



# DJANGO

---

*for*

# BEGINNERS

Build websites with Python & Django

WILLIAM S. VINCENT

# **Django for Beginners**

## **Build websites with Python and Django**

**William S. Vincent**

This book is for sale at  
<http://leanpub.com/djangoforbeginners>

This version was published on 2023-09-28



\* \* \* \* \*

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

\* \* \* \* \*

© 2018 - 2023 William S. Vincent

# Table of Contents

## Chapter 0: Introduction

Prerequisites

Why Django

Why This Book

What's New in Django 4.2

Book Structure

Book Layout

Advice on Getting Stuck

Community

Conclusion

## Chapter 1: Initial Set Up

The Command Line

Shell Commands

Install Python 3 on Windows

Install Python 3 on Mac

Python Interactive Mode

Virtual Environments

PyPI (Python Package Index)

Install Django

First Django Project

The Development Server

Text Editors

VSCode Configurations

Install Git

Conclusion

## Chapter 2: Hello World App

How Websites Work

How Web Frameworks Work

Model-View-Controller vs Model-View-Template

Initial Set Up

Create An App

[Your First View](#)

[URLconfs](#)

[Git](#)

[GitHub](#)

[SSH Keys](#)

[Conclusion](#)

## [Chapter 3: Pages App](#)

[Initial Set Up](#)

[Templates](#)

[Class-Based Views](#)

[URLs](#)

[About Page](#)

[Extending Templates](#)

[Tests](#)

[Git and GitHub](#)

[Hosting Options](#)

[Deployment Checklist](#)

[Fly.io](#)

[Deploy to Fly.io](#)

[Conclusion](#)

## [Chapter 4: Message Board App](#)

[Initial Set Up](#)

[Create a Database Model](#)

[Activating models](#)

[Django Admin](#)

[Views/Templates/URLs](#)

[Adding New Posts](#)

[Static Files](#)

[Tests](#)

[GitHub](#)

[Conclusion](#)

## [Chapter 5: Message Board Deployment](#)

[Previous Deployment Checklist](#)

[Environment Variables](#)

## DATABASES

Pyscopg

CSRF TRUSTED ORIGINS

Updated Deployment Checklist

Web Servers

WSGI

ASGI

WhiteNoise

Middleware

requirements.txt

Final Deployment Checklist

Fly Deployment

Production Database

Conclusion

## Chapter 6: Blog App

Initial Set Up

Databases

Django's ORM

Blog Post Models

Primary Keys and Foreign Keys

Admin

URLs

Views

Templates

Static Files

Individual Blog Pages

get\_absolute\_url()

Tests

Git

Conclusion

## Chapter 7: Forms

CreateView

UpdateView

DeleteView

[Tests](#)

[Conclusion](#)

## [Chapter 8: User Accounts](#)

[Log In](#)

[Updated Homepage](#)

[Log Out Link](#)

[Sign Up](#)

[Sign Up Link](#)

[Static Files](#)

[GitHub](#)

[Conclusion](#)

## [Chapter 9: Blog Deployment](#)

[Gunicorn, Psycopg, and environs](#)

[DATABASES](#)

[WhiteNoise](#)

[requirements.txt](#)

[.dockerignore and .env](#)

[DEBUG](#)

[SECRET KEY](#)

[ALLOWED\\_HOSTS and CSRF\\_TRUSTED\\_ORIGINS](#)

[Fly Launch](#)

[Updated ALLOWED\\_HOSTS and](#)

[CSRF\\_TRUSTED\\_ORIGINS](#)

[Fly Deployment](#)

[Conclusion](#)

## [Chapter 10: Custom User Model](#)

[Initial Set Up](#)

[Git](#)

[Custom User Model](#)

[Forms](#)

[Superuser](#)

[Tests](#)

[Git](#)

[Conclusion](#)

## Chapter 11: User Authentication

[Templates](#)

[URLs](#)

[Admin](#)

[Tests](#)

[Git](#)

[Conclusion](#)

## Chapter 12: Bootstrap

[Pages App](#)

[Tests](#)

[Testing Philosophy](#)

[Bootstrap](#)

[Signup Form](#)

[Git](#)

[Conclusion](#)

