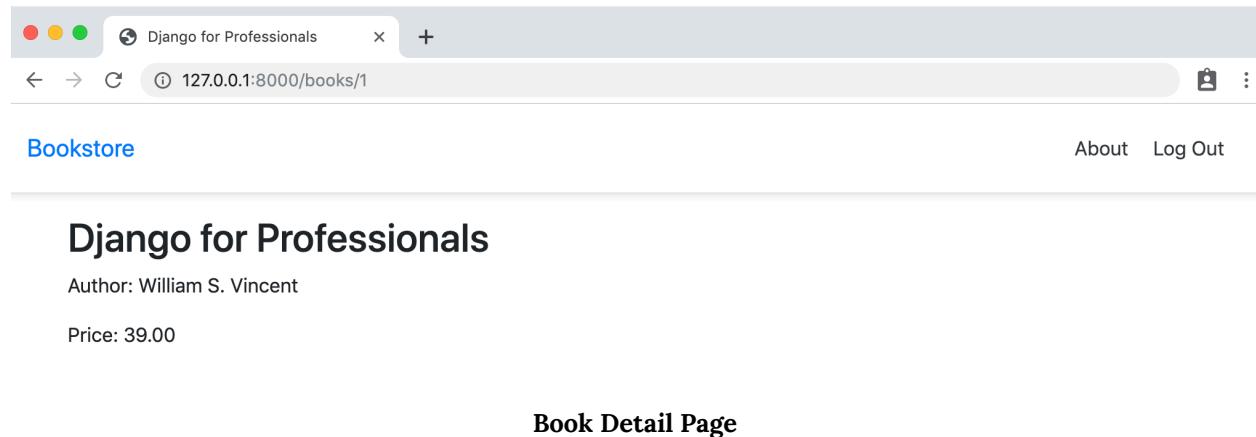
Code

```
<!-- templates/books/book_detail.html -->  
{% extends '_base.html' %}  
  
{% block title %}{{ book.title }}{% endblock title %}  
  
{% block content %}  


## <{{ object.title }}></a></h2> Author: {{ object.author }}</p> Price: {{ object.price }}</p> />

  
{% endblock content %}
```

If you navigate now to <http://127.0.0.1:8000/books/1/> you'll see a dedicated page for our first book post.



The screenshot shows a web browser window with the following details:

- Address Bar:** Shows the URL 127.0.0.1:8000/books/1/.
- Header:** The page is titled "Django for Professionals".
- Content:**
 - The book title is "Django for Professionals".
 - The author is "William S. Vincent".
 - The price is "39.00".
- Navigation:** Includes links for "About" and "Log Out".

context_object_name

Just as `ListView` defaults to `object_list` which we updated to be more specific, so too `DetailView` defaults to `object` which we can make more descriptive using `context_object_name`. We'll set it to `book`.

Code

```
# books/views.py

...
class BookDetailView(DetailView):
    model = Book
    context_object_name = 'book' # new
    template_name = 'books/book_detail.html'
```

Don't forget to update our template too with this change, swapping out `object` for `book` for our three fields.

Code

```
<!-- templates/books/book_detail.html -->
{% extends '_base.html' %}

{% block title %}{{ book.title }}{% endblock title %}

{% block content %}
<div class="book-detail">
    <h2><a href="">{{ book.title }}</a></h2>
    <p>Author: {{ book.author }}</p>
    <p>Price: {{ book.price }}</p>
</div>
{% endblock content %}
```

As a final step update the URL link on the book list page to point to individual page. With the `url template tag` we can point to `book_detail` – the URL name set in `books/urls.py` – and then pass in the `pk`.

Code

```
<!-- templates/books/book_list.html -->

{% extends '_base.html' %}

{% block title %}Books{% endblock title %}

{% block content %}
    {% for book in book_list %}
        <div>
            <h2><a href="{% url 'book_detail' book.pk %}">{{ book.title }}</a></h2>
        </div>
    {% endfor %}
{% endblock content %}
```

Refresh the book list page at <http://127.0.0.1:8000/books/> and links are now all clickable and direct to the correct individual book page.

get_absolute_url

One additional step we haven't made yet, but should is to add a `get_absolute_url()` method which sets a canonical URL for the model. It is also required when using the `reverse()` function which is commonly used.

Here's how to add it to our `books/models.py` file. Import `reverse` at the top. Then add the `get_absolute_url` method which will be the reverse of our URL name, `book_detail`, and passes in the `id` as a string.

Code

```
# books/models.py

from django.db import models
from django.urls import reverse # new

class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.CharField(max_length=200)
    price = models.DecimalField(max_digits=6, decimal_places=2)

    def __str__(self):
        return self.title

    def get_absolute_url(self): # new
        return reverse('book_detail', args=[str(self.id)])
```

Then we can update the templates. Currently our a `href` link is using `{% url 'book_detail' book.pk %}`. However we can instead use `get_absolute_url` directly which already has the `pk` passed in.

Code

```
<!-- templates/books/book_list.html -->
{% extends '_base.html' %}

{% block title %}Books{% endblock title %}

{% block content %}
    {% for book in book_list %}
        <div>
            <h2><a href="{{ book.get_absolute_url }}">{{ book.title }}</a></h2>
        </div>
    {% endfor %}
{% endblock content %}
```

There's no need to use the `url` template tag either, just one canonical reference that can be changed, if needed, in the `books/models.py` file and will propagate throughout the project from there. This is a cleaner approach and should be used whenever you need individual pages for an object.

Primary Keys vs. IDs

It can be confusing whether to use a primary key (PK) or an ID in a project, especially since Django's `DetailView` treats them interchangeably. However there is a subtle difference.

The `id` is a model field automatically set by Django internally to auto-increment. So the first book has an `id` of 1, the second entry of 2, and so on. This is also, by default, treated as the primary key `pk` of a model.

However it's possible to manually change what the primary key is for a model. It doesn't have to be `id`, but could be something like `object_id` depending on the use case. Additionally Python has a built-in `id()` object which can sometimes cause confusion and/or bugs.

By contrast the primary key `pk` refers to the primary key field of a model so you're safer using `pk` when in doubt. And in fact in the next section we will update the `id` of our model!

Slugs vs. UUIDs

Using the `pk` field in the URL of our `DetailView` is quick and easy, but not ideal for a real-world project. The `pk` is currently the same as our auto-incrementing `id`. Among other concerns, it tells a potential hacker exactly how many records you have in your database; it tells them exactly what the `id` is which can be used in a potential attack; and there can be synchronization issues if you have multiple front-ends.

There are two alternative approaches. The first is called a "slug", a newspaper term for a short label for something that is often used in URLs. For example, in our example of "Django for Professionals" its slug could be `django-for-professionals`. There's even a `SlugField` model field that can be used and either added when creating the `title` field by hand or auto-populated upon save. The main challenge with slugs is handling duplicates, though this can be solved by adding random strings or numbers to a given slug field. The synchronization issue remains though.

A better approach is to use a [UUID \(Universally Unique Identifier\)](#) which Django now supports via a dedicated `UUIDField`.

Let's implement a UUID now by adding a new field to our model and then updating the URL path.

Import `uuid` at the top and then update the `id` field to actually be a `UUIDField` that

is now the primary key. We also use `uuid4` for the encryption. This allows us to use `DetailView` which requires either a `slug` or `pk` field; it won't work with a `UUID` field without significant modification.

Code

```
# books/models.py

import uuid # new
from django.db import models
from django.urls import reverse

class Book(models.Model):
    id = models.UUIDField( # new
        primary_key=True,
        default=uuid.uuid4,
        editable=False)
    title = models.CharField(max_length=200)
    author = models.CharField(max_length=200)
    price = models.DecimalField(max_digits=6, decimal_places=2)

    def __str__(self):
        return self.title

    def get_absolute_url(self):
        return reverse('book_detail', args=[str(self.id)])
```

In the URL path swap out the `id` for `uuid` in the detail view.

Code

```
# books/urls.py

from django.urls import path

from .views import BookListView, BookDetailView

urlpatterns = [
    path('', BookListView.as_view(), name='book_list'),
    path('<uuid:pk>', BookDetailView.as_view(), name='book_detail'), # new
]
```

But now we are faced with a problem: there are existing book entries, three in fact, with their own `id`s as well as related migration files that use them. Creating a new migration like this [causes real problems](#). The simplest approach, which we will use, is the most destructive: to simply delete old `books` migrations and start over.

Command Line

```
$ docker-compose exec web rm -r books/migrations
$ docker-compose down
```

One last issue is that we are also persisting our PostgreSQL database via a volume mount that still has records to the older `id` fields. You can see this with the `docker volume ls` command.

Command Line

```
$ docker volume ls
```

DRIVER	VOLUME NAME
local	books_postgres_data

The simplest approach is again to simply delete the volume and start over with Docker. As we're early enough in the project we'll take this route; a more mature project would require considering a more complex approach.

The steps involve starting up our `web` and `db` containers; adding a new initial migration file for the `books` app, applying all updates with `migrate`, and then creating a superuser account again.

Command Line

```
$ docker volume rm books_postgres_data
$ docker-compose up -d
$ docker-compose exec web python manage.py makemigrations books
$ docker-compose exec web python manage.py migrate
$ docker-compose exec web python manage.py createsuperuser
```

Now go into admin and add the three books again. If you then navigate to the main books page and click on an individual book you'll be taken to a new detail page with a UUID in the URL.

The screenshot shows a web browser window with the title bar "Django for Professionals". The address bar displays the URL "127.0.0.1:8000/books/3a2d7e5b-3b27-4675-bcdb-52ecafa86226". The page content includes a header "Bookstore" and "About Log Out". Below the header, the book title "Django for Professionals" is displayed in blue, followed by author information "Author: William S. Vincent" and price "Price: 39.00". A large bold text "Django for Professionals book UUID" is centered on the page.

Navbar

Let's add a link to the `books` page in our navbar. We can use the `url` template tag and the URL name of the page which is `book_list`.

Code

```
<!-- templates/_base.html -->  
<nav class="my-2 my-md-0 mr-md-3">  
  <a class="p-2 text-dark" href="{% url 'book_list' %}">Books</a>  
  <a class="p-2 text-dark" href="{% url 'about' %}">About</a>
```

The screenshot shows a web browser window with the title bar "Books". The address bar displays the URL "127.0.0.1:8000/books/". The page content includes a header "Bookstore" and "Books About Log Out". Below the header, the book titles "Django for Professionals", "Django for Beginners", and "Django for APIs" are listed in blue.

Updated NavBar

Tests

We need to test our model and views now. We want to ensure that the `Post` model works as expected, including its `str` representation. And we want to test both `ListView` and `DetailView`.

Here's what sample tests look like in the `books/tests.py` file.

Code

```
# books/tests.py

from django.test import Client, TestCase
from django.urls import reverse

from .models import Book

class BookTests(TestCase):

    def setUp(self):
        self.book = Book.objects.create(
            title='Harry Potter',
            author='JK Rowling',
            price='25.00',
        )

    def test_book_listing(self):
        self.assertEqual(f'{self.book.title}', 'Harry Potter')
        self.assertEqual(f'{self.book.author}', 'JK Rowling')
        self.assertEqual(f'{self.book.price}', '25.00')
```

```
def test_book_list_view(self):  
    response = self.client.get(reverse('book_list'))  
    self.assertEqual(response.status_code, 200)  
    self.assertContains(response, 'Harry Potter')  
    self.assertTemplateUsed(response, 'books/book_list.html')  
  
def test_book_detail_view(self):  
    response = self.client.get(self.book.get_absolute_url())  
    no_response = self.client.get('/books/12345/')  
    self.assertEqual(response.status_code, 200)  
    self.assertEqual(no_response.status_code, 404)  
    self.assertContains(response, 'Harry Potter')  
    self.assertTemplateUsed(response, 'books/book_detail.html')
```

There's a lot that's new in these tests so we'll walk through them slowly. At the top we import `get_user_model` to reference our active `User`. We import `TestCase` which we've seen before and also `Client()` which is new and used as a dummy Web browser for simulating GET and POST requests on a URL. In other words, whenever you're testing views you should use `client()`.

In our `setUp` method we add a sample book post to test and then confirm that both its string representation and content are correct. Then we use `test_post_list_view` to confirm that our homepage returns a 200 HTTP status code, contains our body text, and uses the correct `home.html` template. Finally `test_post_detail_view` tests that our detail page works as expected and that an incorrect page returns a 404. It's always good to both test that something **does** exist and that something incorrect **doesn't** exist in your tests.

Go ahead and run these tests now. They should all pass.

Command Line

```
$ docker-compose exec web python manage.py test  
Creating test database for alias 'default'...  
System check identified no issues (0 silenced).  
.....  
-----  
Ran 17 tests in 0.369s
```

OK

```
Destroying test database for alias 'default'...
```

Git

We've done a lot of work in this chapter so add it all to version control now with Git by adding new files and adding a commit message.

Command Line

```
$ git status  
$ git add .  
$ git commit -m 'ch10'
```

The official source code for this chapter is [available on Github](#) for reference.

Conclusion

We're at the end of quite a long chapter, but the architecture of our Bookstore project is now much clearer. We've added a `books` model, learned how to change the URL structure, and switched to the much more secure UUID pattern.

In the next chapter we'll learn about foreign key relationships and add a reviews option to our project.

Chapter 11: Reviews App

In this chapter we'll add a `reviews` app so that readers can leave reviews of their favorite books. It gives us a chance to discuss foreign keys, app structure, and dive into forms.

Foreign Keys

We've already used a foreign key with our user model, but didn't have to think about it. Now we do! Fundamentally a database table can be thought of as similar to a spreadsheet with rows and columns. There needs to be a *primary key* field that is unique and refers to each record. In the last chapter we changed that from `id` to a `UUID`, but one still exists!

This matters when we want to link two tables together. For example, our `Books` model will link to a `Reviews` model since each review has to be connected to a relevant book. This implies a foreign key relationship.

There are three possible types of foreign key relationships:

- [One-to-one](#)
- [One-to-many](#)
- [Many-to-many](#)

A `one-to-one` relationship is the simplest kind. An example would be a table of people's names and a table of social security numbers. Each person has only **one** social security numbers and each social security number is linked to only **one** person.

In practice `one-to-one` relationships are rare. It's unusual for both sides of a relationship to only be matched to one counterpart. Some other examples though would be country-flag or person-passport.

A `one-to-many` relationship is far more common and is the [default foreign key](#) setting within Django. For example, `one` student can sign up for `many` classes. Or an employee has `one` job title, maybe "Software Engineer," but there can be `many` software engineers within a given company.

It's also possible to have a [ManyToManyField](#) relationship. Let's consider a list of books and a list of authors: each book could have more than one author and each author can write more than one book. That's a many-to-many relationship. Just as with the previous two examples you need a linked Foreign Key field to connect the two lists. Additional examples include doctors and patients (every doctor sees multiple patients and vice versa) or employees and tasks (each employee has multiple tasks while each task is worked on by multiple employees).

Database design is a fascinating, deep topic that is both an art and a science. As the number of tables grows in a project over time it is almost inevitable that a refactoring will need to occur to address issues around inefficiency, bloat, and outright errors. [Normalization](#) is the process of structuring a relational database though far beyond the scope of this book.

Reviews model

Coming back to our basic reviews app, the first consideration is what type of foreign key relationship will there be. If we are going to link a user to a review, then it is a straightforward `one-to-many` relationship. However it could also be possible to link books to reviews which would be `many-to-many`. The "correct" choice quickly becomes somewhat subjective and certainly dependent upon the particular needs of

the project.

In this project we'll treat the reviews app as a one-to-many between authors and reviews as it's the simpler approach.

Here again we face a choice around how to design our project. Do we add the `Reviews` model within our existing `books/models.py` file or create a dedicated `reviews` app that we then link to? Let's start by adding a `Reviews` model to the `books` app.

Code

```
# books/models.py

import uuid
from django.contrib.auth import get_user_model # new
from django.db import models
from django.urls import reverse

class Book(models.Model):
    id = models.UUIDField(
        primary_key=True,
        default=uuid.uuid4,
        editable=False)
    title = models.CharField(max_length=200)
    author = models.CharField(max_length=200)
    price = models.DecimalField(max_digits=6, decimal_places=2)

    def __str__(self):
        return self.title

    def get_absolute_url(self):
        return reverse('book_detail', kwargs={'pk': str(self.pk)})
```

```
class Review(models.Model): # new
    book = models.ForeignKey(
        Book,
        on_delete=models.CASCADE,
        related_name='reviews',
    )
    review = models.CharField(max_length=255)
    author = models.ForeignKey(
        get_user_model(),
        on_delete=models.CASCADE,
    )

    def __str__(self):
        return self.review
```

At the top under imports include `get_user_model` which is needed to review to our `CustomUser` model, then create a dedicated `Review` model. The `book` field is the one-to-many foreign key that links `Book` to `Review` and we're following the standard practice of naming it the same as the linked model. All many-to-one relationships now require we specify an `on_delete` option, too. The `review` field contains the actual content which perhaps could be a `TextField` depending on how much space you want to provide for review length! For now, we'll force reviews to be short at 255 characters or less. And then we'll also link to the `author` field to auto-populate the current user with the review.

For all many-to-one relationships such as a `ForeignKey` we must also specify an `on_delete` option. And we're using `get_user_model` to reference our custom user model.

Create a new migrations file for our changes and then run `migrate` to apply them.

Command Line

```
$ docker-compose exec web python manage.py makemigrations books  
Migrations for 'books':  
    books/migrations/0002_review.py  
        - Create model Review  
$ docker-compose exec web python manage.py migrate
```

Admin

For the reviews app to appear in the admin we need to update `books/admin.py` substantially by adding the `Review` model and specifying a display of [TabularInline](#).

Code

```
# books/admin.py  
  
from django.contrib import admin  
  
from .models import Book, Review  
  
  
class ReviewInline(admin.TabularInline):  
    model = Review  
  
  
class BookAdmin(admin.ModelAdmin):  
    inlines = [  
        ReviewInline,  
    ]  
    list_display = ("title", "author", "price",)  
  
  
admin.site.register(Book, BookAdmin)
```

Now navigate to the books section at <http://127.0.0.1:8000/admin/books/book/> and then click on any of the books to see the reviews visible on the individual book page.

The screenshot shows the Django admin interface for a book titled 'Django for Professionals'. The book details are as follows:

- Title: Django for Professionals
- Author: William S. Vincent
- Price: 39.00

Below the book details is a 'REVIEWS' section. It contains three review entries, each with a text input field and a 'DELETE?' button. There is also a link to '+ Add another Review'.

At the bottom of the page are three buttons: 'Delete' (red), 'Save and add another' (blue), 'Save and continue editing' (blue), and 'SAVE' (blue).

Django for Professionals Admin Reviews

We're limited to reviews by existing users at this point, although we have previously created a `testuser@email.com` that was deleted when we removed the database volume mount in the previous chapter. There are two options for adding this account: we could go to the main site and use the "Sign Up" link or we can add it directly from the admin. Let's do the latter. From the `Users` section on the Admin homepage click on the "+ Add" button. Add a new user called `testuser`.

Django administration

WELCOME, wsv. [VIEW SITE](#) / [CHANGE PASSWORD](#) / [LOG OUT](#)

Home > Users > Users > Add user

Add user

First, enter a username and password. Then, you'll be able to edit more user options.

Username: testuser
Required. 150 characters or fewer. Letters, digits and @./+/-/_ only.

Password: Your password can't be too similar to your other personal information.
Your password must contain at least 8 characters.
Your password can't be a commonly used password.
Your password can't be entirely numeric.

Password confirmation: Enter the same password as before, for verification.

[Save and add another](#) [Save and continue editing](#) **SAVE**

Admin testuser

Then on the next page add `testuser@email.com` as the email address. Scroll down to the bottom of the page and click the “Save” button.

The user "testuser" was added successfully. You may edit it again below.

Username: testuser
Required: 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password: algorithm: pbkdf2_sha256 iterations: 150000 salt: BtrLJl***** hash: VVRqKJ*****
Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using [this form](#).

Personal info

First name:

Last name:

Email address: testuser@email.com

Admin testuser

Ok, finally, we can add reviews to the “Django for Professionals” book using `testuser`. Navigate back to the Books section and click on the correct book. Write two reviews and as AUTHOR make sure to select `testuser`.

The screenshot shows the Django admin interface for a book named "Django for Professionals". The book has an author listed as "William S. Vincent" and a price of "39.00". In the "REVIEWS" section, there are two entries: "Great book!" and "I enjoyed it.", both attributed to the user "testuser". Each review has a small edit icon and a green plus sign icon. At the bottom of the screen, there are four buttons: a red "Delete" button, and three blue "Save" buttons labeled "Save and add another", "Save and continue editing", and "SAVE".

Add Two Reviews

Templates

With the `reviews` model set it's time to update our templates to display reviews on the individual page for each book. Add a basic "Reviews" section and then loop over all existing reviews. Since this is a foreign key relationship we follow it by using `book.reviews.all`. Then display the review field with `review.review` and the author with `review.author`.