## Chapter 13: Password Change and Reset

[Password Change](#)

[Customizing Password Change](#)

[Password Reset](#)

[Custom Templates](#)

[Try It Out](#)

[Git](#)

[Conclusion](#)

## Chapter 14: Newspaper App

[Articles App](#)

[URLs and Views](#)

[Detail/Edit/Delete](#)

[Create Page](#)

[Additional Links](#)

[Git](#)

[Conclusion](#)

## Chapter 15: Permissions and Authorization

[Improved CreateView](#)

[Authorizations](#)  
[Mixins](#)  
[LoginRequiredMixin](#)  
[UpdateView and DeleteView](#)  
[Template Logic](#)  
[Git](#)  
[Conclusion](#)

## [Chapter 16: Comments](#)

[Model](#)  
[Admin](#)  
[Template](#)  
[Comment Form](#)  
[Comment View](#)  
[Comment Template](#)  
[Comment Post View](#)  
[New Comment Link](#)  
[Git](#)  
[Conclusion](#)

## [Chapter 17: Deployment](#)

[Static Files](#)  
[Deployment Checklist](#)  
[Gunicorn, Psycopg, and environs](#)  
[DATABASES](#)  
[WhiteNoise](#)  
[requirements.txt](#)  
[.dockerignore and .env](#)  
[DEBUG and SECRET KEY](#)  
[ALLOWED HOSTS and CSRF TRUSTED ORIGINS](#)  
[Fly Deployment](#)  
[Production Database](#)  
[Deployment Conclusion](#)

## [Chapter 18: Conclusion](#)

[Next Steps](#)  
[3rd Party Packages](#)

[Learning Resources](#)  
[Python Books](#)  
[Feedback](#)

# Chapter 0: Introduction

Welcome to *Django for Beginners*, a project-based approach to learning web development with the [Django](#) web framework. In this book, you will build and deploy five progressively more complex web applications, starting with a simple *Hello, World* app, progressing to a *Pages* app, a *Message Board* app, a *Blog* app with forms and user accounts, and finally, a *Newspaper* app that uses a custom user model, email integration, foreign keys, authorization, permissions, and more. By the end of this book, you should feel confident creating Django projects from scratch using current best practices.

Django is a free, open-source web framework written in Python. First released in 2005, Django has been in continuous development since then and today powers many of the largest websites in the world, including Instagram, Pinterest, Bitbucket, and Disqus. At the same time, it is flexible enough to be a popular choice for early-stage startups and side projects.

## Prerequisites

*Django for Beginners* is written for Django 4.2 and Python 3.11. All code examples work with these versions. By the time you read this, newer versions of both Django and Python may be available. In general, you should always strive to be on the latest version of Django and Python. As both are mature technologies, any issues arising in the future will be relatively minor.

You don't need previous Python or web development experience to complete this book. Even someone new to

programming and web development can follow along and feel the magic of writing web applications from scratch. This book is also suitable for programmers with previous web development experience but new to Django. There are references throughout the book whenever Django differs from other web frameworks; the most obvious example is that Django adopts an MVT (Model-View-Template) approach slightly different from the dominant MVC (Model-View-Controller) pattern. We will cover these differences thoroughly once we start writing code.

## Why Django

A “web framework” is a collection of tools that abstract away much of the difficulty-and repetition-inherent in web development. For example, most websites need the same basic functionality: the ability to connect to a database, set URL routes, display content on a page, handle security properly, and so on. Rather than recreate all of this from scratch, programmers over the years have created web frameworks in all the major programming languages: *Django* in Python, *Rails* in Ruby, and *Laravel* in PHP, among many, many others.

Django inherited Python’s “batteries-included” approach and includes out-of-the-box support for routine tasks in web development, including:

- User authentication
- Testing
- Database models, forms, URL routes, and templates
- Admin interface
- Security and performance upgrades
- Support for multiple database backends

This approach allows web developers to focus on what makes a web application unique rather than reinventing the wheel every time.

In contrast, some web frameworks like [Flask](#) adopt a *microframework* approach of providing only the bare minimum required for a simple webpage. Flask is far more lightweight than Django and allows maximum flexibility; however, this comes at a cost to the developer. Building a simple website requires adding a dozen or more third-party packages, which may or may not be up-to-date, secure, or reliable. The lack of guardrails also means Flask's project structure varies widely, which makes it difficult to maintain best practices when moving between different projects.

Django remains [under active development](#) with a regular release schedule of monthly security/bug fixes and a major new release every eight months. Millions of programmers have already used Django to build their websites. It doesn't make sense to repeat the same code-and mistakes-when a large community of brilliant developers has already solved these problems for us.

And best of all, Django is written in the wonderfully readable yet still powerful Python programming language. In short, if you're building a website from scratch, Django is a fantastic choice.

## Why This Book

I wrote this book because while Django has incredible documentation, there need to be more beginner-friendly tutorials. The documentation assumes you already know what you want to do and provides relatively terse explanations aimed at intermediate-to-advanced developers; tutorials are typically more geared towards

beginners and hold your hand through all the steps required. When I first learned Django years ago, I struggled to complete the [official polls tutorial](#). I remember thinking, Why was this so hard?

With more experience, I now recognize that the writers of the Django docs faced a difficult choice: they could emphasize Django's ease of use or depth, but not both. They choose the latter, and as a professional developer, I appreciate the decision, but as a beginner, I found it so frustrating! My goal with this book is to fill in the gaps and showcase how beginner-friendly Django can be.

## What's New in Django 4.2

The most recent version of Django, 4.2, was released in April 2023. It has official support for Python 3.8, 3.9, 3.10, and 3.11. The official Django website contains information on all prior releases and [detailed 4.2 release notes](#).

One of the significant efforts for Django since 3.0 has been adding asynchronous support. Django 3.0 added ASGI (Asynchronous Server Gateway Interface), while Django 3.1 included asynchronous views, middleware, tests, and test client. Django 4.0 began making cache backends async-compatible, and Django 4.1 added both asynchronous handlers for class-based views and an asynchronous ORM interface. Django 4.2 supports Psycopg 3, a PostgreSQL database adapter for Python that includes async support. These iterative improvements are bringing Django closer and closer to full asynchronous functionality.

Django 4.2 is an LTS (Long Term Support) release meaning it will receive security updates for at least three years after its release. The previous LTS was Django 3.2, which ends support in April 2024. As with all major releases, there are

many new improvements for the 4.2 release, including a light or dark color theme for the admin.

Django is a mature web framework that strives to remain stable yet advance alongside the modern web. If you find yourself on a project with an older version of Django, there are [detailed instructions](#) for updating to the latest version.

## Book Structure

The book begins by demonstrating how to configure a local development environment for Windows and macOS in **Chapter 1**. We learn about the powerful command line, Git, text editors, and how to install the latest versions of Python and Django.

In **Chapter 2**, we build our first project, a minimal *Hello, World* app demonstrating how to set up new Django projects. Because establishing good software practices is vital, we'll save our work with Git and upload a copy to a remote code repository on GitHub.

In **Chapter 3**, we make, test, and deploy a *Pages* app that introduces templates and class-based views. Templates are how Django allows for DRY (Don't Repeat Yourself) development with HTML and CSS, while class-based views are quite powerful yet require minimal code. We also add our first tests and deploy the website to Fly.io. Using platform-as-a-service providers like Fly transforms deployment from a painful, time-consuming process into something that takes just a few mouse clicks.

We build our first database-backed project in **Chapter 4**, a *Message Board* app, and then deploy it in **Chapter 5**. Django provides a powerful ORM (Object-Relational Mapper) that abstracts away the need to write raw SQL ourselves.

Instead, we can write Python in a `models.py` file that the ORM automatically translates into the correct SQL for multiple database backends (PostgreSQL, MySQL, SQLite, MariaDB, and Oracle). We'll explore the built-in admin app, which provides a graphical way to interact with data. Of course, we also write tests for all our code, store a remote copy on GitHub, and deploy our website using a production database.

In **Chapters 6-9**, we're ready for a *Blog* app that implements CRUD (Create-Read-Update-Delete) functionality. Using Django's generic class-based views, we only have to write a small amount of actual code for this. Then we'll add forms and integrate Django's built-in user authentication system for signup, login, and logout functionality. Our deployment checklist will also grow longer as we add enhanced security to the process.