Code

```
# templates/books/book_detail.html

{% extends '_base.html' %}

{% block title %}{{ book.title }}{% endblock title %}

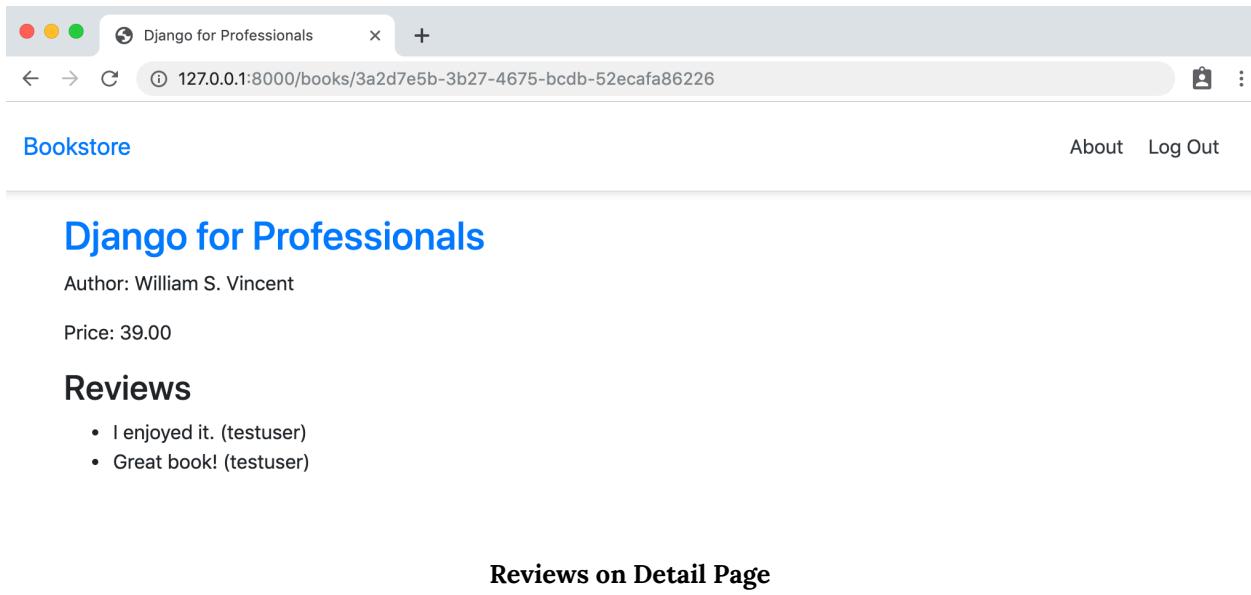
{% block content %}

<div class="book-detail">

    <h2><a href="">{{ book.title }}</a></h2>
    <p>Author: {{ book.author }}</p>
    <p>Price: {{ book.price }}</p>
    <div>
        <h3>Reviews</h3>
        <ul>
            {% for review in book.reviews.all %}
                <li>{{ review.review }} ({{ review.author }})</li>
            {% endfor %}
        </ul>
    </div>
</div>

{% endblock content %}
```

That's it! Navigate over to the "Django for Professionals" individual page to see the result. Your url will be different than the one here because we're using a UUID.



The screenshot shows a web browser window with the title "Django for Professionals". The URL in the address bar is "127.0.0.1:8000/books/3a2d7e5b-3b27-4675-bcde-52ecafa86226". The page content includes a header with "Bookstore" and navigation links for "About" and "Log Out". The main content area displays the book "Django for Professionals" by William S. Vincent, with a price of 39.00. Below the book details is a section titled "Reviews" containing two reviews from "testuser": "I enjoyed it." and "Great book!". At the bottom of the page, there is a link labeled "Reviews on Detail Page".

Tests

Time for tests. We need to create a new user for our review and add a review to the `setUp` method in our test suite. Then we can test that the book object contains the correct review.

This involves importing `get_user_model` as well as adding the `Review` model at the top. We can use `create_user` to make a new user called `reviewuser` and then a `review` object that is linked to our single `book` object. Finally under `test_book_detail_view` we can add an additional `assertContains` test to the `response` object.

Code

```
# books/tests.py

from django.contrib.auth import get_user_model # new
from django.test import Client, TestCase
from django.urls import reverse

from .models import Book, Review # new

class BookTests(TestCase):

    def setUp(self):
        self.user = get_user_model().objects.create_user( # new
            username='reviewuser',
            email='reviewuser@email.com',
            password='testpass123'
        )

        self.book = Book.objects.create(
            title='Harry Potter',
            author='JK Rowling',
            price='25.00',
        )

        self.review = Review.objects.create( # new
            book = self.book,
            author = self.user,
            review = 'An excellent review',
        )
```

```
def test_book_listing(self):
    self.assertEqual(f'{self.book.title}', 'Harry Potter')
    self.assertEqual(f'{self.book.author}', 'JK Rowling')
    self.assertEqual(f'{self.book.price}', '25.00')

def test_book_list_view(self):
    response = self.client.get(reverse('book_list'))
    self.assertEqual(response.status_code, 200)
    self.assertContains(response, 'Harry Potter')
    self.assertTemplateUsed(response, 'books/book_list.html')

def test_book_detail_view(self):
    response = self.client.get(self.book.get_absolute_url())
    no_response = self.client.get('/books/12345/')
    self.assertEqual(response.status_code, 200)
    self.assertEqual(no_response.status_code, 404)
    self.assertContains(response, 'Harry Potter')
    self.assertContains(response, 'An excellent review') # new
    self.assertTemplateUsed(response, 'books/book_detail.html')
```

If you run the tests now they all should pass.

Command Line

```
$ docker-compose exec web python manage.py test  
Creating test database for alias 'default'...  
System check identified no issues (0 silenced).  
.....  
-----  
Ran 17 tests in 0.675s
```

OK

```
Destroying test database for alias 'default'...
```

Git

Add our new code changes to Git and include a commit message for the chapter.

Command Line

```
$ git status  
....  
$ git add .  
$ git commit -m 'ch11'
```

The code for this chapter can be found [on the official Github repository](#).

Conclusion

With more time we might update the reviews' functionality with a form on the page itself, however this means AJAX calls using jQuery, React, Vue, or another dedicated

JavaScript framework. Unfortunately covering that fully is well beyond the scope of this book.

As the project grows it might also make sense to split reviews off into its own dedicated app. Doing so is a very subjective call. In general, keeping things as simple as possible—adding foreign keys within an existing app until it becomes too large to easily understand—is a solid approach.

In the next chapter we will add image uploads to our site so there can be covers for each book.

Chapter 12: File/Image Uploads

We previously configured static assets such as images in Chapter 6, but user-uploaded files, such as book covers, are somewhat different. To start with, Django refers to the former at `static` whereas anything uploaded by a user, whether it be a file or an image, is referred to as `media`.

The process for adding this feature for files or images is similar, but for images the Python image processing library [Pillow](#) must be installed which includes additional features such as basic validation.

Let's install `pillow` using our by-now-familiar pattern of installing it within Docker, stopping our containers, and forcing a build of the new image.

Command Line

```
$ docker-compose exec web pipenv install pillow==5.4.1
$ docker-compose down
$ docker-compose up -d --build
```

Media Files

Fundamentally the difference between static and media files is that we can trust the former, but we definitely can't trust the latter by default. There are always [security concerns](#) when dealing with [user-uploaded content](#). Notably, it's important to validate *all uploaded files* to ensure they are what they say they are. There are a number of nasty ways a malicious actor can attack a website that blindly accepts user uploads.

To start let's add two new configurations to the `bookstore_project/settings.py` file. By default `MEDIA_URL` and `MEDIA_ROOT` are empty and not displayed so we need to configure them:

- `MEDIA_ROOT` is the absolute file system path to the directory for user-uploaded files
- `MEDIA_URL` is the URL we can use in our templates for the files

For convenience lump the static and media file configurations together so add both of these settings after `STATICFILES_FINDERS` near the bottom of the file. We'll use the common convention of calling both `media`. Don't forget to include the trailing slash / for `MEDIA_URL`!

Code

```
# bookstore_project/settings.py

MEDIA_URL = '/media/' # new

MEDIA_ROOT = os.path.join(BASE_DIR, 'media') # new
```

Next add a new directory called `media` and a subdirectory called `covers` within it.

Command Line

```
$ mkdir media
$ mkdir media/covers
```

And finally since user-uploaded content is assumed to exist in a production context, to see media items locally we need to update `bookstore_project/urls.py` to show the files locally. This involves importing both `settings` and `static` at the top and then adding an additional line at the bottom.

Code

```
# bookstore_project/urls.py

from django.conf import settings # new
from django.conf.urls.static import static # new
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    # Django admin
    path('admin/', admin.site.urls),

    # User management
    path('accounts/', include('allauth.urls')),

    # Local apps
    path('', include('pages.urls')),
    path('books/', include('books.urls')),
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT) # new
```

Models

With our generic media configuration out of the way we can now turn to our models. To store these images we'll use Django's [ImageField](#) which comes with some basic image processing validation included.

The name of the field is `cover` and we specify the location of the uploaded image will be in `MEDIA_ROOT/covers` (the `MEDIA_ROOT` part is implied based on our earlier `settings.py` config).

Code

```
# bookstore_project/models.py

class Book(models.Model):

    id = models.UUIDField(
        primary_key=True,
        default=uuid.uuid4,
        editable=False)

    title = models.CharField(max_length=200)
    author = models.CharField(max_length=200)
    price = models.DecimalField(max_digits=6, decimal_places=2)
    cover = models.ImageField(upload_to='covers/') # new

    def __str__(self):
        return self.title

    def get_absolute_url(self):
        return reverse('book_detail', kwargs={'pk': str(self.pk)})
```

If we wanted to allow uploads of a regular file rather than an image file the only difference could be to change `ImageField` to `FileField`.

Since we've updated the model it's time to create a migrations file.

Command Line

```
$ docker-compose exec web python manage.py makemigrations books  
You are trying to add a non-nullable field 'cover_image' to book  
without a default; we can't do that (the database needs something to populate  
existing rows).
```

Please select a fix:

- 1) Provide a one-off default now (will be set on all existing rows with a
null value for this column)
- 2) Quit, and let me add a default in models.py

Select an option:

Oops! What happened? We're adding a new database field, but we already have three entries in our database for each book. Yet we failed to set a default value for `cover`.

To fix this type 2 to quit and we'll add a `blank` field set to `True` for existing images.

Code

```
# bookstore_project/models.py  
cover = models.ImageField(upload_to='covers/', blank=True) # new
```

It's common to see `blank` and `null` used together to set a default value on a field. A gotcha is that the field type – `ImageField` vs. `CharField` and so on – dictates how to use them properly so closely read the documentation for future use.

Now we can create a migrations file without errors.

Command Line

```
$ docker-compose exec web python manage.py makemigrations books
Migrations for 'books':
  books/migrations/0003_book_cover.py
    - Add field cover to book
```

And then apply the migration to our database.

Command Line

```
$ docker-compose exec web python manage.py migrate
Operations to perform:
  Apply all migrations: account, admin, auth, books, contenttypes, reviews, sessions, sites, users
Running migrations:
  Applying books.0003_book_cover... OK
```

Admin

We're in the home stretch now! Navigate over to the admin and to the entry for the book "Django for Professionals." The `cover` field is visible already and we already have a copy of it locally within `static/images/djangoforprofessionals.jpg` so use that file for the upload and then click the "Save" button in bottom right.

The screenshot shows the Django administration interface for a book. The URL in the browser is `127.0.0.1:8000/admin/books/book/46083b56-20b3-47ef-b64d-a82de9d578ee/change/`. The top navigation bar includes links for 'Home', 'Books', 'Books', and 'Django for Professionals'. On the right, there are 'HISTORY' and 'VIEW ON SITE' buttons.

The main form contains the following fields:

- Title:** Django for Professionals
- Author:** William S. Vincent
- Price:** 39.00
- Cover:** Choose File `djangoforprofessionals.jpg`

Below the form is a 'REVIEWS' section:

REVIEW	AUTHOR	DELETE?
Great book!	testuser	<input type="checkbox"/>
I enjoyed it.	testuser	<input type="checkbox"/>
(empty)	-----	<input type="checkbox"/>
(empty)	-----	<input type="checkbox"/>
(empty)	-----	<input type="checkbox"/>

At the bottom of the reviews section is a link: `+ Add another Review`.

At the very bottom of the page are three buttons: 'Delete' (red), 'Save and add another' (blue), 'Save and continue editing' (blue), and 'SAVE' (dark blue).

Admin add cover

This will redirect back to the main Books section. Click on the link again for “Django for Professionals” and we can see it currently exists in our desired location of `covers/`.

The screenshot shows the Django administration interface for a book entry. The URL in the browser is `127.0.0.1:8000/admin/books/book/46083b56-20b3-47ef-b64d-a82de9d578ee/change/`. The top navigation bar includes links for 'Home', 'Books', 'Books', and 'Django for Professionals'. On the right, there are 'HISTORY' and 'VIEW ON SITE' buttons.

Book Details:

- Title:** Django for Professionals
- Author:** William S. Vincent
- Price:** 39.00
- Cover:** Currently: `covers/djangoforprofessionals.jpg` (with a 'Clear' link) | Change: No file chosen

Reviews Section:

REVIEWS		AUTHOR	DELETE?
REVIEW	Great book!	testuser	<input type="checkbox"/>
I enjoyed it.	I enjoyed it.	testuser	<input type="checkbox"/>
		-----	<input type="checkbox"/>
		-----	<input type="checkbox"/>
		-----	<input type="checkbox"/>
+ Add another Review			

Action Buttons:

-
-
-
-

Admin with cover

Template

OK, final step. Let's update our template to display the book cover on the individual page. The route will be `book.cover.url` pointing to the location of the cover in our file system.

Here's what the updated `book_detail.html` file looks like with this one line change above the title.

Code

```
# templates/books/book_detail.html

{% extends '_base.html' %}

{% block title %}{{ book.title }}{% endblock title %}

{% block content %}

<div class="book-detail">

    <a href="">{{ book.title }}</a></h2>

    <p>Author: {{ book.author }}</p>

    <p>Price: {{ book.price }}</p>

    <div>

        <h3>Reviews</h3>

        <ul>

            {% for review in book.reviews.all %}

                <li>{{ review.review }} ({{ review.author }})</li>

            {% endfor %}

        </ul>

    </div>

</div>

{% endblock content %}
```

If you now visit the page for “Django for Professionals” you’ll see the cover image proudly there!

The screenshot shows a web browser window with the title bar "Django for Professionals". The address bar displays the URL "127.0.0.1:8000/books/46083b56-20b3-47ef-b64d-a82de9d578ee". The page content is a book detail page for "Django for Professionals" by William S. Vincent. The book cover is purple with white text. It features a yellow star rating of "2.2" at the top. The title "DJANGO" is in large white capital letters, with "for" in smaller white lowercase letters below it, and "PROFESSIONALS" in large white capital letters below that. Below the title, the subtitle "Production websites with Python & Django" is written in small white text. The author's name, "WILLIAM S. VINCENT", is on an orange horizontal bar at the bottom of the cover. Below the book cover, the title "Django for Professionals" is displayed in blue text. Underneath the title, the author's name "Author: William S. Vincent" and the price "Price: 39.00" are shown.

Reviews

- I enjoyed it. (testuser)
- Great book! (testuser)

Cover image

One potential gotcha is that our template now expects a `cover` to be present. If you navigate to either of the two other books, for which we have not added a cover, you'll see the following error message.

The 'cover' attribute has no file associated with it.

```
Request Method: GET
Request URL: http://127.0.0.1:8000/books/30ba68f5-a06a-4610-8db5-c38f65f88700
Django Version: 2.2.1
Exception Type: ValueError
Exception Value: The 'cover' attribute has no file associated with it.
Exception Location: /usr/local/lib/python3.7/site-packages/django/db/models/fields/files.py in _require_file, line 38
Python Executable: /usr/local/bin/python
Python Version: 3.7.3
Python Path: ['/code',
             '/usr/local/lib/python37.zip',
             '/usr/local/lib/python3.7',
             '/usr/local/lib/python3.7/lib-dynload',
             '/usr/local/lib/python3.7/site-packages']
Server time: Sat, 29 Jun 2019 18:05:41 +0000
```

Error during template rendering

In template /code/templates/_base.html, error at line 7

The 'cover' attribute has no file associated with it.

```
1 {% load static %}
2 <!DOCTYPE html>
```

Cover image error

We must add some basic logic to our template so that if a cover is not present the template doesn't look for it! This can be done using an `if` statement that checks for `book.cover` and displays it if it exists.

Code

```
# templates/books/book_detail.html

{% extends '_base.html' %}

{% block title %}{{ book.title }}{% endblock title %}

{% block content %}



{% if book.cover %}
        
    {% endif %}
    <p>Author: {{ book.author }}</p>


```

....

If you refresh either book page now you'll see they display the correct page albeit without a cover.

Next Steps

There are several additional steps that might be nice to take in a project, but are beyond the scope of this book. These include adding dedicated create/edit/delete forms for the creation of books and cover image. A quite lengthy list of extra validations can and should be placed on the image-uploading form to ensure that only a normal image is added to the database.

A further step would be to store `media` files in a dedicated CDN (Content Delivery Network) for additional security. This can also be helpful for performance on very large sites for `static` files, but for `media` files is a good idea regardless of the size.

Finally tests would be nice to have here although they would be primarily focused on the form validation section, not the basic image-uploading via the admin. Again this is an area that can become quite complex, but is worthy of further study.

Git

Make sure to create a new Git commit for the changes in this chapter.

Command Line

```
$ git status  
$ git add .  
$ git commit -m 'ch12'
```

As always you can compare your code against the [official source code on Github](#).

Conclusion

This chapter demonstrated how to add user files to a project. In practice it is straightforward, but the additional layer of security concerns makes it an area worthy of focus at scale.

In the next chapter we will add permissions to our site to lock it down.

Chapter 13: Permissions

Currently there are no permissions set on our Bookstore project. Any user, even one not logged in, can visit any page and perform any available action. While this is fine for prototyping, implementing a robust permissions structure is a must before deploying a website to production.

Django comes with [built-in authorization options](#) for locking down pages to either logged in users, specific groups, or users with the proper individual permission.

Logged-In Users Only

Confusingly there are multiple ways to add even the most basic permission: restricting access only to logged-in users. It can be done in a [raw way](#) using the `login_required()` decorator, or since we are using class-based views so far via the [LoginRequiredMixin](#).

Let's start by limiting access to the Books pages only to logged-in users. There is a link for it in the navbar so this is not the case of a user accidentally finding a URL (which also can happen); in this case the URL is quite public.

First import `LoginRequiredMixin` at the top and then add it before `ListView` since mixins are loaded from left-to-right. That way the first thing that is checked is whether the user is logged in; if they're not there's no need to load the `ListView`. The other part is setting a `login_url` for the user to be redirected to. This is the URL name for log in which, since we're using `django-allauth` is `account_login`. If we were using the traditional Django authentication system then this link would be called simply `login`.

The structure for BookDetailView is the same: add LoginRequiredMixin and a login_url route.

Code

```
# books/views.py

from django.contrib.auth.mixins import LoginRequiredMixin # new
from django.views.generic import ListView, DetailView

from .models import Book

class BookListView(LoginRequiredMixin, ListView): # new
    model = Book
    context_object_name = 'book_list'
    template_name = 'books/book_list.html'
    login_url = 'account_login' # new

class BookDetailView(LoginRequiredMixin, DetailView): # new
    model = Book
    context_object_name = 'book'
    template_name = 'books/book_detail.html'
    login_url = 'account_login' # new
```

And that's it! If you now log out of your account and click on the "Books" link it will automatically redirect you to the Log In page. However if you are logged in, the Books page loads normally.

Even if you somehow knew the UUID of a specific book page you'd be redirected to Log In as well!

Permissions

Django comes with a basic [permissions system](#) that is controlled through the Django admin. To demonstrate it we need to create a new user account. Navigate back to the Admin homepage and then click on “+ Add” next to **Users**.

We'll call this new user `special` and set a password of `testpass123`. Click on the “Save” button.

The screenshot shows the Django Admin 'Add user' page. The URL in the browser is `127.0.0.1:8000/admin/users/customuser/add/`. The page title is 'Django administration'. The 'Username' field contains 'special'. The 'Password' and 'Password confirmation' fields both contain masked input. Validation messages for the password are visible: 'Your password can't be too similar to your other personal information.', 'Your password must contain at least 8 characters.', 'Your password can't be a commonly used password.', and 'Your password can't be entirely numeric.' At the bottom, there are three buttons: 'Save and add another', 'Save and continue editing', and a larger 'SAVE' button.

Add User

The second page allows us to set an “Email address” to `special@email.com`. We’re using `django-allauth` so that our log in page requires only email and the sign up page also only uses email, but since we didn’t customize the admin as well it still expects a `username` when creating a new user this way.

The screenshot shows the Django admin interface for a custom user model named 'special'. The top navigation bar includes the title 'Change user | Django site admin', the URL '127.0.0.1:8000/admin/users/customuser/3/change/', and links for 'WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT'. The main content area is titled 'Change user' and shows the following fields:

- Username:** special (input field)
- Password:** algorithm: pbkdf2_sha256 iterations: 150000 salt: vz6DxA***** hash: xVt9j5***** (text field)
- Personal info:** Fields for First name, Last name, and Email address (special@email.com).

A note under the password field states: "Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using [this form](#)".

User Email

If we had wanted to fully rip out the default user system that would mean using `AbstractBaseUser` rather than `AbstractUser` back in Chapter 3 when we configured our custom user model.

Scrolling down further on the page to the bottom there are options to set Groups as well as User permissions. This is a long list of defaults Django provides.

The screenshot shows the Django admin interface for a user named 'CustomUser'. The top navigation bar indicates the current view is 'Change user | Django site adm...'. The URL in the address bar is '127.0.0.1:8000/admin/users/customuser/3/change/'. Below the header, there are two main sections for managing user permissions:

- User permissions:** This section lists "Available user permissions" on the left and "Chosen user permissions" on the right. The available permissions include various account-related actions like 'Can add email address', 'Can change email address', etc. The chosen permissions list is currently empty.
- Important dates:** This section displays the user's last login information. It shows the date as '2019-06-29' and time as '19:29:37'. A note below states: "Note: You are 4 hours behind server time."

At the bottom of the page, there are three buttons: 'Delete', 'Save and add another', 'Save and continue editing', and a large blue 'SAVE' button.

User Permissions

For now we won't use them since we'll create a custom permission in the next section so just click on the "Save" button in the lower right corner so that our email address is updated for the user account.

Custom Permissions

Setting [custom permissions](#) is a much more common occurrence in a Django project. We can set them via the `Meta` class on our database models.

For example, let's add a special status so that an author can read all books. In other words they have access to the `DetailView`. We could be much more specific with the permissions, restricting them per book, but this is a good first step.

In the `books/models.py` file we'll add a `Meta` class and set both the permission name and a description which will be visible in the admin.

Code

```
# books/models.py

...
class Book(models.Model):
    id = models.UUIDField(
        primary_key=True,
        default=uuid.uuid4,
        editable=False)
    title = models.CharField(max_length=200)
    author = models.CharField(max_length=200)
    price = models.DecimalField(max_digits=6, decimal_places=2)
    cover = models.ImageField(upload_to='covers/', blank=True)

class Meta: # new
    permissions = [
        ("special_status", "Can read all books"),
    ]
```

```
def __str__(self):
    return self.title

def get_absolute_url(self):
    return reverse('book_detail', args=[str(self.id)])
...
```

The order of the inner classes and methods here is deliberate. It follows the [Model style](#) section from the Django documentation.

Since we have updated our database model we must create a new migrations file and then apply it.

Command Line

```
$ docker-compose exec web python manage.py makemigrations books
$ docker-compose exec web python manage.py migrate
```

User Permissions

Now we need to apply this custom permission to our new special user. Thanks to the admin this is not a difficult task. Navigate to the `Users` section where the three existing users are listed: `special@email.com`, `testuser@email.com`, and `will@wsvincent.com` which is my superuser account.

The screenshot shows the Django administration interface for the 'CustomUser' model. The title bar reads 'Select user to change | Django'. The main content area is titled 'Django administration' and shows the URL '127.0.0.1:8000/admin/users/customuser/'. The top right has links for 'WELCOME, wsv', 'VIEW SITE / CHANGE PASSWORD / LOG OUT'. Below that is a breadcrumb trail: 'Home > Users > Users'. On the right, there's a 'ADD USER +' button. A search bar with a magnifying glass icon and a 'Search' button is at the top left. A 'FILTER' sidebar on the right includes sections for 'By staff status' (All, Yes, No), 'By superuser status' (All, Yes, No), and 'By active' (All, Yes, No). The main table lists three users:

<input type="checkbox"/>	EMAIL ADDRESS	USERNAME
<input type="checkbox"/>	special@email.com	special
<input type="checkbox"/>	testuser@email.com	testuser
<input type="checkbox"/>	will@wsvincent.com	wsv

Below the table, it says '3 users'.

Three Users

Click on the `special@email.com` user and then scroll down to `User permissions` near the bottom of the page. Within it search for `books | book | Can read all books` and select it by clicking on the \rightarrow arrow to add it to "Chosen user permissions." Don't forget to click the "Save" button at the bottom of the page.

The screenshot shows the Django admin interface for changing a user. The URL is `127.0.0.1:8000/admin/users/customuser/9/change/`. The top navigation bar shows "Change user | Django site adm".

User permissions:

Available user permissions (list view):

- account | email address | Can add email address
- account | email address | Can change email address
- account | email address | Can delete email address
- account | email address | Can view email address
- account | email confirmation | Can add email confirmation
- account | email confirmation | Can change email confirmation
- account | email confirmation | Can delete email confirmation
- account | email confirmation | Can view email confirmation
- admin | log entry | Can add log entry
- admin | log entry | Can change log entry
- admin | log entry | Can delete log entry
- admin | log entry | Can view log entry
- auth | group | Can add group

Chosen user permissions (list view):

- books | book | Can read all books

Important dates

Last login: Date: 2019-06-29 Today Time: 20:03:11 Now

Note: You are 4 hours behind server time.

Date joined: Date: 2019-06-29 Today Time: 18:54:09 Now

Note: You are 4 hours behind server time.

Action buttons: Delete, Save and add another, Save and continue editing, SAVE.

Add Permission

PermissionRequiredMixin

The last step is to apply the custom permission using the `PermissionRequiredMixin`. One of the many great features of class-based views is we can implement advanced functionality with very little code on our part and this particular mixin is a good

example of that.

Add `PermissionRequiredMixin` to our list of imports on the top line. Then add it to `DetailView` after `LoginRequiredMixin` but before `DetailView`. The order should make sense: if a user isn't already logged in it makes no sense to do the additional check of whether they have permission. Finally add a `permission_required` field which specifies the desired permission. In our case its name is `special_status` and it exists on the `books` model.

Code

```
# books/views.py

from django.contrib.auth.mixins import (
    LoginRequiredMixin,
    PermissionRequiredMixin # new
)
from django.views.generic import ListView, DetailView

from .models import Book

class BookListView(LoginRequiredMixin, ListView):
    model = Book
    context_object_name = 'book_list'
    template_name = 'books/book_list.html'
    login_url = 'account_login'

class BookDetailView(
    LoginRequiredMixin,
    PermissionRequiredMixin, # new
```

```
DetailView):  
    model = Book  
    context_object_name = 'book'  
    template_name = 'books/book_detail.html'  
    login_url = 'account_login'  
    permission_required = 'books.special_status' # new
```

Although we are not doing it here it is possible to add **multiple permissions** via the `permission_required` field.

To try out our work, log out of the admin. This is necessary because the superuser account is used for the admin and by default has access to everything. Not a great user account to test with!

Log in to the Bookstore site using the `testuser@email.com` account and then navigate to the Books page listing the three available titles. If you then click on any one of the books, you'll see a "403 Forbidden" error because permission was denied.



Now go back to the homepage at <http://127.0.0.1:8000/> and log out. Then log in using the `special@email.com` account. Navigate again to the Books page and each individual book page is accessible.

Groups & UserPassesTestMixin

The third permissions mixin available is [UserPassesTestMixin](#) which restricts a view's access only to users who pass a specific test.

And in large projects [Groups](#), which are Django's way of applying permissions to a category of users, become prominent. If you look on the Admin homepage there is a dedicated `Groups` section where they can be added and have permissions set. This is far more efficient than adding permissions on a per-user basis.

An example of groups is if you have a premium section on your website, a user upgrading could switch them into the premium group and then have access to however many specific extra permissions that involves.

Git

Make sure to create a new Git commit for the changes in this chapter.

Command Line

```
$ git status  
$ git add .  
$ git commit -m 'ch13'
```

As always you can compare your code again the [official source code on Github](#).

Conclusion

Permissions and groups are a highly subjective area that vary widely from project to project. However the basics remain the same and mimic what we've covered here.

The first pass is typically to restrict access to only logged in users, then add additional custom permissions from there around groups or users.

In the next chapter we'll build upon this by adding payments to our Bookstore site.

Chapter 14: Orders with Stripe

[Stripe](#) is one of the most popular payment providers and what we'll use in this book to power book order purchases. It has two main offerings from Stripe: [Checkout](#) which allows for the use of pre-built forms from Stripe and [Connect](#) which is used for a marketplace with multiple buyers and sellers. For example if we added book authors as users and wanted to process payments on their behalf, taking a commission for ourselves on the Bookstore website, then we would use Connect. But since we are just processing payments we will use Checkout.

Checkout itself is undergoing rapid iteration. There are now [two Checkout options](#) available to developers: a “Client Integration” where the payment form is hosted on Stripe itself and a “Server Integration” where we host the form ourselves. Since we’re using Django we’ll opt for the Server Integration approach.

The second major change is a new API that relies on [Sessions](#), however, as of the writing of this book, Sessions is not fully implemented and poorly documented. Therefore we will use the traditional Stripe approach which will be supported well into the future. Once you’ve understood how Stripe works under-the-hood making the switch in the future will be much easier.

It’s easy to become lost in all the complexity around payments, however, the important part for this book is understanding how payments are securely processed. That is what we’ll do here. By the end of this chapter we’ll have a working payments solution and the ability to further customize it as needed.

Payments Flow

Before we become lost in the implementation details, let's plan out how the payments flow should work. Currently there is a books page that lists all available books and then individual pages for each book. In the last chapter we saw how to add a permission for access to all books. Ultimately when an order is successfully completed, that user needs to have this permission flag flipped in the database. That's all we're doing here!

When a user is on the books page we'll include a link to a dedicated orders page which, upon success, we'll redirect back to the books page with all books now available. We can add in some template logic to replace "Order" buttons with "Read" buttons for the appropriate user.

Keep this high-level flow in mind as we go through the implementation process!

Orders app

We'll create a dedicated `orders` app and then configure it in the standard way: adding to `INSTALLED_APPS` configuration, updating `urls.py` files, creating views, and then templates.

Ready? Here we go. Start by creating a new `orders` app.

Command Line

```
$ docker-compose exec web python manage.py startapp orders
```

Then add it to the `INSTALLED_APPS` configuration in `bookstore_project/settings.py`.

Code

```
# bookstore_project/settings.py

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.sites',

    # Third-party
    'crispy_forms',
    'allauth',
    'allauth.account',

    # Local
    'users.apps.UsersConfig',
    'pages.apps.PagesConfig',
    'books.apps.BooksConfig',
    'orders.apps.OrdersConfig', # new
]
```

Update the top-level `bookstore_project/urls.py` file with `orders` routes which will live at `orders/`.

Code

```
# bookstore_project/urls.py

from django.conf import settings
from django.conf.urls.static import static
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    # Django admin
    path('admin/', admin.site.urls),

    # User management
    path('accounts/', include('allauth.urls')),

    # Local apps
    path('', include('pages.urls')),
    path('books/', include('books.urls')),
    path('orders/', include('orders.urls')), # new
] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

And then create a `orders/urls.py` file to create app-level URL routes.

Command Line

```
$ touch orders/urls.py
```

Since the top-level `urls.py` file is at `orders/` we can simply use the empty string '' for the main `orders` page. Provide a name of the path of `orders` and reference the view `OrdersPageView` which we'll create next.

Code

```
# orders/urls.py

from django.urls import path

from .views import OrdersPageView

urlpatterns = [
    path('', OrdersPageView.as_view(), name='orders'),
]
```

The view file will simply use `TemplateView` for now.

Code

```
# orders/views.py

from django.views.generic.base import TemplateView

class OrdersPageView(TemplateView):
    template_name = 'orders/purchase.html'
```

Finally we have the template which will live in a `templates/orders/` directory.

Command Line

```
$ mkdir templates/orders
$ touch templates/orders/purchase.html
```

Let's just use a placeholder heading of "Orders page" for now to test that it's working correctly.

Code

```
<!-- templates/orders/purchase.html -->  
{% extends '_base.html' %}  
  
{% block title %}Orders{% endblock title %}  
  
{% block content %}  
Orders page  
{% endblock content %}
```

The code is all done. But we must restart our containers so that the `settings.py` file update—adding `orders` to `INSTALLED_APPS`—is loaded into Django.

Command Line

```
$ docker-compose down  
$ docker-compose up -d
```

In your web browser visit <http://127.0.0.1:8000/orders/> to see our new orders page.



Orders Page

Stripe

We turn our attention to Stripe which needs to be installed locally. The Python library for Stripe is currently undergoing rapid iteration and is available on [Github](#).

Command Line

```
$ docker-compose exec web pipenv install stripe==2.32.0
$ docker-compose down
$ docker-compose up -d --build
```

Then go to the Stripe website and [register for a new account](#). Stripe regularly updates its new user onboarding flow, but as of the writing of this book the next page asks whether you want to use the Stripe API or an app. We want the API so select that option which redirects to the [dashboard](#) page.

Adding an account name is optional but recommended in the upper left corner. I've selected "DFP Book". Now click on the "Developers" link in the left sidebar.

The screenshot shows the Stripe Dashboard for a test account named "DFP Book". The left sidebar lists various sections: Home, Payments, Balance, Customers, Atlas, Radar, Billing, Connect, Orders, Terminal, and Developers. The "Developers" section is highlighted with a red box and a red arrow pointing to it from below. Below the sidebar, a message says "Viewing test data". The main area displays "TEST DATA" for Today and Yesterday, both showing \$0.00. It includes a timeline from 12:00 AM to 11:59 PM with a marker at "Now, 4:12 PM". To the right, there are sections for Deposited payouts (\$0.00) and Expected payouts (\$0.00), with a "View balance" link. At the bottom, there's a "Customize" button and a "Settings" link.

Developers Link

From dropdown list click on “API keys”.

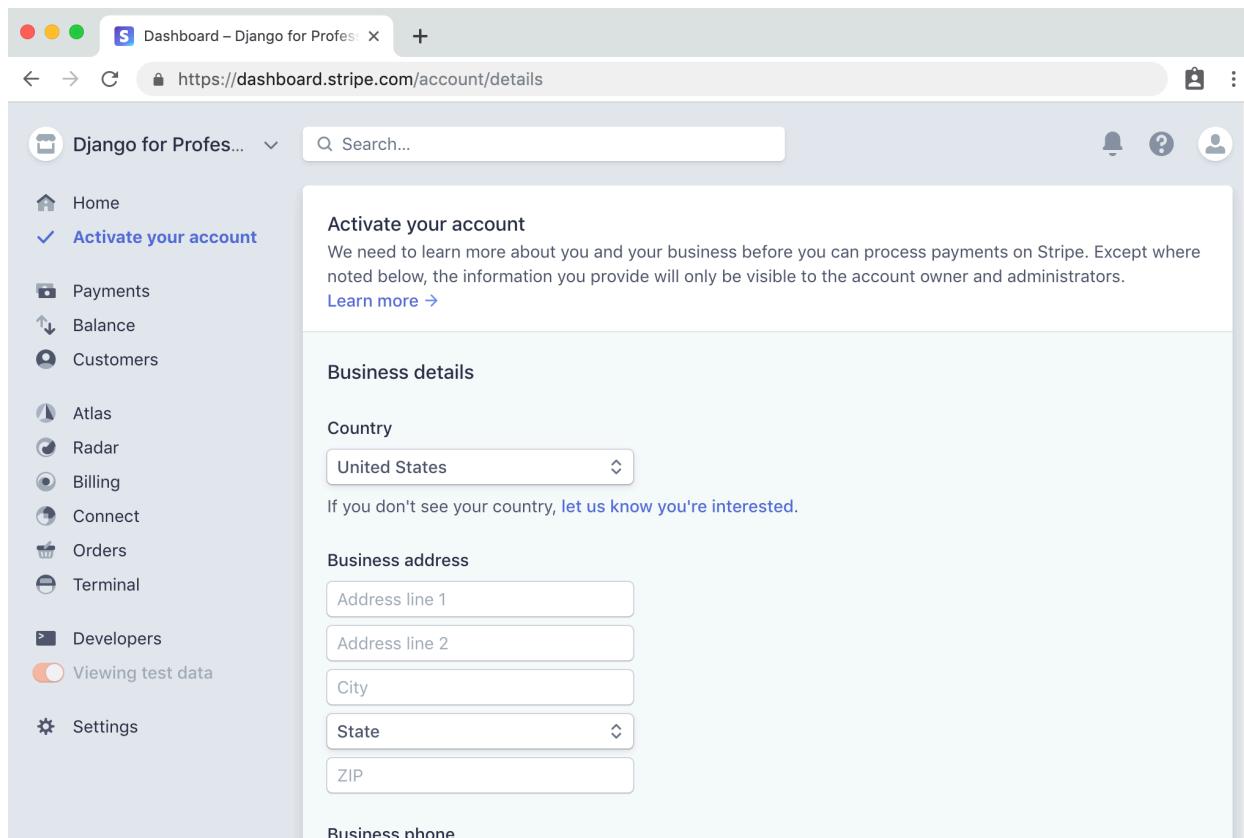
The screenshot shows the Stripe Dashboard for a user named "DFP Book". The left sidebar has a "Developers" section with "API keys" highlighted by a red box and an arrow pointing to it. The main area shows API version information (TEST DATA, VERSION: 2019-05-16, LAST USED: —, LAST 7 DAYS: —, USAGE: 0%), monitoring (0 successful requests, 0 errors), and API error distribution (GET, POST, DELETE). A caption "API Keys Link" is centered below the dashboard.

Each Stripe account has four API keys: two for testing and two for live use in production. Currently we are viewing the Test keys. We know this because there is a “TEST DATA” indicator at the top of the page and the keys (also called tokens) contain `test` in the name.

The screenshot shows the Stripe dashboard for the 'DFP Book' account. The left sidebar lists various sections: Home, Activate your account, Payments, Balance, Customers, Atlas, Radar, Billing, Connect, Orders, Terminal, Developers, API keys (which is selected), Webhooks, Events, Logs, Settings, and a toggle for viewing test data. The main content area is titled 'API keys' with a 'TEST DATA' button. It displays two sections: 'Standard keys' and 'Restricted keys'. Under 'Standard keys', there are two entries: 'Publishable key' with token 'pk_test_YzhTTabinrmqEFW2HvyIrmR400FzsSrQK8' and 'Secret key'. A 'Reveal test key token' button is next to the secret key entry. Under 'Restricted keys', it says 'No restricted keys'. There is also a '+ Create restricted key' button.

Test Keys

Viewing live keys requires both confirming your account via email and filling out an “Activate Your Account” page that is prompted if you click on the link at the top to toggle the keys.



Activate Your Account

Filling this page out is somewhat onerous, but we are dealing with money here so the extra information is warranted. However doing so now is optional. We can use the test keys and later swap in the live keys when we actually deploy the final site.

Publishable & Secret Keys

There are two types of keys for testing: a “publishable key” and a “secret key”. The publishable key will be embedded in the JavaScript on our webpage; it is therefore public and visible. The secret key is stored on the server and is for private use only. Keep this key secret!

That means using environment variables which we’ll do now. At the bottom of your bookstore_project/settings.py file, add the following two lines.

Code

```
# bookstore_project/settings.py

# Stripe

STRIPE_TEST_PUBLISHABLE_KEY=os.environ.get('STRIPE_TEST_PUBLISHABLE_KEY')
STRIPE_TEST_SECRET_KEY=os.environ.get('STRIPE_TEST_SECRET_KEY')
```

Add the environment variables to `docker-compose.yml` in the `web` section.

docker-compose.yml

```
version: '3.7'

services:
  web:
    build: .
    command: python /code/manage.py runserver 0.0.0.0:8000
    environment:
      - SECRET_KEY=p_o3vp1rg5)t^lxm9-43%0)s-=1qpeq%o7gfq+e4#!t+_ev82
      - DEBUG=True
      - STRIPE_TEST_PUBLISHABLE_KEY=<pk_test_your_publishable_key_here>
      - STRIPE_TEST_SECRET_KEY=<sk_test_your_secret_key_here>
    volumes:
      - .:/code
    ports:
      - 8000:8000
    depends_on:
      - db
  db:
    image: postgres:11
    volumes:
```

```
- postgres_data:/var/lib/postgresql/data/
```

volumes:

```
postgres_data:
```

Note that both environment variables should be filled with your unique API keys. Never share—especially in a book!—your actual Stripe secret key.

Then restart the Docker containers to load in the environment variables.

Command Line

```
$ docker-compose down
```

```
$ docker-compose up -d
```

Stripe Checkout

Step one is to add the Stripe Checkout form to our `orders/purchase.html` template.

Code

```
<!-- templates/orders/purchase.html -->
{% extends '_base.html' %}

{% block title %}Orders{% endblock title %}

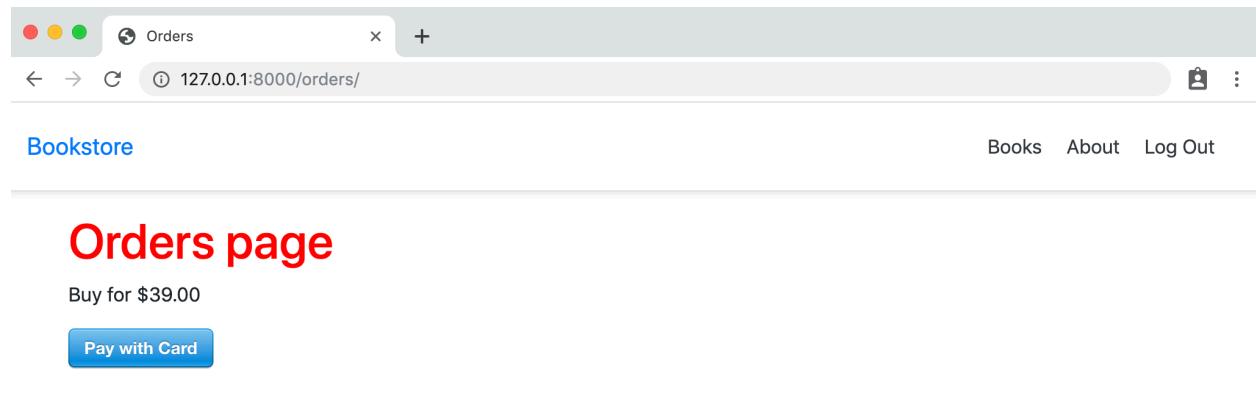
{% block content %}
<h1>Orders page</h1>