The remainder of the book is dedicated to building and deploying a robust *Newspaper* site. **Chapter 10** demonstrates how to use a custom user model, a Django best practice rarely addressed in tutorials. **Chapter 11** covers user authentication: login, logout, and signup. **Chapter 12** adds Bootstrap for enhanced CSS styling, and **Chapter 13** implements password reset and change via email. In **Chapters 14-16**, we add articles, comments, proper permissions, and authorizations. Finally, in **Chapter 17** production-ready deployment is covered.

The **Conclusion** provides an overview of the central concepts introduced in the book and a list of recommended resources for further learning.

While it may be tempting to skip around in this book, I highly recommend reading the chapters in order. Each

chapter introduces a new concept and builds upon past teachings.

By the end of this book, you'll have a solid understanding of Django, the ability to build your apps, and the background required to fully take advantage of additional resources for learning intermediate and advanced Django techniques.

## Book Layout

There are many code examples in this book styled as follows:

Code

---

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

---

For brevity, we will use three dots, ..., when the existing code has not changed. The section of code that *has* changed is highlighted using a # new comment.

Code

---

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

---

## Advice on Getting Stuck

Getting stuck on an issue happens to every programmer at every level. The only thing that changes as you become more experienced in your career is the difficulty of tackling the question. Part of learning how to be a better developer is accepting this frustration, finding help, asking targeted questions, and determining when the best course of action is to step away from the computer and walk around the block to clear your head.

The good news is that whatever error you are having, it is likely that you are not the first! Copy and paste your error into a search engine like Google or DuckDuckGo; it will typically bring up something from StackOverflow or a personal blog detailing the same issue. Experienced programmers often joke that their ability to Google more quickly towards an answer is the only thing that separates them from junior programmers. There is some truth to this.

You can only trust some of what you read online, of course, and with experience, you will develop the context to see how the pieces of Django and code fit together.

What do you do if you are stuck on something in this book? First, carefully check your code against what is in the book. If you're still stuck, you can look at the official source code, which is [available on GitHub](#). A common error is subtle white spacing differences that are almost impossible to detect to the naked eye. You can try copying and pasting the official source code if you suspect this might be the issue.

The next step is to walk away from the computer or even sleep on the problem. It's incredible what a small amount of rest and distance will do to your mind when solving problems.

There are two fantastic online resources where the Django community gathers to ask and answer questions. The first is the [official Django Forum](#), and the second is the [Django Users Google Group](#). Each is an excellent next step if you need additional help.

## Community

The success of Django owes as much to its community as it does the technological achievement of the framework itself.

“Come for the framework, stay for the community” is a common saying among Django developers. It extends to the technical development of Django, which happens online via the [django-developers](#), the non-profit [Django Software Foundation](#) that oversees Django, annual DjangoCon conferences, and local meetups where developers gather to share knowledge and insights.

Regardless of your level of technical expertise, becoming involved in Django is a great way to learn, meet other developers, and enhance your reputation.

## Conclusion

In the next chapter, you’ll learn how to properly set up your computer and create your first Django project. Let’s begin!

# Chapter 1: Initial Set Up

This chapter focuses on configuring your Windows or macOS computer to work on Django projects. You are probably eager to dive right in, but setting up your computer now is a one-time pain that will pay dividends.

Before installing Django, we must look at the *Command Line*, a powerful text-only interface developers use extensively. Next, we will install the latest version of Python, learn how to create dedicated virtual environments, and install Django. As a final step, we will explore using Git for version control and working with a text editor. By the end of this chapter, you will have created your first Django project from scratch and be able to create and modify new Django projects with just a few keystrokes.

## The Command Line

The command line is a text-only interface that harkens back to the original days of computing. If you have ever seen a television show or movie where a hacker is furiously typing into a black window: that's the command line. It is an alternative to the mouse or finger-based graphical user interface familiar to most computer users. Regular computer users will never need to use the command line. Still, for software developers, it is a vital and regularly-used tool necessary to execute programs, install software, use Git for version control, and connect to servers in the cloud. With practice, most developers find that the command line is a faster and more powerful way to navigate and control a computer.