Buy for $39.00


<script src="https://checkout.stripe.com/checkout.js" class="stripe-button">
  data-key="{{ stripe_key }}"
</script>
```

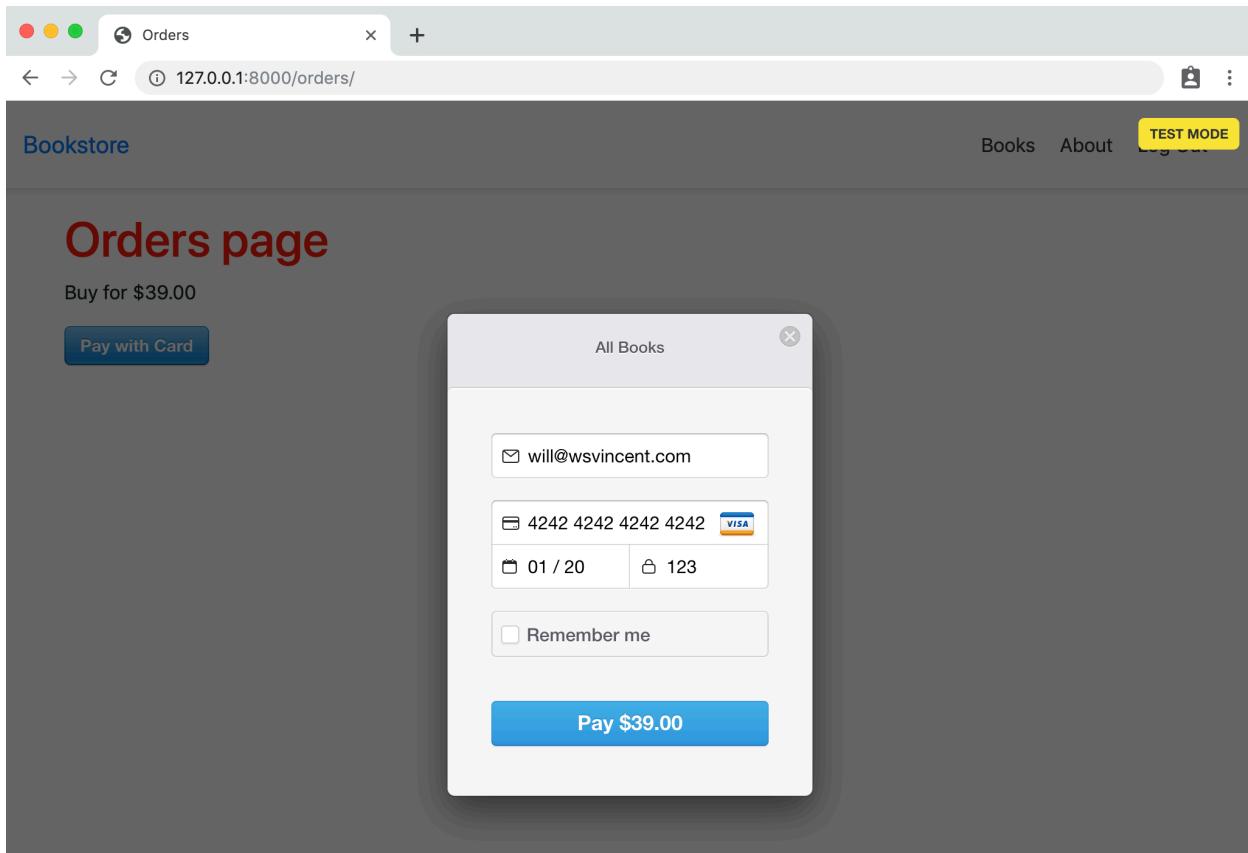
```
data-description="All Books"  
data-amount="3900"  
data-locale="auto">>  
</script>  
{% endblock content %}
```

If you refresh the web page at <http://127.0.0.1:8000/orders/> the default Stripe Checkout blue button appears.



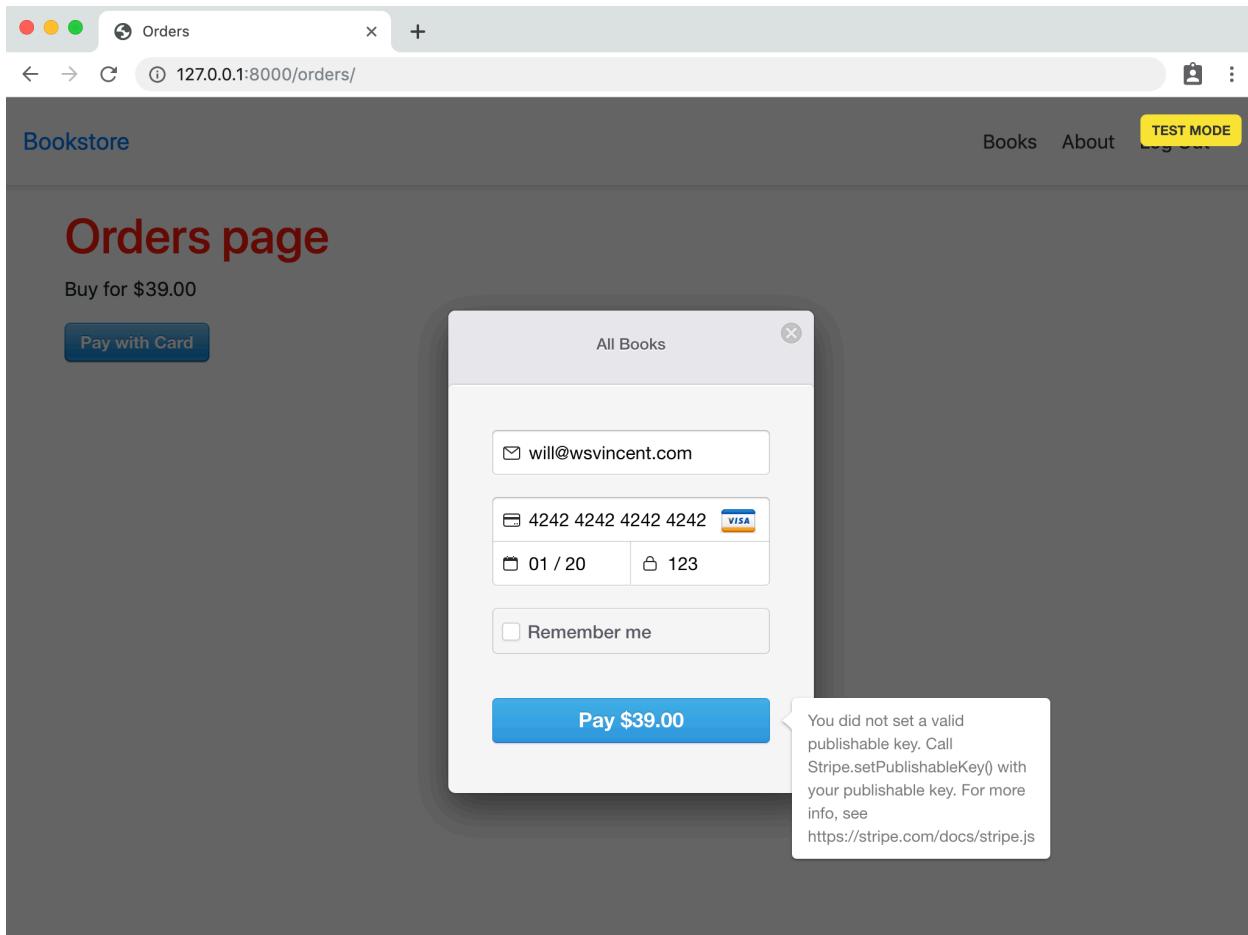
Orders Page

Click on the “Pay with Card” button and the Checkout modal appears. We can test the form by using one of several [test card numbers](#) Stripe provides. Let’s use 4242 4242 4242 4242. Make sure the expiration date is in the future and add any 3 numbers for the CVC.



Checkout Modal

But there's a problem after clicking on the "Pay \$39.00" blue button. Stripe notes that we did not set a valid publishable key!



Checkout Modal Error

This value needs to be passed into our template and while we could hard code this it's far better to pass in the value as a variable matching our environment variable setting.

In Django each template is rendered with context data provided by the `views.py` file. By overriding `get_context_data()` we can elegantly pass this information in with our `TemplateView`.

Update `orders/views.py` as follows.

Code

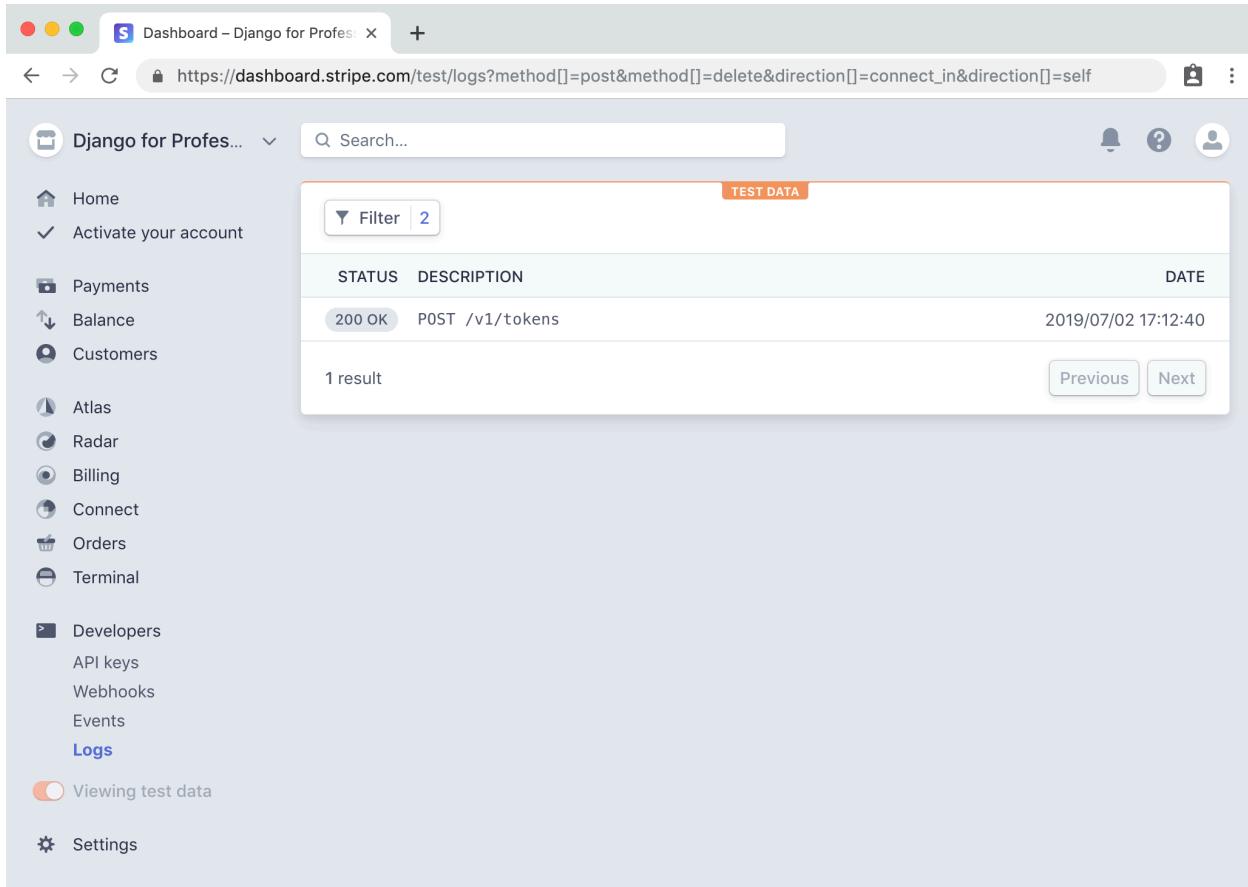
```
# orders/views.py

from django.conf import settings # new
from django.views.generic.base import TemplateView

class OrdersPageView(TemplateView):
    template_name = 'orders/purchase.html'

    def get_context_data(self, **kwargs): # new
        context = super().get_context_data(**kwargs)
        context['stripe_key'] = settings.STRIPE_TEST_PUBLISHABLE_KEY
        return context
```

Now refresh the web page and try again. It will “work” in that the button turns green with a checkmark indicating payment went through. If you look at the Stripe Dashboard and click on “Logs” under “Developers” in the left menu, you can see that tokens were created.



The screenshot shows the Stripe Dashboard interface. The left sidebar contains links for Home, Activate your account, Payments, Balance, Customers, Atlas, Radar, Billing, Connect, Orders, Terminal, Developers (API keys, Webhooks, Events, Logs), and Settings. The main content area is titled "TEST DATA" and shows a table with one result. The table has columns for STATUS, DESCRIPTION, and DATE. The single entry is: 200 OK POST /v1/tokens, dated 2019/07/02 17:12:40. There are "Previous" and "Next" buttons at the bottom of the table.

Stripe Dashboard Logs

But if you then click on “Payments” in the same lefthand menu, there are no charges. So what’s happening?

Think back to the Stripe flow. We have used the publishable key to send the credit card information to Stripe, and Stripe has sent us back a unique token for the order. But we haven’t used that token yet to make a charge! Recall that we send an order form to Stripe with the Publishable Key, Stripe validates it and sends back a token, and then we process the charge using both the token and our own Secret Key.

That’s the missing charge piece which we’ll implement now.

Charges

Creating a charge is not as hard as it seems. The first step is to make our payment button a Django form so we can pass in additional information via a charge view that we'll define next. And since it is a POST we include the `{% csrf_token %}` for additional security.

Code

```
<!-- templates/orders/purchase.html -->

{% extends '_base.html' %}

{% block title %}Orders{% endblock title %}

{% block content %}

<h1>Orders page</h1>
<p>Buy for $39.00</p>
<form action="{% url 'charge' %}" method="post">
    {% csrf_token %}
    <script src="https://checkout.stripe.com/checkout.js" class="stripe-button"
        data-key="{{ stripe_key }}"
        data-description="All Books"
        data-amount="3900"
        data-locale="auto">
    </script>
</form>
{% endblock content %}
```

Note it will redirect to a charge page so let's create that now.

Command Line

```
$ touch templates/orders/charge.html
```

Add some text to it.

Code

```
<!-- templates/orders/charge.html -->  
{% extends '_base.html' %}  
  
{% block title %}Charge{% endblock title %}  
  
{% block content %}  
<h2>Thank you for your order! You now have access to  
<a href="{% url 'book_list' %}">All Books</a>.</h2>  
{% endblock content %}
```

Then update our URL routes with the new `orders/charge/` page.

Code

```
# orders/urls.py  
from django.urls import path  
  
from .views import OrdersPageView, charge # new  
  
urlpatterns = [  
    path('charge/', charge, name='charge'), # new  
    path('', OrdersPageView.as_view(), name='orders'),  
]
```

Now for the “magic” logic which will occur in the `orders/views.py` file. Create a charge view that receives the token from Stripe, makes the charge, and then redirects to the charge page upon success.

At the top of the file import the `stripe` library we already installed. It will look for a secret key called `stripe.api_key` which we can set to that value. Then also import `render` which will be used for the function-based charge view.

Code

```
# orders/views.py

import stripe # new
from django.conf import settings
from django.views.generic.base import TemplateView
from django.shortcuts import render # new

stripe.api_key = settings.STRIPE_TEST_SECRET_KEY

class OrdersPageView(TemplateView):
    template_name = 'orders/purchase.html'

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['stripe_key'] = settings.STRIPE_TEST_PUBLISHABLE_KEY
        return context

def charge(request): # new
    if request.method == 'POST':
        charge = stripe.Charge.create(
            amount=3900,
            currency='usd',
```

```
        description='Purchase all books',
        source=request.POST['stripeToken']

    )

return render(request, 'orders/charge.html')
```

The `charge` function-based view assumes a `POST` request: we are sending data to Stripe here. We make a `charge` that includes the amount, currency, description, and crucially the `source` which has the unique token Stripe generated for this transaction called `stripeToken`. Then we return the `request` object and load the `charge.html` template.

Adding robust error handling here is probably warranted on a large site, but beyond the scope of this chapter.

Ok, refresh the orders page at <http://127.0.0.1:8000/orders/>. Click on the “Pay with Card” button again and use the credit card number 4242 4242 4242 4242, an expiration date in the future such as 01/22, and you’ll end up on our `charge` page.



Thank you for your order! You now have access to [All Books](#).

Charge Page

To confirm a charge was actually made, go back to the Stripe dashboard under “Payments” on the lefthand sidebar.

The screenshot shows the Stripe dashboard under the 'Payments' section. A single payment is listed with the status 'Succeeded'. The details are as follows:

AMOUNT	DESCRIPTION	CUSTOMER	DATE
\$39.00 USD	Succeeded ✓ Purchase all books	will@wsvincent.com	Jul 3, 2019, 8:43 AM

Below the table, it says '1 result'. At the bottom right of the main content area, there are 'Previous' and 'Next' buttons.

Stripe Payment

It worked!

Stripe + Permissions

There's one last step we must implement and that's to link up the order with a change in the given user's permissions. In other words, currently we are charging \$39 successfully but the user is not getting anything in return! There is still no access to the individual books.

But this is easily fixed. Again we'll focus solely on the `orders/views.py` file. At the top import `Permission` and then under the `charge` we first access the appropriate permission which is called `special_status`. Then we find the current user using `request.user`. And finally we add the given permission change to the user's permission

set.

Here is what it looks like in code:

Code

```
# orders/views.py

import stripe

from django.conf import settings
from django.contrib.auth.models import Permission # new
from django.views.generic.base import TemplateView
from django.shortcuts import render

stripe.api_key = settings.STRIPE_TEST_SECRET_KEY

class OrdersPageView(TemplateView):
    template_name = 'orders/purchase.html'

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['stripe_key'] = settings.STRIPE_TEST_PUBLISHABLE_KEY
        return context

    def charge(request): # new
        # Get the permission
        permission = Permission.objects.get(codename='special_status')

        # Get user
        u = request.user

        # Add to user's permission set
```

```
u.user_permissions.add(permission)

if request.method == 'POST':
    charge = stripe.Charge.create(
        amount=3900,
        currency='usd',
        description='Purchase all books',
        source=request.POST['stripeToken']
    )
    return render(request, 'orders/charge.html')
```

To test this out log in with our `testuser@email.com` account. It does not have access to this special permission which can be confirmed both within the `Users` section of the Admin under “Permissions” and also by the simple fact that if you try to access any individual books with this account, you won’t be able to!

Refresh the orders page and attempt to make a charge again. It will complete. Now visit the books list page and you can click through to each individual book. Success!

Templates

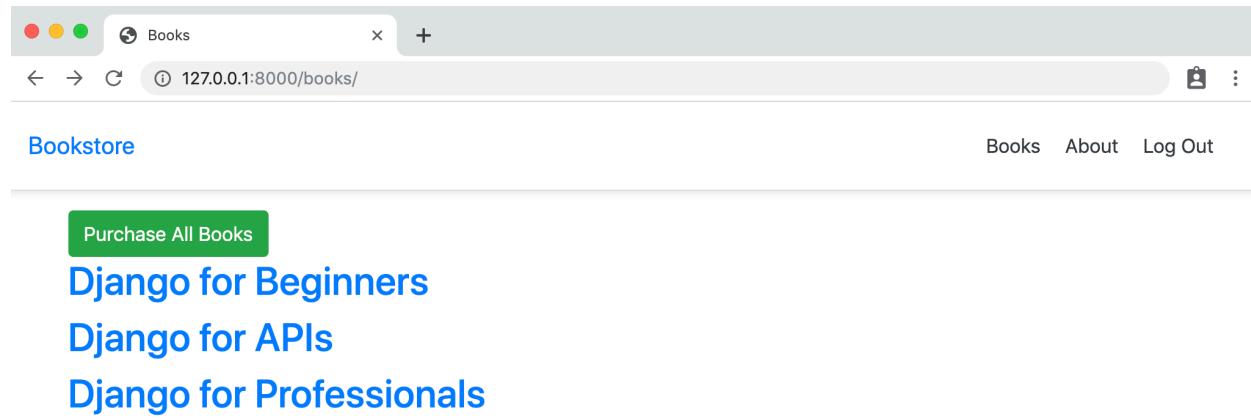
The final step is to add a button that links to the Orders page from the Books page. That means updating the `books/book_list.html` template by adding a Bootstrap styled button. Since the URL name of the orders page is `orders`—recall that this was set in `orders/urls.py`—we can use the `url` template tag to link to it.

The ultimate code is a one-line update at the top of the content in the file.

Code

```
<!-- templates/books/book_list.html -->  
{% extends '_base.html' %}  
  
{% block title %}Books{% endblock title %}  
  
{% block content %}  
  
<a href="{% url 'orders' %}" class="btn btn-success" >Purchase All Books</a>  
  
{% for book in book_list %}  
  
<div>  
  <h2><a href="{{ book.get_absolute_url }}">{{ book.title }}</a></h2>  
  </div>  
  
{% endfor %}  
  
{% endblock content %}
```

Refresh the books page and the button is now visible.



Clicking on it redirects to the Orders page. An additional step would be to add template logic that checks if the current logged-in user *already* has the proper permission, in which case the button would not be visible.

Tests

Typically the next step would be to add testing, but this example highlights an important point: it's hard to test integrations that involve 3rd party services. Doing so goes well beyond the abilities of core Django. However, it can be done with various mocking libraries and potentially using a service like [Cypress](#), but covering this is well beyond the scope of the book.

Git

There have been a lot of code changes in this chapter so make sure to commit everything with Git.

Command Line