Given its minimal user interface—just a blank screen and a blinking cursor—the command line is intimidating to newcomers. There is often no feedback after a command has run, and it is possible to wipe the contents of an entire computer with a single command if you’re not careful: no warning will pop up! As a result, use the command line with caution. Do not blindly copy and paste commands you find online blindly; rely only on trusted resources.

In everyday usage, multiple terms are used to refer to the command line: Command Line Interface (CLI), console, terminal, shell, or prompt. Technically speaking, the *terminal* is the program that opens up a new window to access the command line, a *console* is a text-based application, and the *shell* is the program that runs commands on the underlying operating system. The *prompt* is where commands are typed and run. It is easy to be confused by these terms initially, but they all essentially mean the same thing: the command line is where we run and execute text-only commands on our computer.

The built-in terminal and shell on Windows are both called *PowerShell*. To access it, locate the taskbar next to the Windows button on the bottom of the screen and type in “PowerShell” to launch the app. It will open a new window with a dark blue background and a blinking cursor after the > prompt. Here is how it looks on my computer.

---

Shell

---

PS C:\Users\wsv>

---

Before the prompt is PS, which refers to PowerShell, the initial c directory of the Windows operating system, followed by the Users directory and the current user, which on my personal computer, is wsv. Your username will be different. Don’t worry about what comes to the left of the > prompt at

this point: it varies depending on each computer and can be customized later. The shorter prompt of > will be used going forward for Windows.

On macOS, the built-in terminal is called, appropriately enough, *Terminal*. Open it via the Spotlight app: simultaneously press the Command and space bar keys and then type in “terminal.” Alternatively, open a new Finder window, navigate to the *Applications* directory, scroll down to open the *Utilities* directory, and double-click the Terminal application, which opens a new screen with a white background by default and a blinking cursor after the % prompt. Don’t worry about what comes *to the left* of the % prompt. It varies by computer and can be customized later on.

---

### Shell

---

Wills-Macbook-Pro:~ wsv%

---

The default shell for macOS since 2019 is [zsh](#), which uses the % prompt. If you see \$ as your prompt, then you are using the previous default macOS shell, [Bash](#). While most of the commands in this book will work interchangeably, it is recommended to look online at how to change to zsh via System Preferences if your computer still uses Bash.

**Note:** In this book, we will use the universal \$ Unix prompt for all shell commands rather than alternating between > on Windows and % on macOS.

## Shell Commands

There are many available shell commands, but developers generally rely on half a dozen commands for day-to-day use and look up more complicated commands as needed.

In most cases, Windows (PowerShell) and macOS commands are similar. For example, the command `whoami` returns the computer name/username on Windows and the username on macOS. As with all shell commands, type the command itself followed by the `return` key. Note that the `#` symbol represents a comment; these are not executed on the command line.

---

#### Shell

---

```
# Windows  
$ whoami  
wsv2023/wsv
```

```
# macOS  
$ whoami  
wsv
```

---

Sometimes, however, the shell commands on Windows and macOS are entirely different. A good example is the command for outputting a basic “Hello, World” message to the console. On Windows, the command is `Write-Host`, while on macOS, it is `echo`.

---

#### Shell

---

```
# Windows  
$ Write-Host "Hello, World"  
Hello, World
```

```
# macOS  
$ echo "Hello, World"  
Hello, World
```

---

A frequent task on the command line is navigating within the computer filesystem. On Windows and macOS, the command `pwd` (print working directory) outputs the current location within the file system.

---

#### Shell

---

```
# Windows  
$ pwd
```

```
Path  
----
```

```
C:\Users\wsv
```

```
# macOS  
$ pwd  
/Users/wsv
```

---

You can save your Django code anywhere, but we will place our code in the desktop directory for convenience. The command `cd` (change directory) followed by the intended location works on both systems.

#### Shell

---

```
# Windows  
$ cd onedrive\desktop  
$ pwd  
  
Path  
----  
C:\Users\wsv\onedrive\desktop  
  
# macOS  
$ cd desktop  
$ pwd  
/Users/wsv/desktop
```

---

**Tip:** The `tab` key will autocomplete a command, so if you type `cd d` and then hit `tab`, it will automatically fill in the rest of the name. If more than two directories start with `d`, hit the `tab` key again to cycle through them.