```
$ git status  
$ git add .  
$ git commit -m 'ch14'
```

And if you have any errors make sure to look at your logs with `docker-compose logs` and compare your code with the [official source code on Github](#).

Conclusion

This chapter demonstrated how to add payments to a Django site. To review we used Stripe Checkout and our publishable key to send a customer's credit card information to Stripe. The Stripe API then sent us back a unique token for the customer, which we used alongside our secret key on the server to submit a charge. Alongside this charge we also updated the given user's permissions.

There are a number of additional steps that might be taken such as allowing payments for an individual book, a bundle, and so on. The process is the same as what we've done here: create a permission or even a group potentially and then link the charge to that.

Chapter 15: Search

Search is a fundamental feature of most websites and certainly anything e-commerce related like our Bookstore. In this chapter we will learn how to implement basic search with forms and filters. Then we will improve it with additional logic and touch upon ways to go even more deeply with search options in Django.

We only have three books in our database now but the code here will scale to as many books as we'd like.

Search functionality consists of two parts: a form to pass along a user search query and then a results page that performs a filter based on that query. Determining “the right” type of filter is where search becomes interesting and hard. But first we need to create both a form and the search results page.

We could start with either one at this point, but I'll we configure the filtering first and then the form.

Search Results Page

We'll start with the search results page. As with all Django pages that means adding a dedicated URL, view, and template. The implementation order doesn't particularly matter, but we will add them in that order.

Within `books/urls.py` add a `search/` path that will take a view called `SearchResultsView` and has a URL name of `search_results`.

Code

```
# books/urls.py

from django.urls import path

from .views import BookListView, BookDetailView, SearchResultsView # new

urlpatterns = [
    path('', BookListView.as_view(), name='book_list'),
    path('<uuid:pk>', BookDetailView.as_view(), name='book_detail'),
    path('search/', SearchResultsView.as_view(), name='search_results'), # new
]
```

Next up is the view `SearchResultsView` which is, for now, a listing of all available books. That's a prime candidate for using `ListView`. Its template will be called `search_results.html` and live within the `templates/books/` directory. The only new code is for `SearchResultsView` as we have previously imported both `ListView` and the `Book` model at the top of the file.

Code

```
# books/views.py

...
class SearchResultsView(ListView): # new
    model = Book
    context_object_name = 'book_list'
    template_name = 'books/search_results.html'
```

Last up is our template which must be created.

Command Line

```
$ touch templates/books/search_results.html
```

For now it will list all available book's by title, author, and price.

Code

```
<!-- templates/books/search_results.html -->

{% extends '_base.html' %}

{% block title %}Search{% endblock title %}

{% block content %}

<h1>Search Results</h1>

{% for book in book_list %}

<div>
    <h3><a href="#">book.get\_absolute\_url">{{ book.title }}</a></h3>
    <p>Author: {{ book.author }}</p>
    <p>Price: $ {{ book.price }}</p>
</div>
{% endfor %}

{% endblock content %}
```

The search results page is now available at <http://127.0.0.1:8000/books/search/>.

The screenshot shows a web browser window with the address bar displaying '127.0.0.1:8000/books/search/'. The page title is 'Search Results'. At the top, there are links for 'Books', 'About', and 'Log Out'. Below the title, three book entries are listed:

- Django for Beginners**
Author: William S. Vincent
Price: \$ 39.00
- Django for APIs**
Author: William S. Vincent
Price: \$ 39.00
- Django for Professionals**
Author: William S. Vincent
Price: \$ 39.00

At the bottom right of the page, there is a link labeled 'Search page'.

And there it is!

Basic Filtering

In Django a `QuerySet` is used to filter the results from a database model. Currently our search results page doesn't feel like one because it is outputting *all* results from the `Book` model. Ultimately we want to run the filter based on the user's search query, but first we'll work through multiple filtering options.

It turns out there are multiple ways to customize a queryset including via a `manager` on the model itself but to keep things simple, we can add a filter with just one line. So let's do that!

We can override the default `queryset` method on `ListView` which by default shows all results. The queryset documentation is quite robust and detailed, but often using

`icontains` is a good starting point. We will implement the filter based on the `title` that “contains” the name “beginners”. Note that `icontains` is case-insensitive; not all filters are.

Code

```
# books/views.py

class SearchResultsView(ListView):
    model = Book
    context_object_name = 'book_list'
    template_name = 'books/search_results.html'

    queryset = Book.objects.filter(title__icontains='beginners') # new
```

Refresh the search results page and now only a book with the title containing “beginners” is displayed. Success!



The screenshot shows a web browser window with a light gray header bar. On the left is a window control area with red, yellow, and green buttons. In the center is a search bar with a magnifying glass icon and the word "Search". To the right of the search bar are standard window controls: a close button (X), a plus sign (+), and a refresh/circular arrow icon. Below the header, the URL "127.0.0.1:8000/books/search/" is displayed. The main content area has a white background. At the top left, the word "Bookstore" is written in blue. To the right of "Bookstore" are three links: "Books", "About", and "Log Out", also in blue. A horizontal line separates this from the main content. The main content is titled "Search Results" in large red text. Below the title, the book "Django for Beginners" is listed in blue text. Underneath the book title, the author's name "Author: William S. Vincent" and the price "Price: \$ 39.00" are shown in smaller black text. At the very bottom of the main content area, the text "Search page for ‘beginners’" is centered in a small black font.

For basic filtering most of the time the `built-in queryset methods` of `filter()`, `all()`, `get()`, or `exclude()` will be enough. However there is also a very robust and detailed `QuerySet API` available as well that is worthy of further study.

Q Objects

Using `filter()` is powerful and it's even possible to [chain filters](#) together such as search for all titles that contain “beginners” and “django”. However often you'll want more complex lookups that can use “OR” not just “AND”; that's when it is time to turn to [Q objects](#).

Here's an example where we set the filter to look for a result that matches a title of either “beginners” or “api”. It's as simple as importing `Q` at the top of the file and then subtly tweaking our existing query. The `|` symbol represents the “or” operator. We can filter on any available field: not just `title` but also `author` or `price` as desired.

As the number of filters grows it can be helpful to separate out the `queryset` override via `get_queryset()`. That's what we'll do here but note that this choice is optional.

Code

```
# books/views.py

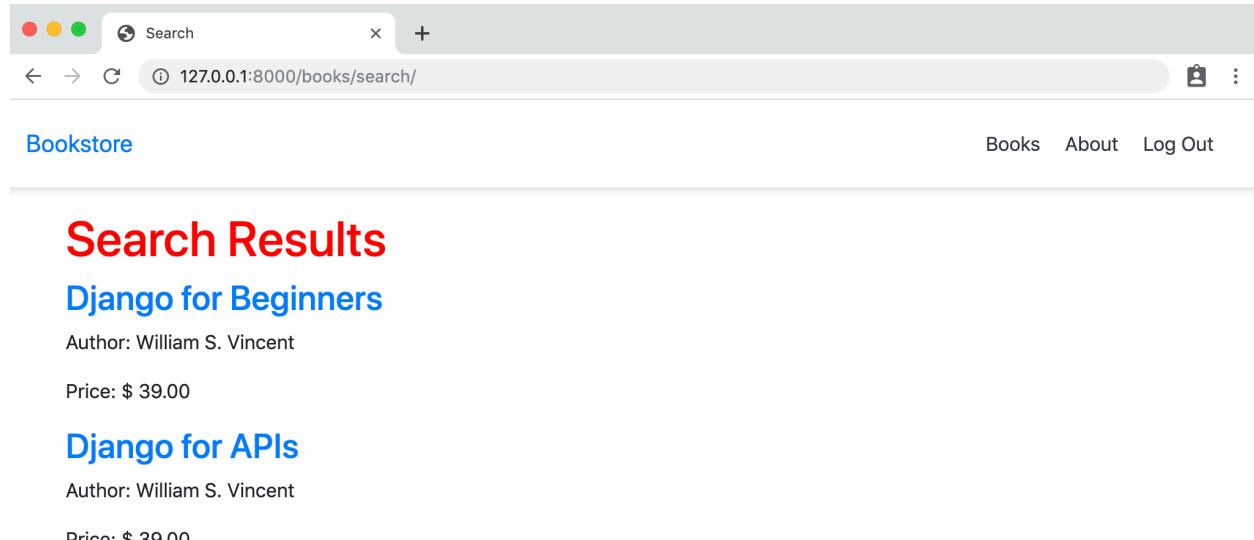
from django.db.models import Q # new

...

class BookListView(ListView):
    model = Book
    context_object_name = 'book_list'
    template_name = 'books/book_list.html'

    def get_queryset(self): # new
        return Book.objects.filter(
            Q(title__icontains='beginners') | Q(title__icontains='api')
        )
```

Refresh the search results page to see the new result.



The screenshot shows a web browser window with a title bar "Search" and a URL bar "127.0.0.1:8000/books/search/". The main content area displays a "Bookstore" page titled "Search Results". It lists two books: "Django for Beginners" by William S. Vincent (Author: William S. Vincent, Price: \$ 39.00) and "Django for APIs" by William S. Vincent (Author: William S. Vincent, Price: \$ 39.00). A link "Search with Q objects" is visible at the bottom of the page.

Search with Q objects

Now let's turn our attention to the corresponding search form so that rather than hardcode our filters in we can populate them based on the user's search query.

Forms

Fundamentally a web form is simple: it takes user input and sends it to a URL via either a `GET` or `POST` method. However in practice this fundamental behavior of the web can be monstrously complex.

The first issue is sending the form data: where does the data actually go and how do we handle it once there? Not to mention there are numerous security concerns whenever we allow users to submit data to a website.

There are only two options for “how” a form is sent: either via `GET` or `POST` HTTP methods.

A `POST` bundles up form data, encodes it for transmission, sends it to the server, and

then receives a response. Any request that changes the state of the database—creates, edits, or deletes data—should use a `POST`.

A `GET` bundles form data into a string that is added to the destination URL. `GET` should only be used for requests that do not affect the state of the application, such as a search where nothing within the database is changing, basically we’re just doing a filtered list view.

If you look at the URL after visiting `Google.com` you’ll see your search query in the actual search results page URL itself.

For more information, Mozilla has detailed guides on both [sending form data](#) and [form data validation](#) that are worth reviewing if you’re not already familiar with form basics.

Search Form

Let’s add a basic search form to the current homepage right now. It can easily be placed in the navbar or on a dedicated search page as desired in the future.

We start with HTML `<form>` tags and use Bootstrap’s styling to make them look nice. The `action` specifies where to redirect the user after the form is submitted, which will be the `search_results` page. As with all URL links this is the URL name for the page. Then we indicate the desired `method` of `get` rather than `post`.

The second part of the form is the `input` which contains the user search query. We provide it with a variable `name`, `q`, which will be later visible in the URL and also available in the views file. We add Bootstrap styling with the `class`, specify the `type` of input is text, add a `Placeholder` which is default text that prompts the user. The last part, `aria-label`, is the name provided to screen reader users. Accessibility is a big part of web development and should always be considered from the beginning: include `aria-labels` with all your forms!

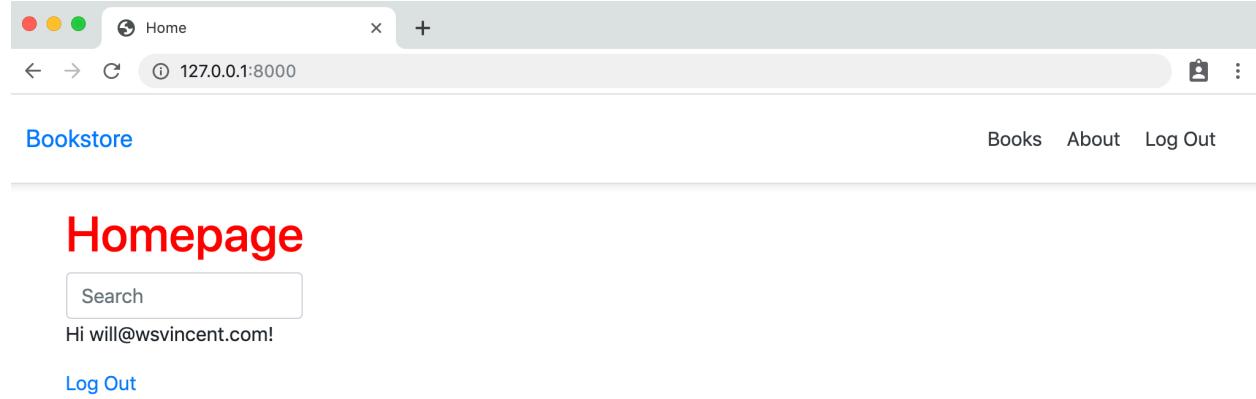
Code

```
<!-- templates/home.html -->  
{% extends '_base.html' %}  
{% load static %}  
  
{% block title %}Home{% endblock title %}  
  
{% block content %}  


# Homepage</h1> method="get"> <input name="q" class="form-control mr-sm-2" type="text" placeholder="Search" aria-label="Search"> </form> {% endblock content %}

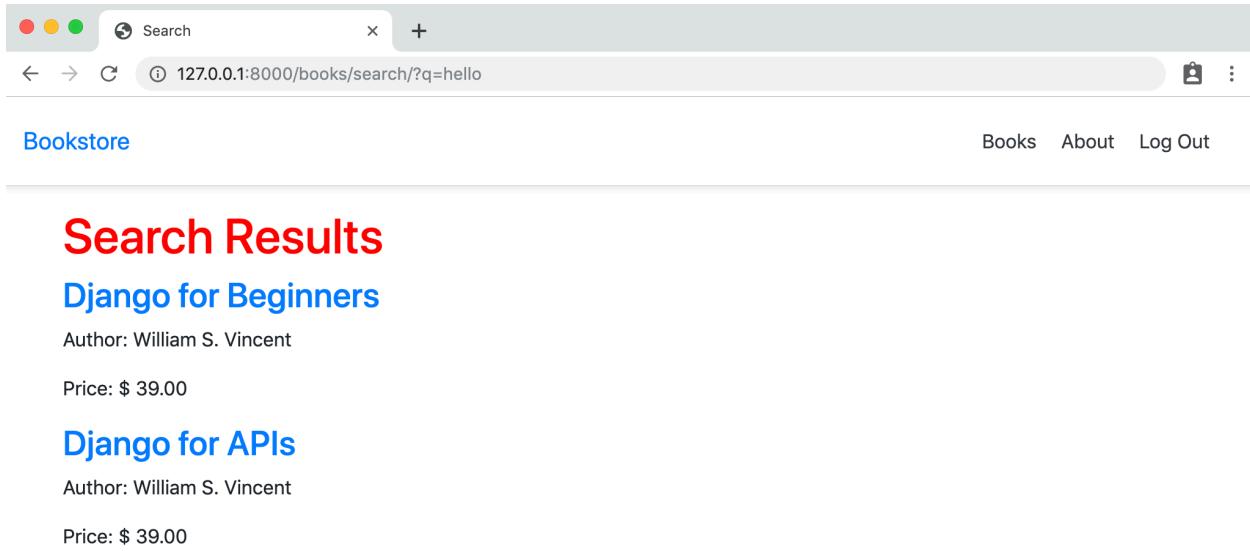

```

Navigate to the homepage and the new search box is present.

**Homepage with search box**

Try inputting a search, for example for “hello”. Upon hitting Return you are redirected to the search results page. Note in particular the URL contains the search query

<http://127.0.0.1:8000/books/search/?q=hello>.



The screenshot shows a web browser window with the address bar containing '127.0.0.1:8000/books/search/?q=hello'. The page title is 'Bookstore'. The main content area displays 'Search Results' in red. Below it, two book entries are listed: 'Django for Beginners' by William S. Vincent at \$39.00 and 'Django for APIs' by William S. Vincent at \$39.00.

Title	Author	Price
Django for Beginners	William S. Vincent	\$ 39.00
Django for APIs	William S. Vincent	\$ 39.00

URL with query string

However the results haven't changed! And that's because our `SearchResultsView` still has the hardcoded values from before. The last step is to take the user's search query, represented by `q` in the URL, and pass it in to the actual search filters.

Code

```
# books/views.py

class SearchResultsView(ListView):
    model = Book
    context_object_name = 'book_list'
    template_name = 'books/search_results.html'

    def get_queryset(self): # new
        query = self.request.GET.get('q')
        return Book.objects.filter(
            Q(title__icontains=query) | Q(author__icontains=query)
        )
```

What changed? We added a `query` variable that takes the value of `q` from the form submission. Then updated our filter to use `query` on either a `title` or an `author` field. That's it! Refresh the search results page—it still has the same URL with our `query`—and the result is expected: no results on either title or author for “hello”.

Go back to the homepage and try a new search such as for “django” or “beginners” or “william” to see the complete search functionality in action.

Git

Make sure to save our current work in this chapter by committing the new code to Git.

Command Line

```
$ git status  
$ git add .  
$ git commit -m 'ch15'
```

The official source code for this chapter is [available on Github](#).

Conclusion

Our basic search is now complete, but we've only scratched the surface of potential search optimizations. For example, maybe we want a button added to the search form that could be clicked in addition to hitting the Return key? Or better yet include form validation. Beyond filtering with ANDs and ORs there are other factors if we want a Google-quality search, things like relevancy and much more.

A next-step would be to use a third-party package like [django-watson](#) or [django-haystack](#) to add more advanced search.

Or given that we're using PostgreSQL as the database take advantage of its [full text search](#).

A final option is either use an enterprise-level solution like [ElasticSearch](#) that must be running on a separate server (not the hardest thing with Docker), or rely on a hosted solution like [Swiftype](#) or [Algolia](#).

In the next chapter we'll explore the many performance optimizations available in Django as we prepare our Bookstore project for eventual deployment.

Chapter 16: Performance

The first priority for any website is that it must work properly and contain proper tests. But if your project is fortunate enough to receive a large amount of traffic the focus quickly shifts to performance and making things as efficient as possible. This is a fun and challenging exercise for many engineers, but it can also be a trap.

The computer scientist Donald Knuth has [a famous quote](#) worth reading in its entirety:

“The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.”

While it’s important to set up proper monitoring so you *can* optimize your project later on, don’t focus too much on it upfront. There’s no way to properly mimic production environments locally. And there is no way to predict exactly how a site’s traffic will look. But it is possible to spend far too much time eeking out tiny performance gains in the early stages instead of talking to users and making larger code improvements!

In this chapter we’ll focus on the broad strokes of Django-related performance and highlight areas worth further investigation at scale. Generally speaking performance comes down to four major areas: optimizing database queries, caching, indexes, and compressing front-end assets like images, JavaScript, and CSS.

django-debug-toolbar

Before we can optimize our database queries we need to see them. And for this the default tool in the Django community is the third-party package [django-debug-toolbar](#).

It comes with a configurable set of panels to inspect the complete request/response cycle of any given page.

Per usual we can install it within Docker and stop our running containers.

Command Line

```
$ docker-compose exec web pipenv install django-debug-toolbar==1.11  
$ docker-compose down
```

There are three separate configurations to set in our `bookstore_project/settings.py` file:

1. `INSTALLED_APPS`
2. `Middleware`
3. `INTERNAL_IPS`

First add Debug Toolbar to the `INSTALLED_APPS` configuration. Note that the proper name is `debug_toolbar` not `djang DEBUG_toolbar` as might be expected.

Code

```
# bookstore_project/settings.py  
  
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'django.contrib.sites',  
  
    # Third-party
```

```
'crispy_forms',
'allauth',
'allauth.account',
'debug_toolbar', # new

# Local
'users.apps.UsersConfig',
'pages.apps.PagesConfig',
'books.apps.BooksConfig',
'orders.apps.OrdersConfig',
]
```

Second, add Debug Toolbar to the Middleware where it is primarily implemented.

Code

```
# bookstore_project/settings.py

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    'debug_toolbar.middleware.DebugToolbarMiddleware', # new
]
```

And third, set the INTERNAL_IPS as well. If we were not in Docker this could be set to '127.0.0.1', however, since we're running our web server within Docker an additional

step is required so that it matches the machine address of Docker. Add the following lines at the bottom of `bookstore_project/settings.py`.

Code

```
# bookstore_project/settings.py

...
# django-debug-toolbar
import socket
hostname, _, ips = socket.gethostbyname_ex(socket.gethostname())
INTERNAL_IPS = [ip[:-1] + "1" for ip in ips]
```

Phew. That looks a bit scary, but basically it ensures that our `INTERNAL_IPS` matches that of our Docker host.

Now rebuild the base image so it contains the package and the updated settings configuration.

Command Line