To make a new directory, use the command `mkdir` followed by the name. We will create one called `code` on the desktop and a new directory within it, `ch1-setup`.

#### Shell

---

```
# Windows  
$ mkdir code  
$ cd code  
$ mkdir ch1-setup  
$ cd ch1-setup  
  
# macOS  
$ mkdir code  
$ cd code  
$ mkdir ch1-setup  
$ cd ch1-setup
```

---

You can check that it has been created by looking on your desktop or running the command `pwd`.

---

### Shell

---

```
# Windows  
$ pwd  
  
Path  
----  
C:\Users\wsv\onedrive\Desktop\code\ch1-setup  
  
# macOS  
$ pwd  
/Users/wsv/Desktop/code/ch1-setup
```

---

**Tip:** The `clear` command will clear the terminal of past commands and outputs, so you have a clean slate. The `tab` command autocompletes the line, as we've discussed. And the `↑` and `↓` keys cycle through previous commands to save yourself from typing the same thing over and over again.

To exit, you could close the terminal with your mouse, but the hacker way is to use the shell command `exit` instead, which works by default on Windows; on macOS, the Terminal preferences need to be changed. Click Terminal at the top of the screen, then Preferences from the dropdown menu. Click on Profiles in the top menu and then Shell from the list below. There is a radio button for "When the shell exits." Select "Close the window."

---

### Shell

---

```
$ exit
```

---

With practice, the command line is a far more efficient way to navigate and operate your computer than a mouse. You don't need to be a command line expert to complete this book: I will provide the exact instructions to run each time. But if you are curious, a complete list of shell commands for each operating system is available at [ss64.com](http://ss64.com).

## Install Python 3 on Windows

On Windows, Microsoft hosts a community release of Python 3 in the Microsoft Store. In the search bar at the bottom of your screen, type in “python” and click on the best match result, automatically launching Python 3.11 on the Microsoft Store. Click on the blue “Get” button to download it.

To confirm Python is installed correctly, open a new Terminal window with PowerShell and then type `python --version`.

Shell

---

```
$ python --version
Python 3.11.2
```

---

The result should be at least Python 3.11. Then type `python` to open the Python interpreter from the command line shell.

Shell

---

```
$ python
Python 3.11.2 (tags/v3.11.2:878ead1, Feb 2 2023, 16:38:35)
[MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits", or "license" for more information.
>>>
```

---

## Install Python 3 on Mac

On Mac, the official installer on the Python website is the best approach. In a new browser window, go to the [Python downloads page](#) and click on the button underneath the text “Download the latest version for Mac OS X.” As of this writing, that is Python 3.11. The package will be in your Downloads directory: double-click on it to launch the Python Installer, and follow the prompts.

To confirm the download was successful, open a new Terminal window and type `python3 --version`.

Shell

---

```
$ python3 --version  
Python 3.11.2
```

---

Then type `python3` to open the Python interpreter.

---

#### Shell

---

```
$ python3  
Python 3.11.2 (v3.11.2:878ead1ac1, Feb  7 2023, 10:02:41)  
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin  
Type "help", "copyright", "credits" or "license" for more information.  
>>>
```

---

## Python Interactive Mode

From the command line, type either `python` on Windows or `python3` on macOS to bring up the Python Interpreter, also known as Python Interactive mode. The new prompt of `>>>` indicates that you are now inside Python itself and **not** the command line. If you try any previous shell commands we ran—`cd`, `ls`, `mkdir`—they will raise errors. What *will* work is actual Python code. For example, try out both `1 + 1` and `print("Hello Python!")` making sure to hit the Enter OR Return key after each to run them.

---

#### Shell

---

```
>>> 1 + 1  
2  
>>> print("Hello Python!")  
Hello Python!
```

---

Python's interactive mode is a great way to save time if you want to try out a short bit of Python code. But it has several limitations: you can't save your work in a file, and writing longer code snippets is cumbersome. As a result, we will spend most of our time writing Python and Django in files using a text editor.