```
$ docker-compose up -d --build
```

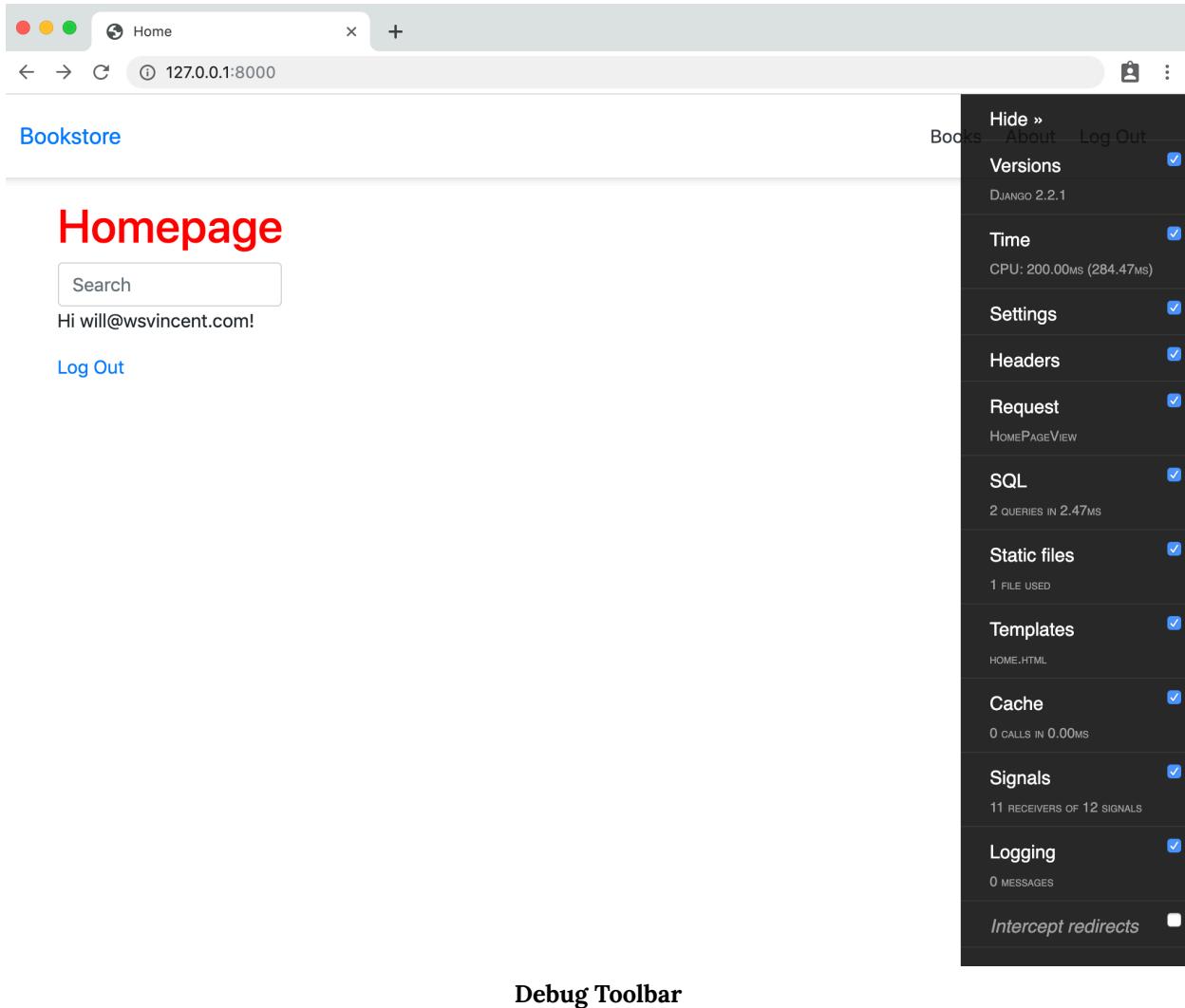
There's one last step and that is to update our URLconf. We only want Debug Toolbar to appear if `DEBUG` is true so we'll add logic to display it only in this case.

Code

```
# bookstore_project/urls.py

...
if settings.DEBUG:
    import debug_toolbar
    urlpatterns = [
        path('__debug__/', include(debug_toolbar.urls)),
    ] + urlpatterns
```

Now if you refresh the homepage you'll see the `djangodjango-debug-toolbar` on the righthand side.



Debug Toolbar

If you click the “Hide” link on top it becomes a much smaller sidebar on the righthand side of the page.

Analyzing Pages

Debug Toolbar has [many possible customizations](#) but the default settings visible tell us a lot about our homepage. For instance, we can see the current version of Django being used as well as the Time it took to load the page. Also the specific request called which was `HomePageView`. This may seem obvious but on large codebases especially if

you are jumping in as a new developer, it may not be obvious which view is calling which page. Debug Toolbar is a helpful quickstart to understanding existing sites.

Probably the most useful item, however, is SQL which shows two queries are being run and the time for them. If you click on it even more data appears.

The screenshot shows the Django Debug Toolbar interface. At the top, there's a navigation bar with icons for Home, Back, Forward, and a URL field showing 127.0.0.1:8000. Below that is a yellow header bar with the text "SQL queries from 1 connection". The main content area is divided into several sections:

- default**: Shows 2.47 ms (2 queries). It lists two SQL queries with their execution times (1.57 ms and 0.90 ms) and provides "Sel" and "Expl" buttons for each.
- Timeline**: A horizontal bar showing the timing of the queries.
- Time (ms)**: The total time taken by the queries.
- Action**: Buttons for selecting or exploring the queries.
- Versions**: Shows Djang 2.2.1.
- Time**: Shows CPU: 200.00MS (284.47MS).
- Settings**
- Headers**
- Request**: Shows REQUEST_METHOD: GET, PATH_INFO: /home, and SERVER_NAME: 127.0.0.1.
- SQL**: Shows 2 QUERIES IN 2.47MS. It lists the two queries again with their details.
- Static files**: Shows 1 FILE USED.
- Templates**: Shows HOME.HTML.
- Cache**: Shows 0 CALLS IN 0.00MS.
- Signals**: Shows 11 RECEIVERS OF 12 SIGNALS.
- Logging**: Shows 0 MESSAGES.
- Intercept redirects**: A checkbox that is unchecked.

Debug Toolbar

Large and poorly optimized sites can have hundreds or even thousands of queries being run on a single page!

select_related and prefetch_related

What are the options if you do find yourself working on a Django site with way too many SQL queries per page? In general, though, fewer large queries will be faster than many smaller queries, though it's possible and required to test this in practice. Two common techniques for doing so are `select_related()` and `prefetch_related()`.

`select_related` is used for single-value relationships through a forward one-to-many or a one-to-one relationship. It creates a SQL join and includes the fields of the related object in the `SELECT` statement, which results in all related objects being included in a single more complex database query. This single query is typically more performant than multiple, smaller queries.

`prefetch_related` is used for a set or list of objects like a many-to-many or many-to-one relationship. Under the hood a lookup is done for each relationship and the “join” occurs in Python, not SQL. This allows it to prefetch many-to-many and many-to-one objects, which cannot be done using `select_related`, in addition to the foreign key and one-to-one relationships that are supported by `select_related`.

Implementing one or both on a website is a common first pass towards reducing queries and loading time for a given page.

Caching

Consider that our Bookstore project is a dynamic website. Each time a user requests a page our server has to make various calculations including database queries, template rendering, and so on before servicing it. This takes time and is much slower than simply reading a file from a static site where the content does not change.

On large sites, though, this type of overhead can be quite slow and caching is one of the first solutions in a web developer's tool bag. Implementing caching on our

current project is definitely overkill, but we will nonetheless review the options and implement a basic version.

A cache is an in-memory storing of an expensive calculation. Once executed it doesn't need to be run again! The two most popular options are [Memcached](#) which features native Django support and [Redis](#) which is commonly implemented with the [django-redis](#) third-party package.

Django has its own [cache framework](#) which includes four different caching options in descending order of granularity:

- 1) The [per-site cache](#) is the simplest to set up and caches your entire site.
- 2) The [per-view cache](#) lets you cache individual views.
- 3) [Template fragment caching](#) lets you specify a specific section of a template to cache.
- 4) The [low-level cache API](#) lets you manually set, retrieve, and maintain specific objects in the cache.

Why not just cache everything all the time? One reason is that cache memory is expensive, as it's stored as RAM: think about the cost of going from 8GB to 16GB of RAM on your laptop vs. 256GB to 512GB of hard drive space. Another is the cache must be "warm," that is filled with updated content, so depending upon the needs of a site, optimizing the cache so it is accurate, but not wasteful, takes quite a bit of tuning.

If you wanted to implement per-site caching, which is the simplest approach, you'd add `UpdateCacheMiddleware` at the very top of the `MIDDLEWARE` configuration in `bookstore--project/settings.py` and `FetchFromCacheMiddleware` at the very bottom. Also set three additional fields `CACHE_MIDDLEWARE_ALIAS`, `CACHE_MIDDLEWARE_SECONDS`, and `CACHE_MIDDLEWARE_KEY_PREFIX`.

Code

```
# bookstore_project/settings.py

MIDDLEWARE = [
    'django.middleware.cache.UpdateCacheMiddleware', # new
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'debug_toolbar.middleware.DebugToolbarMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    'debug_toolbar.middleware.DebugToolbarMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware', # new
]

CACHE_MIDDLEWARE_ALIAS = 'default'
CACHE_MIDDLEWARE_SECONDS = 604800
CACHE_MIDDLEWARE_KEY_PREFIX = ''
```

The only default you might want to adjust is `CACHE_MIDDLEWARE_SECONDS` which is the default number of seconds (600) to cache a page. After the period is up, the cache expires and becomes empty. A good default when starting out is 604800 seconds or 1 week (60secs x 60minutes x 168hours) for a site with content that doesn't change very often. But if you find your cache filling up rapidly or you are running a site where the content changes on a frequent basis, shortening this setting is a good first step.

Implementing caching is strictly optional at this point though. Once a website is up and running the need for caching—per site, per page, and so on—will quickly become

apparent. There is also extra complexity as Memcache must be run as a separate instance. On the hosting service Heroku, which we'll use in chapter 18 for deployment, there is a free tier available via [Memcachier](#).

Indexes

[Indexing](#) is a common technique for speeding up database performance. It is a separate data structure that allows faster searches and is typically only applied to the primary key in a model. The downside is that indexes require additional space on a disk so they must be used with care.

Tempting as it is to simply add indexes to primary keys from the beginning, it is better to start without them and only add them later based on production needs. A general rule of thumb is that if a given field is being used frequently, such as 10-25% of all queries, it is a prime candidate to be indexed.

Historically an index field could be created by adding `db_index=True` to any model field. For example, if we wanted to add one to the `id` field in our `Book` model it would look as follows (don't actually implement this though).

Code

```
# books/models.py

...
class Book(models.Model):
    id = models.UUIDField(
        primary_key=True,
        db_index=True, # new
        default=uuid.uuid4,
        editable=False)
...
```

This change would need to be added via a migration file and migrated.

Starting in [Django 1.11](#) class-based model indexes were added so can include in the [Meta section instead](#). So you could write the previous index as follows:

Code

```
# books/models.py

...
class Book(models.Model):
    id = models.UUIDField(
        primary_key=True,
        default=uuid.uuid4,
        editable=False)
    ...

class Meta:
    indexes = [ # new
        models.Index(fields=['id'], name='id_index'),
    ]
    permissions = [
        ("special_status", "Can read all books"),
    ]
```

Since we've changed the model we must create a migrations file and apply it.

Command Line

```
$ docker-compose exec web python manage.py makemigrations books  
$ docker-compose exec web python manage.py migrate
```

django-extensions

Another very popular third-party package for inspecting a Django project is [django-extensions](#) which adds a number of helpful [custom extensions](#).

One that is particularly helpful is [shell_plus](#) which will autoload all models into the `shell` which makes working with the Django ORM much easier.

Front-end Assets

A final major source of bottlenecks in a website is loading front-end assets. CSS and JavaScript can become quite large and therefore tools like [django-compressor](#) can help to minimize their size.

Images are often the first place to look in terms of asset size. The static/media file set up we have in place will scale to a quite large size, but for truly large sites it is worth investigating the use of a [Content Delivery Network \(CDN\)](#) for images instead of storing them on the server filesystem.

You can also serve different size images to users. For example, rather than shrink down a large book cover for a list or search page you could store a smaller thumbnail version instead and serve *that* where needed. The third-party [easy-thumbnails](#) package is a good place to start for this.

A fantastic free e-book on the topic is [Essential Image Optimization](#) by Addy Osmani that goes into depth on image optimization and automations.

As a final check there are automated tests for front-end speed such as Google's [PageSpeed Insights](#) that will assign a score based on how quickly a page loads.

Git

There's been a lot of code changes in this chapter so make sure to commit everything with Git.

Command Line

```
$ git status  
$ git add .  
$ git commit -m 'ch16'
```

If you have any errors make sure to look at your logs with `docker-compose logs` and compare your code with the [official source code on Github](#).

Conclusion

There is an almost endless list of performance optimizations that can be applied to a project. But take care to recall Donald Knuth's sage advice and not go too crazy on this. Bottlenecks will reveal themselves in production and should largely be addressed then; not in advance.

You should remember that performance problems are a good problem to have! They are fixable and mean that your project is being heavily used.

Chapter 17: Security

The World Wide Web is a dangerous place. There are many bad actors and even more automated bots that will try to hack into your website and cause ill. Therefore understanding and implementing security features is a must in any website.

Fortunately Django has a very strong record when it comes to security thanks to its years of experience handling web security issues as well as a robust and regular security update cycle. [New feature releases](#) come out roughly every 9 months such as 2.2 to 3.0 but there are also patch releases around bugs and security like 2.2.2 to 2.2.3 that occur almost monthly.

However as with any tool it's important to implement security features correctly and in this chapter we'll cover how to do so in our bookstore project.

Social Engineering

The biggest security risk to any website is ultimately not technical: it is people. The term [social engineering](#) refers to the technique of finding individuals with access to a system who will willingly or unwillingly share their login credentials with a bad actor.

These days [phishing](#) is probably the most likely culprit if you are in a technical organization. All it takes is one bad click on an email link for a malicious actor to potentially gain access to the system, or at least all the access the compromised employee has.

To mitigate this risk, implement a robust permissions scheme and only provide the exact security access an employee needs, not more. Does every engineer need access

to the production database? Probably not. Do non-engineers need write access? Again, probably not. These are discussions best had up front and a good default is to only add permissions as needed, not to default to superuser status for everyone!

Django updates

Keeping your project up-to-date with the latest version of Django is another important way to stay secure. And I don't just mean being current with the [latest feature release](#) (2.2, 3.0, 3.1, etc) which comes out roughly every 9 months. There are also monthly security patch updates that take the form of 2.2.1, 2.2.2, 2.2.3, etc.

What about **long-term support (LTS) releases**? Certain feature releases designated as LTS receive security and data loss fixes for a guaranteed period of time, usually around 3 years. For example, Django 2.2 is an LTS and will be supported into 2022 when Django 4.0 is released as the next LTS version. Can you stay on LTS versions? Yes. Should you? No. It is better and more secure to stay up-to-date.

Resist the temptation and reality of many real-world projects which is not to devote a portion of developer time to staying current with Django versions. A website is like a car: it needs regular maintenance to run at its best. You are only compounding the problem if you put off updates.

How to update? Django features [deprecation warnings](#) that can and should be run for each new release by typing `python -Wa manage.py test`. It is far better to update from 2.0 to 2.1 to 2.2 and run the deprecation warnings each time rather than skipping multiple versions.

Deployment Checklist

To assist with deployment and checking security settings, the Django docs contain a dedicated [deployment checklist](#) that further describes security settings.

Even better there is a command we can run to automate Django's recommendations, `python manage.py check --deploy`, that will check if a project is deployment ready. It uses the Django [system check framework](#) which can be used to customize similar commands in mature projects.

Since we are working in Docker we must prepend `docker-compose exec web` to the command though.

Command Line

```
$ docker-compose exec web python manage.py check --deploy
```

```
System check identified some issues:
```

```
WARNINGS:
```

```
...
```

```
System check identified 9 issues (0 silenced).
```

How nice! A descriptive and lengthy list of issues which we can go through one-by-one to prepare our Bookstore project.

Local vs. Production

Ultimately our local development settings will differ from those used in production. There are a number of techniques to manage this complexity including the use of multiple `settings.py` files, however, a cleaner approach is to take advantage of our existing use of Docker and environment variables.

There already exist environment variables within `bookstore_project/settings.py` so rather than update that file we can simply create a separate `docker-compose-prod.yml` file just for production.

Command Line

```
$ touch docker-compose-prod.yml
```

For now copy and paste the existing `docker-compose.yml` file into the `docker-compose-prod.yml` file and remove any *volumes*. The volumes serve to persist information locally within the Docker containers but are not needed in production.

One addition we'll make is adding `ENVIRONMENT=production` as an environment variable which will allow for additional logic shortly.

`docker-compose-prod.yml`

```
version: '3.7'

services:
  web:
    build: .
    command: python /code/manage.py runserver 0.0.0.0:8000
    environment:
      - ENVIRONMENT=production
      - SECRET_KEY=p_o3vp1rg5)t^lxm9-43%0)s-=1qpeq%o7gfq+e4#*!t+_ev82
      - DEBUG=1
      - STRIPE_TEST_PUBLISHABLE_KEY=<pk_test_your_publishable_key_here>
      - STRIPE_TEST_SECRET_KEY=<sk_test_your_secret_key_here>
    ports:
      - 8000:8000
    depends_on:
```

```
- db  
db:  
  image: postgres:11
```

Make sure SECRET_KEY, STRIPE_TEST_PUBLISHABLE_KEY, and STRIPE_TEST_SECRET_KEY contain your own specific values, not the placeholders included here!

We'll also add a similar ENVIRONMENT variable to `docker-compose.yml` but this one will equal development.

docker-compose.yml

```
version: '3.7'  
  
services:  
  web:  
    build: .  
    command: python /code/manage.py runserver 0.0.0.0:8000  
    environment:  
      - ENVIRONMENT=development  
    ...
```

DEBUG

First up is the `DEBUG` setting, currently set to “True” with the number “1” in the Compose file. Debug should **never** be on when deploying a site to production.

One of debug modes main features is detailed error pages that display a host of metadata about the environment including most currently defined settings. This is helpful for spotting errors but a recipe for potential hackers to compromise a website.

Debug mode also remembers every SQL query executed which helps with debugging, but dramatically consumes memory on a production server.

Therefore we must switch `DEBUG` to “0”, or “False”, in `docker-compose-prod.yml`.

docker-compose-prod.yml

`DEBUG=0`

ALLOWED HOSTS

Next up is the `ALLOWED_HOSTS` configuration which controls the host/domain names our Django site can serve. It likely exists right below `DEBUG` in the `bookstore-project/settings.py` file. By default in development it is set to `[]`, an empty list. But for production, when `DEBUG` is `False`, it must be set explicitly and include values.

The two ways we access it locally which are via either `127.0.0.1` or `localhost`. We will be using Heroku for deployments in the next section and all of its hosted sites are at the subdomain of '`.herokuapp.com`', so we can add that now.

Code

```
# bookstore_project/settings.py  
ALLOWED_HOSTS = ['.herokuapp.com', 'localhost', '127.0.0.1']
```

To confirm, spin down the Docker host now and restart it via the `-f` flag to specify an **alternate compose file**. By default Docker assumes a `docker-compose.yml` so adding the `-f` flag is unnecessary in that case.

Command Line

```
$ docker-compose down  
$ docker-compose -f docker-compose-prod.yml up -d --build  
$ docker-compose exec web python manage.py migrate
```

The `--build` flag is added for the initial building of the image, along with all the corresponding software packages, for the new compose file. Also `migrate` is run on the new database. This is an entirely new instance of our project! As such it won't have a superuser account or any of our data such as book information. But that's OK for now; that information can be added in production and for now our focus to is pass the deployment checklist!

Run the `--deploy` check again.

Command Line

```
$ docker-compose exec web python manage.py check --deploy  
...  
System check identified 7 issues (0 silenced).
```

There is still a long string of warnings but only 7 issues now, not 9 because `DEBUG` and `ALLOWED_HOSTS` have been fixed. Success! Two down and many more to go.

Web Security

Now it is time for a brief dive into web security. Django handles most common use cases, however, it is still vital to understand frequent attack methods and the steps Django takes to mitigate them. You can find an overview on the [Django security page](#), but we'll go into further depth here.

Django comes by default with a number of additional [security middlewares](#) that guard against other request/response cycle attacks.

A full explanation of each is beyond the scope of this book, but it is worth reading about the protections provided by the Django security team over the years. Do not change the defaults without good cause.

SQL injection

Let's start with a [SQL injection attack](#) which occurs when a malicious user can execute arbitrary SQL code on a database. Consider a log in form on a site. What happens if a malicious user instead types `DELETE from users WHERE user_id=user_id?` If this is run against the database without proper protections it could result in the deletion of all user records! Not good. This [XKCD comic](#) provides a humorous though potentially accurate example of how this can occur.

Fortunately the Django ORM automatically sanitizes user inputs by default when constructing querysets to prevent this type of attack. Where you need to be careful is that Django does provide the option to execute [custom sql](#) or [raw queries](#). These should both be used with extreme caution since they could open up a vulnerability to SQL injection.

The non-profit Open Web Application Security Project (OWASP) has a fantastic and very detailed [SQL Injection Cheat Sheet](#) that is recommended for further reading.

XSS (Cross Site Scripting)

[Cross-site scripting \(XSS\)](#) is another classic attack that occurs when an attacker is able to inject small bits of code onto web pages viewed by other people. This code,

typically JavaScript, if stored in the database will then be retrieved and displayed to other users.

For example, consider the form used for writing book reviews on our current site. What if instead of typing, “This book was great” a user typed something with JavaScript? For example, `<script>alert('hello');</script>`. If this script were stored on the database then every future user’s page would have a pop-up saying “hello”. While this particular example is more annoying than dangerous, a site vulnerable to XSS is very dangerous because a malicious user could insert *any* JavaScript into the page, including JavaScript that steals pretty much anything from an unsuspecting user.

To prevent an XSS attack Django templates [automatically escape](#) specific characters that are potentially dangerous including brackets (< and >), single quotes ', double quotes ", and the ampersand &. There are some edge cases where you might want to turn [autoescape off](#) but this should be used with extreme caution.

One step we do want to take is to set `SECURE_BROWSER_XSS_FILTER` to `True` which will use the [X-XSS-Protection Header](#) to help guard against XSS attacks.

We can use the `ENVIRONMENT` variable now to add `if/else` logic at the bottom of our `bookstore_project/settings.py` file.

Code

```
# bookstore_project/settings.py

# production

if ENVIRONMENT == 'production':
    SECURE_BROWSER_XSS_FILTER = True # new
```

Spin down the container and start it up again to register the changes to our settings file. Running the `--deploy` check again shows we’re now down to 6 issues!

Command Line

```
$ docker-compose down  
$ docker-compose -f docker-compose-prod.yml up -d  
$ docker-compose exec web python manage.py check --deploy
```

Even with Django's protections in place always be careful when storing HTML in a database that will then be displayed to users. OWASP's [XSS Cheat Sheet](#) is recommended for further reading.

Cross-Site Request Forgery (CSRF)

A [Cross-Site Request Forgery \(CSRF\)](#) is the third major type of attack but generally lesser known than SQL Injection or XSS. Fundamentally it exploits that trust a site has in a user's web browser.

When a user logs in to a website, let's call it a banking website for illustration purposes, the server sends back a session token for that user. This is included in the HTTP Headers of all future requests and authenticates the user. But what happens if a malicious actor somehow obtains access to this session token?

For example, consider a user who logs into their bank in one browser tab. Then in another tab they open their email and click on an email link from a malicious actor. This link looks legitimate, but in fact it is pointing to the user's bank which they are still logged into! So instead of leaving a blog comment on this fake site, behind the scenes the user's credentials are used to transfer money from their account to the hacker's account.

In practice there are multiple ways to obtain a user's credentials via a CSRF attack, not just links, but hidden forms, special image tags, and even AJAX requests.

Django provides [CSRF protection](#) by including a random secret key both as a cookie via [CSRF Middleware](#) and in a form via the `csrf_token` template tag. A 3rd party website will not have access to a user's cookies and therefore any discrepancy between the two keys causes an error.

As ever, Django does allow customization: you can disable the CSRF middleware and use the `csrf_protect()` template tag on specific views. However, undertake this step with extreme caution.

The OWASP [CSRF Cheat Sheet](#) provides a comprehensive look at the issue. Almost all major websites have been victims of CSRF attacks at some point in time.

A good rule of thumb is whenever you have a form on your site, think about whether you need to include the `csrf_token` tag in it. Most of the time you will!

Clickjacking Protection

[Clickjacking](#) is yet another attack when a malicious site tricks a user into clicking on a hidden frame. An internal frame, known as an iframe, is commonly used to embed one website within another. For example, if you wanted to include a Google Map or YouTube video on your site you would include the `iframe` tag that puts that site within your own. This is very convenient.

But it has a security risk which is that a frame can be hidden from a user. Consider if a user is already logged into their Amazon account and then visits a malicious site that purports to be a picture of kittens. The user clicks on said malicious site to see more kittens, but in fact they click an iFrame of an Amazon item that is unknowingly purchased. This is but one example of clickjacking.

To prevent against this Django comes with a default [clickjacking middleware](#) that checks whether or not a resource can be loaded within a frame or iframe. You can

turn this protection off if desired or even set it at a per view level. As ever, do so with a degree of caution and [research](#).

For production though we will set it to `DENY` rather than the default of `SAMEORIGIN`. Note that strings must be placed around it so use '`DENY`' rather than simply `DENY`.

Code

```
# bookstore_project/settings.py

# production

if ENVIRONMENT == 'production':

    SECURE_BROWSER_XSS_FILTER = True

    X_FRAME_OPTIONS = 'DENY' # new
```

Spin down the server, rebuild it, and run the tests again.

Command Line

```
$ docker-compose down

$ docker-compose -f docker-compose-prod.yml up -d --build

$ docker-compose exec web python manage.py check --deploy
```

Now only 5 issues remaining!

HTTPS/SSL

All modern websites should use HTTPS which provides encrypted communication between a client and server. [HTTP \(Hypertext Transfer Protocol\)](#) is the backbone of the modern web, but it does not, by default, have encryption.

The “s” in HTTPS refers to its encrypted nature first due to SSL (Secure Sockets Layer) and these days its successor [TLS \(Transport Layer Security\)](#).

With HTTPS enabled, which we will do in our deployment chapter, malicious actors can't sniff the incoming and outgoing traffic for data like authentication credentials or API keys.

In our `settings.py` file we can force all non-HTTPS traffic to be redirected to HTTPS. Add the following line at the bottom of the file.

Code

```
# bookstore_project/settings.py

# production

if ENVIRONMENT == 'production':
    SECURE_BROWSER_XSS_FILTER = True
    X_FRAME_OPTIONS = 'DENY'
    SECURE_SSL_REDIRECT = True # new
```

HTTP Strict Transport Security (HSTS)

HTTP Strict Transport Security (HSTS) is a security policy that lets our server enforce that web browsers should only interact via HTTPS by adding a [Strict-Transport-Security header](#).

It's best to start with a small value of time for testing, such as 3600 seconds, one hour, and then later extending it to one year (314,536,000 seconds). This is done in the `SECURE_HSTS_SECONDS` config which is implicitly set to 0.

We don't have any subdomains in our Bookstore project so it makes sense to force any subdomains to also exclusively use SSL via the `SECURE_HSTS_INCLUDE_SUBDOMAINS` setting.

Also `SECURE_HSTS_PRELOAD` to `True`.

Finally also `SECURE_CONTENT_TYPE_NOSNIFF` which controls `nosniff` set to `True`

Code

```
# bookstore_project/settings.py

# production

if ENVIRONMENT == 'production':

    SECURE_BROWSER_XSS_FILTER = True

    X_FRAME_OPTIONS = 'DENY'

    SECURE_SSL_REDIRECT = True

    SECURE_HSTS_SECONDS = 3600 # new

    SECURE_HSTS_INCLUDE_SUBDOMAINS = True # new

    SECURE_HSTS_PRELOAD = True # new

    SECURE_CONTENT_TYPE_NOSNIFF = True # new
```

Secure Cookies

An [HTTP Cookie](#) is used to store information on a client's computer such as authentication credentials. This is necessary because the HTTP protocol is stateless by design: there's no way to tell if a user is authenticated other than including an identifier in the HTTP Header!

Django uses sessions and cookies for this, as do most websites. But cookies can and should be forced over HTTPS as well via the [SESSION_COOKIE_SECURE](#) config. It defaults to `False` so we must set it to `True` in production. We can also do the same for CSRF cookies using [CSRF_COOKIE_SECURE](#).

Code

```
# bookstore_project/settings.py

# production

if ENVIRONMENT == 'production':

    SECURE_BROWSER_XSS_FILTER = True

    X_FRAME_OPTIONS = 'DENY'

    SECURE_SSL_REDIRECT = True

    SECURE_HSTS_SECONDS = 3600

    SECURE_HSTS_INCLUDE_SUBDOMAINS = True

    SECURE_HSTS_PRELOAD = True

    SECURE_CONTENT_TYPE_NOSNIFF = True

    SESSION_COOKIE_SECURE = True # new

    CSRF_COOKIE_SECURE = True # new
```

Spin down and up the containers one last time and then run the --check to confirm there are no more errors!

Command Line

```
$ docker-compose down

$ docker-compose -f docker-compose-prod.yml up -d --build

$ docker-compose exec web python manage.py check --deploy

System check identified no issues (0 silenced).
```

Admin Hardening

So far it may seem as though the advice is to rely on Django defaults, use HTTPS, add csrf_token tags on forms, and set a permissions structure. All true. But one step Django does not take on our behalf is hardening the Django admin.

Consider that every Django website sets the admin, by default, to the `/admin` URL. This is a prime suspect for any hacker trying to access a Django site. Therefore an easy step is to simply change the admin URL to literally anything else!

To do this, open up the `bookstore_project/urls.py` file. In this example it's been set to `anything-but-admin/`.

Code

```
# bookstore_project/urls.py

from django.conf import settings
from django.conf.urls.static import static
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    # Django admin
    path('anything-but-admin/', admin.site.urls), # new

    # User management
    path('accounts/', include('allauth.urls')),

    # Local apps
    path('', include('pages.urls')),
    path('books/', include('books.urls')),
]

if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

A fun 3rd party package `django-admin-honeypot` will generate a fake admin log in

screen and email [site admins](#) the IP address of anyone trying to attack your site at `/admin`. These IP addresses can then be added to a blocked address list for the site.

It's also possible via [django-two-factor-auth](#) to add two-factor authentication to your admin for an even further layer of protection.

Git

This chapter has been particularly heavy on code changes so make sure to commit all the updates with Git.

Command Line

```
$ git status  
$ git add .  
$ git commit -m 'ch17'
```

If you have any errors, check your logs with `docker-compose logs` and compare your code with the [official source code on Github](#).

Conclusion

Security is a major concern for any website, especially one that handles payments like our Bookstore. By using a `docker-test-prod.yml` file we can accurately test, within Docker, our production settings before deploying the site live. Django comes with many built-in security features and with the addition of the deployment checklist we can now deploy our site now with a high degree of confidence that it is secure.

Ultimately security is constant battle and while the steps in this chapter cover most areas of concern, keeping your website up-to-date with the latest Django version is vital for continued safety.

Chapter 18: Deployment

So far we have been working entirely in a local development environment on our computer. But now it is time to deploy our project so that it is accessible to the public. In truth the topic of deployment is worth an entire book on its own. Compared to other web frameworks Django is very hands-off and agnostic on the topic. There are no one-click deploys for most hosting platforms and while this requires more developer work it also allows, in typical Django fashion, for a high degree of customization.

In the previous chapter we configured a completely separate `docker-compose-prod.yml` file and updated `bookstore_project/settings.py` to be production-ready. In this chapter we'll review how to choose a hosting provider, add a production-ready web server, and properly configure static/media files before deploying our Bookstore site!

PaaS vs IaaS

The first question is whether to use a Platform-as-a-Service (PaaS) or Infrastructure-as-a-Service (IaaS). A PaaS is an opinionated hosting option that handles much of the initial configuration and scaling needed for a website. Popular examples include [Heroku](#), [PythonAnywhere](#), and [Dokku](#) among many others. While a PaaS costs more money upfront than an IaaS it saves an incredible amount of developer time, handles security updates automatically, and can be quickly scaled.

An IaaS by contrast provides total flexibility is typically cheaper, but it requires a high degree of knowledge and effort to properly set up. Prominent IaaS options include [DigitalOcean](#), [Linode](#), [Amazon EC2](#), and [Google Compute Engine](#) among many others.

So which one to use? Django developers tend to fall in one of two camps: either they already have a deployment pipeline configured with their IaaS of choice or they use a PaaS. Since the former is far more complex and varies widely in its configuration, we will use a PaaS in this book, specifically Heroku.

The choice of Heroku is somewhat arbitrary, but it is a mature technology that comes with a truly free tier sufficient for deploying our Bookstore project.

WhiteNoise

For local development Django relies on the [staticfiles app](#) to automatically gather and serve static files from across the entire project. This is convenient, but quite inefficient and likely insecure, too.

For production the [collectstatic](#) must be run to compile all static files into a single directory specified by [STATIC_ROOT](#). They can then be served either on the same server, a separate server, or a dedicated cloud service/CDN by updating [STATICFILES_STORAGE](#).

While it is tempting to jump right to a dedicated CDN beware premature optimization: the default option of serving from your server's filesystem scales to a quite large size. If you decide to go this route the [django-storages](#) project is a popular approach.

In our project we will rely on serving files from our server with the aid of the [WhiteNoise](#) project which works extremely well on Heroku and is both faster and more configurable than Django defaults.

The first step is to install `whitenoise` within Docker and stop the running containers.

Command Line

```
$ docker-compose exec web pipenv install whitenoise==4.1.2  
$ docker-compose down
```

We won't rebuild the image just yet because we also have to make changes to our settings. Since we're using Docker it's possible to switch to WhiteNoise locally as well as in production. While it's possible to do this by passing in a `--nostatic` flag to the `runserver` command, this becomes tiring in practice. A better approach is to add `whitenoise.runserver_nostatic` before `django.contrib.staticfiles` in the `INSTALLED_APPS` config which will do the same thing. We'll also add it to our `MIDDLEWARE` right below `SecurityMiddleware`.

Code

```
# bookstore_project/settings.py  
  
INSTALLED_APPS = [  
  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'whitenoise.runserver_nostatic', # new  
    'django.contrib.staticfiles',  
    'django.contrib.sites',  
    ...  
]  
  
MIDDLEWARE = [  
    'django.middleware.cache.UpdateCacheMiddleware',  
    'django.middleware.security.SecurityMiddleware',
```

```
'whitenoise.middleware.WhiteNoiseMiddleware', # new  
...  
]
```

With all our changes made we can now start up our project again in local development mode.

Command Line

```
$ docker-compose up -d --build
```

WhiteNoise has additional options to serve compressed compressed content and far-future cache headers on content that won't change. But for now, go ahead and run the `collectstatic` command one more time.

Command Line

```
$ docker-compose exec web python manage.py collectstatic
```

There will be a warning about overwriting existing files. That's fine. Type "yes" and then hit the "Return" key to continue.

Gunicorn

When we ran the `startproject` command way back in Chapter 3 a `wsgi.py` file was created with a default [WSGI \(Web Server Gateway Interface\)](#) configuration. This is a specification for how a web app (like our Bookstore project) communicates with a web server.

For production it is common to swap this out for either [Gunicorn](#) or [uWSGI](#). Both offer a performance boost, but Gunicorn is more focused and simpler to implement so it will be our choice.

The first step is to install it within our project and stopping our containers.

Command Line

```
$ docker-compose exec web pipenv install gunicorn==19.9.0
$ docker-compose down
```

Because we are using Docker our local environment can mimic production quite easily so we'll update the `build` command in both `docker-compose.yml` and `docker-compose-prod.yml` to use Gunicorn instead of the local server.

docker-compose.yml

```
# docker-compose.yml

# command: python /code/manage.py runserver 0.0.0.0:8000
command: gunicorn bookstore_project.wsgi -b 0.0.0.0:8000 # new
```

docker-compose-prod.yml

```
# docker-compose.yml

# command: python /code/manage.py runserver 0.0.0.0:8000
command: gunicorn bookstore_project.wsgi -b 0.0.0.0:8000 # new
```

Now start up the containers again building a new image with the Gunicorn package and our updated environment variables.

Command Line

```
$ docker-compose up -d --build
```

dj-database-url

We will ultimately spin up a dedicated PostgreSQL database within Heroku for our deployment. The way database information is supplied to Heroku is via an environment variable named `DATABASE_URL`. We can use the [dj-database-url](#) package to parse

the DATABASE_URL environment variable and automatically convert it to the proper configuration format.

For the last time in this book, install the package within Docker and then stop the containers.

Command Line

```
$ docker-compose exec web pipenv install dj-database-url==0.5.0  
$ docker-compose down
```

Then add three lines to the bottom of the bookstore_project/settings.py file.

Code

```
# bookstore_project/settings.py  
  
# Heroku  
  
import dj_database_url  
  
db_from_env = dj_database_url.config(conn_max_age=500)  
  
DATABASES['default'].update(db_from_env)
```

And then build our new image, start the containers, and load the updated settings into our project.

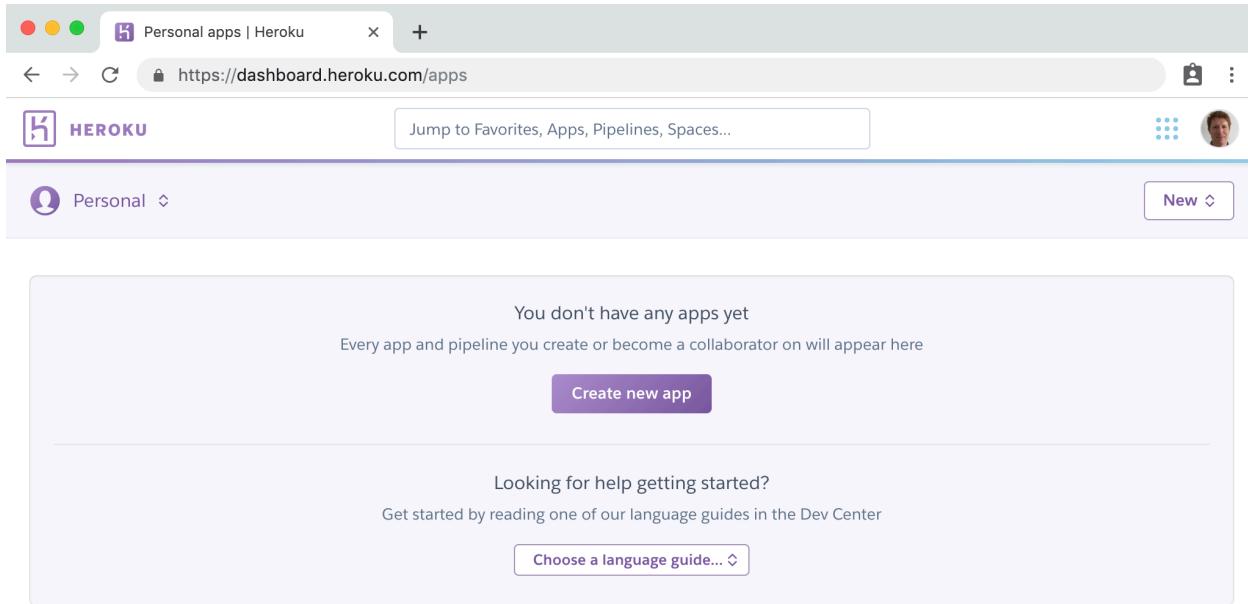
Command Line

```
$ docker-compose up -d --build
```

We're done with local changes and can now fully switch over to deploying with Heroku.

Heroku

Head over to the [Heroku](#) website and sign up for a free account. After you confirm your email Heroku will redirect you to the dashboard section of the site.



Heroku Dashboard

Next make sure to install Heroku's *Command Line Interface (CLI)* so we can deploy from the command line. There are detailed [instructions here](#).

The final step is to log in with your Heroku credentials via the command line by typing `heroku login`. Use the email and password for Heroku you just set.

Command Line

```
$ heroku login
```

All set! If you have any issues you can type `heroku help` on the command line or visit the Heroku site for additional information.

Deploying with Docker

Now we are presented with a choice: deploy the traditional way on Heroku or with Docker containers. The latter is a new approach Heroku and other hosting providers

have only recently added. However, just as Docker has taken over local development, it is starting to take over deployments as well. And once you've configured containers for deployment it is far easier to switch between potential hosting providers rather than if you configure their specific way. So we will deploy with Docker containers.

Even then we have, yet again, a choice to make as there are [two different container options available](#): using a container registry to deploy pre-built images or adding a `heroku.yml` file. We will use the latter approach as it will allow additional commands and more closely mimics the traditional Heroku approach of adding a `Procfile` for configuration.

heroku.yml

Traditional non-Docker Heroku relies on a custom `Procfile` for configuring a site for deployment. For containers Heroku relies on a similar approach of a custom file but called `heroku.yml` in the root directory. It is similar to `docker-compose.yml` which is used for building local Docker containers.

Let's create our `heroku.yml` file now.

Command Line

```
$ touch heroku.yml
```

There are four [top-level sections](#) available for configuration: `setup`, `build`, `release`, and `run`.

The main function of `setup` is to specify which add-ons are needed. These are hosted solutions Heroku provides, typically for a fee. The big one is our database which will rely on the free [heroku-postgresql](#) tier. Heroku takes care of provisioning it, security updates, and we can easily upgrade the database size and uptime as needed.

The `build` section is how we specify the `Dockerfile` should be, well, built. This relies on our current `Dockerfile` in the root directory.

The `release` phase is used to run tasks before each new release is deployed. For example, we can make sure `collectstatic` is run on every deploy automatically.

Finally there is the `run` phase where we specify which processes actually run the application. Notably, the use of `gunicorn` as the web server.

heroku.yml

```
setup:  
  addons:  
    - plan: heroku-postgresql  
  
build:  
  docker:  
    web: Dockerfile  
  
release:  
  image: web  
  command:  
    - python manage.py collectstatic --noinput  
  
run:  
  web: gunicorn bookstore_project.wsgi
```

Make sure to add the new deployment updates to Git and commit them. In the next section we'll push all our local code to Heroku itself.

Command Line

```
$ git status  
$ git add .  
$ git commit -m 'ch18'
```

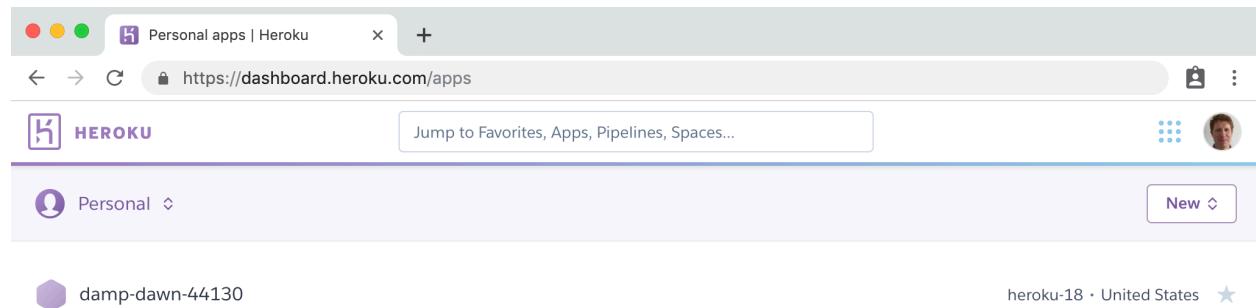
Heroku Deployment

Now create a new app on Heroku for our Bookstore project. If you type `heroku create` then Heroku will assign a random name. Since names are global in Heroku, it's unlikely that common ones like "blog" or "webapp" will be available. The name can always be changed later within Heroku to an available global namespace.

Command Line

```
$ heroku create  
Creating app... done, ⚡ damp-dawn-44130  
https://damp-dawn-44130.herokuapp.com/ |  
https://git.heroku.com/damp-dawn-44130.git
```

In this case Heroku assigned my app the name `damp-dawn-44130`. If you refresh the Heroku dashboard on the website you will now see the newly created app.



Heroku New App

Click on the new app, damp-dawn-44130 in my case, to open the “Overview” page.

The screenshot shows the Heroku Dashboard interface. At the top, there's a header bar with the Heroku logo, a search bar labeled 'Jump to Favorites, Apps, Pipelines, Spaces...', and a user profile icon. Below the header, the app name 'damp-dawn-44130' is displayed, along with 'Personal' and 'More' dropdown menus, an 'Open app' button, and a star icon. A navigation bar at the bottom of the header includes links for Overview, Resources, Deploy, Metrics, Activity, Access, and Settings, with 'Overview' being the active tab.

The main content area is divided into several sections:

- Installed add-ons:** \$0.00/month. A message states there are no add-ons for this app, with a link to 'Configure Add-ons'.
- Dyno formation:** \$0.00/month. A message states the app has no process types yet, with a link to 'Configure Dynos'.
- Latest activity:** All Activity (0). It shows two recent events:
 - william.s.vincent@gmail.com: Enable Logplex (Today at 10:04 PM · v2)
 - william.s.vincent@gmail.com: Initial release (Today at 10:04 PM · v1)
- Collaborator activity:** Manage Access (0). A message states there is no recent activity on this app, with a note that it will be shown when there are recent deploys.

Heroku Overview Page

Then click on the “Settings” option at the top as we want to set our production environment variables within Heroku.

The screenshot shows the Heroku dashboard interface. At the top, there's a header with the Heroku logo, a search bar labeled 'Jump to Favorites, Apps, Pipelines, Spaces...', and user profile icons. Below the header, the app name 'damp-dawn-44130' is displayed, along with navigation links for Overview, Resources, Deploy, Metrics, Activity, Access, and Settings. The 'Settings' link is currently selected. Under the 'Settings' section, there's a table with one row containing the 'Name' column (value: 'damp-dawn-44130') and an 'Edit' button. Further down, there's a 'Config Vars' section with a 'Reveal Config Vars' button.

Heroku App Settings

Click on “Reveal Config Vars”. Then add environment variables for `ENVIRONMENT` to “production,” the `SECRET_KEY`, and `DEBUG` equal to “0” from the `docker-compose-prod.yml` file.

Name **damp-dawn-44130** [Edit](#)

Config Vars

ENVIRONMENT	production	Edit	Delete
SECRET_KEY	p_o3vp1rg5)t^lxm9-43%0)s=1qp)	Edit	Delete
DEBUG	0	Edit	Delete
KEY	VALUE	Add	

Heroku Config Vars

It's also possible to add config variables from the command line to Heroku. Doing so via the Dashboard is easier to see, which is why it is demonstrated either way. Both approaches work.

Now set the **stack** to use our Docker containers, not Heroku's default buildpack. Include your app name here at the end of the command after `heroku stack:set container -a`.

Command Line

```
$ heroku stack:set container -a damp-dawn-44130  
Stack set. Next release on ⚡ damp-dawn-44130 will use container.  
Run git push heroku master to create a new release on ⚡ damp-dawn-44130.
```

To confirm this change executed correctly, refresh the Heroku dashboard page and note that under the “Info” section, for “Stack” it now features “container.” That’s what we want.

The screenshot shows the Heroku dashboard for the app 'damp-dawn-44130'. At the top, there's a navigation bar with tabs for Overview, Resources, Deploy, Metrics, Activity, Access, and Settings. The Settings tab is active. Below the navigation, there's a section for 'Config Vars' with a 'Reveal Config Vars' button. Under the 'Info' section, there's a table with the following data:

Region	United States
Stack	container
Framework	No framework detected
Slug Size	No slug detected
Heroku Git URL	https://git.heroku.com/damp-dawn-44130.git

A red box highlights the 'Stack' row in the table.

Heroku Stack

Before pushing our code to Heroku specify the hosted PostgreSQL database we want. In our case, the free hobby-dev tier works well; it can always be updated in the future.

Command Line

```
$ heroku addons:create heroku-postgresql:hobby-dev -a damp-dawn-44130
Creating heroku-postgresql:hobby-dev on ⚡ damp-dawn-44130... free
Database has been created and is available
  ! This database is empty. If upgrading, you can transfer
    ! data from another database with pg:copy
Created postgresql-opaque-38157 as DATABASE_URL
Use heroku addons:docs heroku-postgresql to view documentation
```

The `dj-database-url` setting we set previously will automatically find and use this `DATABASE_URL` for us.

We're ready! Create a [Heroku remote](#), which means a version of our code that will live on a Heroku-hosted server. Make sure to include `-a` and the name of your app. Then “push” the code to Heroku which will result in building our Docker image and running the containers.

Command Line

```
$ heroku git:remote -a damp-dawn-44130
$ git push heroku master
```

The initial push might take a while to complete. You can see active progress by clicking on the “Activity” tab on the Heroku dashboard.

Our Bookstore project should now be available online. Remember that while the code mirrors our own local code, the production site has its own database that has no information in it. To run commands on it add `heroku run` to standard commands. For example, we should `migrate` our initial database and then create a superuser account.

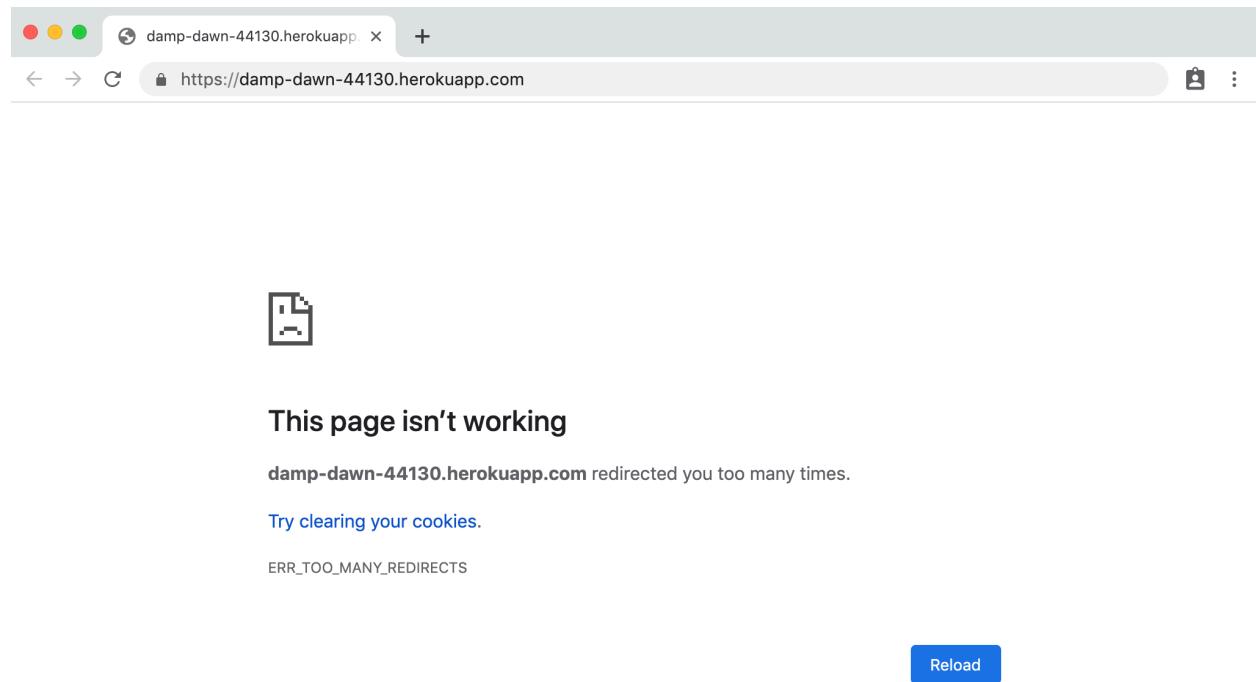
Command Line

```
$ heroku run python manage.py migrate  
$ heroku run python manage.py createsuperuser
```

There are two ways to open the newly-deployed application. From the command line you can type `heroku open -a` and the name of your app. Or you can click on the “Open app” button in the upper right corner of the Heroku dashboard.

Command Line

```
$ heroku open -a damp-dawn-44130
```



Heroku Redirects

But...ack! What's this? A redirect error. Welcome to the joys of deployment where issues like this will crop up all the time.

SECURE_PROXY_SSL_HEADER

Some sleuthing uncovers that the issue is related to our `SECURE_SSL_REDIRECT` setting. Heroku uses proxies and so we must find the proper header and update `SECURE_PROXY_SSL_HEADER` accordingly.

Since we *do* trust Heroku we can add Django's default suggestion. So update the "production" section of `bookstore_project/settings.py` with the following line.

Code

```
# bookstore_project/settings.py

# production

if ENVIRONMENT == 'production':
    SECURE_BROWSER_XSS_FILTER = True
    X_FRAME_OPTIONS = 'DENY'
    SECURE_SSL_REDIRECT = True
    SECURE_HSTS_SECONDS = 3600
    SECURE_HSTS_INCLUDE_SUBDOMAINS = True
    SECURE_HSTS_PRELOAD = True
    SECURE_CONTENT_TYPE_NOSNIFF = True
    SESSION_COOKIE_SECURE = True
    CSRF_COOKIE_SECURE = True
    SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https') # new
```

While we're at it, we can also update `ALLOWED_HOSTS` with the exact URL name of our Heroku deployment. Mine is `damp-dawn-44130.herokuapp.com/` so the updated configuration looks as follows:

Code

```
# bookstore_project/settings.py  
ALLOWED_HOSTS = ['damp-dawn-44130.herokuapp.com', 'localhost', '127.0.0.1']
```

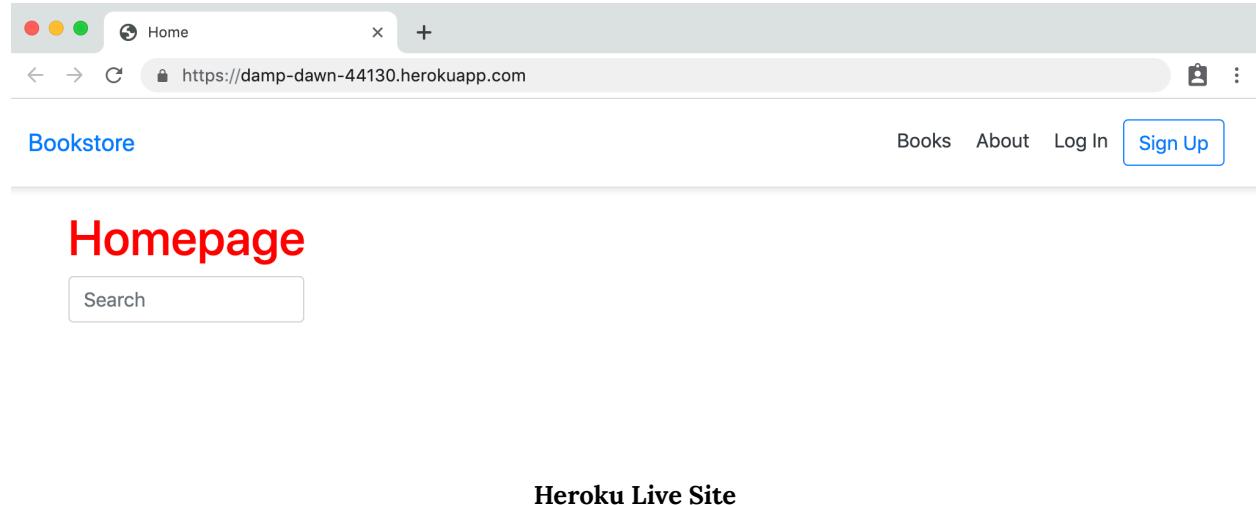
Use your own unique Heroku subdomain here!

Finally, commit these changes to Git and then push the updated code to Heroku.

Command Line

```
$ git status  
$ git add .  
$ git commit -m 'secure_proxy_ssl_header and allowed_hosts update'  
$ git push heroku master
```

After the build has completed refresh the webpage for your site. There it is!



Heroku Logs

It is inevitable that you will have errors in your deployment at some point. When you do, run `heroku logs --tail` to see error and info logs and debug what's going on.

Hopefully this deployment process was smooth. But in practice, even with an established Platform-as-a-Service like Heroku, it is highly likely that issues will occur. If you see an error page, type `heroku logs --tail`, which displays info and error logs, to diagnose the issue.

Stripe Live Payments

Our orders page is still running with Stripe in test mode. How do we update this to use live keys and make actual payments? First, make sure to fully activate your account on the Stripe website by filling in additional personal and banking information about yourself. Second, add the live publishable and secret keys to your `docker-compose-prod.yml` file—`docker-compose.yml` is for local development only.

`docker-compose-prod.yml`

```
version: '3.7'

services:
  web:
    build: .
    command: gunicorn bookstore_project.wsgi -b 0.0.0.0:8000
    environment:
      - ENVIRONMENT=production
      - SECRET_KEY=p_o3vp1rg5)t^lxm9-43%0)s-=1qpeq%o7gfq+e4#!t+_ev82
      - DEBUG=0
      - STRIPE_LIVE_PUBLISHABLE_KEY=<pk_live_your_publishable_key_here>
      - STRIPE_LIVE_SECRET_KEY=<sk_live_your_secret_key_here>
    ports:
      - 8000:8000
    depends_on:
```

```
- db  
db:  
  image: postgres:11
```

And third, add these two new environment variables to `bookstore_project/settings.py` under the existing entries for Stripe test values.

Code

```
# bookstore_project/settings.py  
STRIPE_LIVE_PUBLISHABLE_KEY=os.environ.get('STRIPE_LIVE_PUBLISHABLE_KEY')  
STRIPE_LIVE_SECRET_KEY=os.environ.get('STRIPE_LIVE_SECRET_KEY')
```

To confirm these production settings work as expected make sure to stop any running local Docker containers, then restart with `docker-compose-prod.yml` and try it out.

Command Line

```
$ docker-compose down  
$ docker-compose -f docker-compose-prod.yml up -d --build
```

Be aware that this is a real payment! While it is going to your own banking account Stripe will still deduct its standard 2.9% + 30 cents per transaction so test this sparingly.

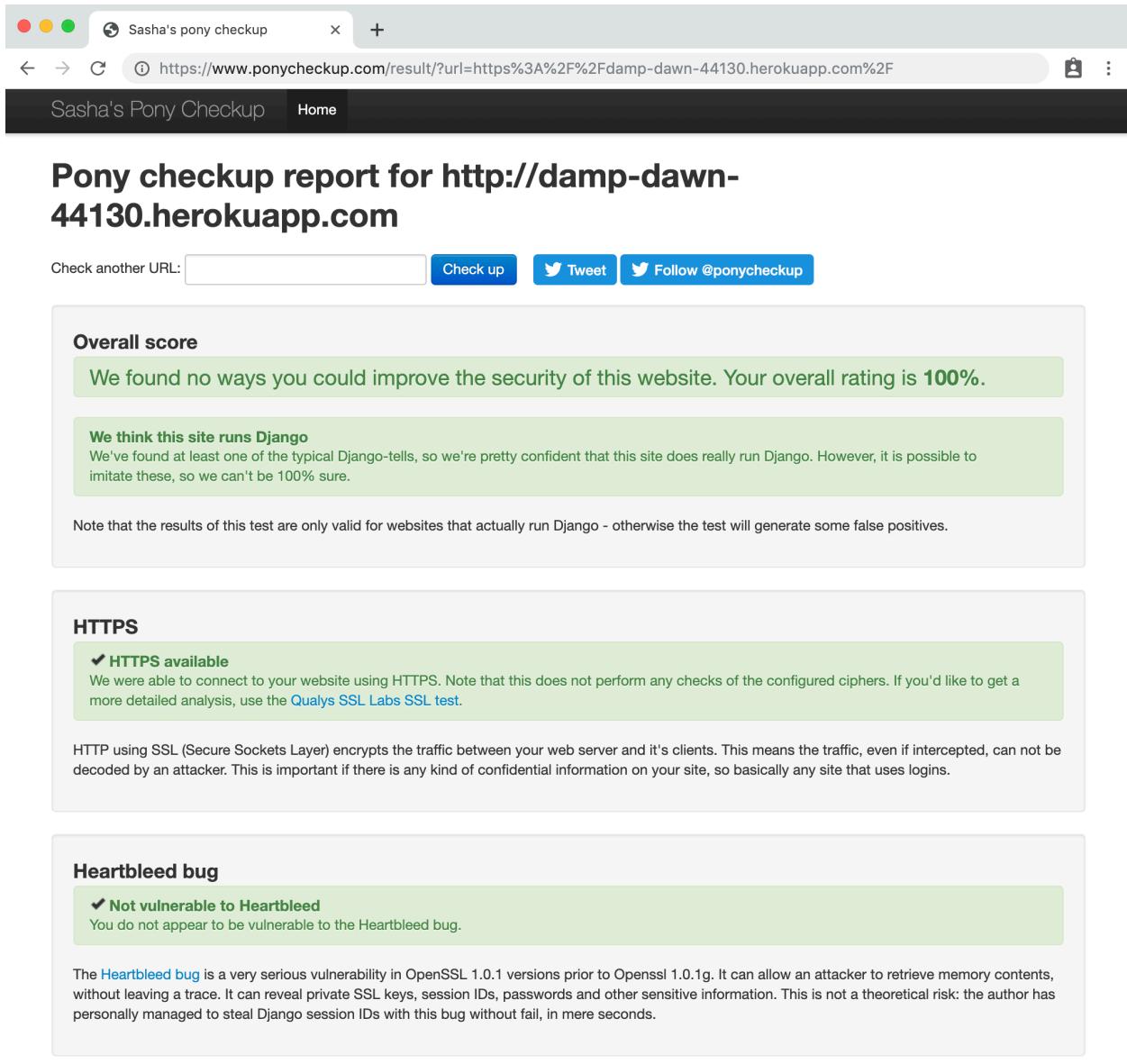
Heroku Add-ons

Heroku comes with a large list of [add-on services](#) that, for a fee, can be quickly added to any site. For example, to enable caching with Memcache, [Memcachier](#) is an option to consider.

[Daily backups](#) are an additional, but essential, feature of any production database.

PonyCheckup

A popular way to test Django deployments is with [Pony Checkup](#) by Sasha Romijn, a long-standing member of the Django Security team.



The screenshot shows a browser window displaying the results of a PonyCheckup scan for a Django application running on Heroku. The title bar says "Sasha's pony checkup". The address bar shows the URL: https://www.ponycheckup.com/result/?url=https%3A%2F%2Fdamp-dawn-44130.herokuapp.com%2F. The main content area has a dark header "Sasha's Pony Checkup" and "Home". Below it, the main heading is "Pony checkup report for http://damp-dawn-44130.herokuapp.com". A search bar says "Check another URL:" followed by a "Check up" button, a "Tweet" button, and a "Follow @ponycheckup" button. The first section, "Overall score", shows a green box stating "We found no ways you could improve the security of this website. Your overall rating is 100%." The next section, "We think this site runs Django", includes a note that the site might run Django due to typical Django-tells, but it's not 100% certain. A note below states that results are valid for actual Django sites only. The "HTTPS" section shows a green box with a checkmark and the text "HTTPS available", noting that SSL encrypts traffic between server and client. It also links to Qualys SSL Labs for more detailed analysis. The "Heartbleed bug" section shows a green box with a checkmark and the text "Not vulnerable to Heartbleed", stating that the site is not vulnerable. It also links to the Heartbleed bug for more information. At the bottom center is the "Pony Checkup" logo.

If you have any errors, please check the [official source code on Github](#).

Conclusion

Even with all the advantages of a modern Platform-as-a-Service like Heroku, deployment remains a complicated and often frustrating task for many developers. Personally, I want my web apps to “just work”. But many engineers come to enjoy the challenges of working on performance, security, and scaling. After all, it is far easier to measure improvements in this realm: did page load times decrease? Did site uptime improve? Is security up-to-date? Working on these problems can often feel far more rewarding than debating which new feature to add to the site itself.

Conclusion

Building a “professional” website is no small task even with all the help that a batteries-included web framework like Django provides. Docker provides a major advantage in standardizing both local and production environments regardless of local machine—and especially in a team context. However Docker is a complicated beast on its own. While we have used it judiciously in this book there is much more that it can do depending on the needs of a project.

Django itself is friendly to small projects because its defaults emphasize rapid local development but these settings must be systematically updated for production, from upgrading the database to PostgreSQL, using a custom user model, environment variables, configuring user registration flow, static assets, email...on and on it goes.

The good news is that the steps needed for a production-level approach are quite similar. Hence the first half of this book is deliberately agnostic about the eventual project that is built: you’ll find these steps are standard on almost any new Django project. The second half focused on building a real Bookstore site with modern best practices, added Reviews, image uploads, set permissions, configured payments with Stripe, added search, reviewed performance and security measures, and finally deployed on Heroku with containers.

For all the content covered in this book we’ve really only scratched the surface of what Django can do. This is the nature of modern web development: constant iteration.

Django is a magnificent partner in building out a professional website because so many of the considerations required have already been thought of and included. But knowledge is needed to know how to turn these production switches on to take full advantage of the customization Django allows. Ultimately that is the goal of this

book: to expose you, the reader, to the full spectrum of what Django and professional websites require.

As you learn more about web development and Django I'd urge caution when it comes to premature optimization. It is always tempting to features and optimizations to your project that you think you'll need later. The short list includes adding a CDN for static and media assets, judiciously analyzing database queries, adding indexes to models, and so on.

The truth is that in any given web project there will always be more to do than time allows. This book has covered the fundamentals that are worthy of upfront time to get right. Additional steps around security, performance, and features will present themselves to you in real-time. Try to resist the urge to add complexity until absolutely necessary.

If you have feedback on this book or examples of what you've built as a result, I read and respond to every email I receive at will@wsvincent.com. I look forward to hearing from you!