To exit Python from the command line, type either `exit()` and the Enter key or use `Ctrl + z` on Windows or `Ctrl + d` on

macOS.

## Virtual Environments

Installing the latest version of Python and Django is the correct approach for any new project. But in the real world, it is common that existing projects rely on older versions of each. Consider the following situation: *Project A* uses Django 3.2, but *Project B* uses Django 4.2. By default, Python and Django are installed *globally* on a computer: installing and reinstalling different versions every time you want to switch between projects is quite a pain.

Fortunately, there is a straightforward solution. *Virtual environments* allow you to create and manage separate environments for each Python project on the same computer. Many areas of software development are hotly debated, but using virtual environments for Python development is not. You should use a dedicated virtual environment for each new Python and Django project.

There are several ways to implement virtual environments, but the simplest is with the [venv](#) module already installed as part of the Python 3 standard library. To try it out, navigate to your desktop's existing ch1-setup directory.

---

### Shell

---

```
# Windows  
$ cd onedrive\desktop\code\ch1-setup  
  
# macOS  
$ cd ~/desktop/code/ch1-setup
```

---

To create a virtual environment within this new directory, use the format `python -m venv <name_of_env>` on Windows or `python3 -m venv <name_of_env>` on macOS. The `-m` part of this command is known as a *flag*, which is a convention to indicate the user is requesting non-default behavior. The

format is usually - and then a letter or combination of letters. The [-m flag](#) is necessary since `venv` is a module name. It is up to the developer to choose a proper environment name, but a common choice is to call it `.venv`, as we do here.

---

#### Shell

---

```
# Windows  
$ python -m venv .venv  
  
# macOS  
$ python3 -m venv .venv
```

---

On Windows, the command `ls` will display the `.venv` directory in our directory. However, on macOS using `ls` it will appear empty. The `.venv` directory *is* there; it's just that it is "hidden" due to the period `.` that precedes the name. Hidden files and directories are a way for developers to indicate that the contents are important and should be treated differently than regular files. To view it, try `ls -la`, which shows all directories and files, even hidden ones.

---

#### Shell

---

```
$ ls -la  
total 0  
drwxr-xr-x  3 wsv  staff   96 Dec 12 11:10 .  
drwxr-xr-x  3 wsv  staff   96 Dec 12 11:10 ..  
drwxr-xr-x  6 wsv  staff  192 Dec 12 11:10 .venv
```

---

You will see that `.venv` is there and can be accessed via `cd` if desired. In the directory is a copy of the Python interpreter and a few management scripts, but you will not need to use it directly in this book.

Once created, a virtual environment must be *activated*. On Windows, an *Execution Policy* must be set to enable running scripts. You only have to do this once to tell Windows that, I know what I'm doing here. The Python docs recommend allowing scripts for the CurrentUser only, which is what we

will do. On macOS, there are no similar restrictions on scripts, so it is possible to run `source .venv/bin/activate` directly.

Here is what the complete commands look like to create and activate a new virtual environment called `.venv`:

---

#### Shell

---

```
# Windows
$ python -m venv .venv
$ Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
$ .venv\Scripts\Activate.ps1
(.venv) $
```

---

```
# macOS
$ python3 -m venv .venv
$ source .venv/bin/activate
(.venv) $
```

---

The shell prompt now has the environment name (`.venv`) prefixed, which indicates that the virtual environment is active. Any future Python packages installed or updated within this location will be confined to the active virtual environment.

To deactivate and leave a virtual environment, type `deactivate`.

---

#### Shell

---

```
(.venv) $ deactivate
$
```

---

The shell prompt no longer has the virtual environment name prefixed, which means the session is now back to normal.

## PyPI (Python Package Index)

[PyPI \(Python Package Index\)](#) is the central location for all Python projects. You can see [Django is there](#) along with

every other Python package we will use in this book.

We will use [pip](#), the most popular package installer, to install Python packages. It already comes included with Python 3, but to ensure we are on the latest version of pip, let's take a moment to update it. Here is the command to run:

Shell

---

```
$ python -m pip install --upgrade pip
```

---

This command will install and upgrade (if needed) the latest version of pip. Notice that we are not in a virtual environment, so this version of pip will be installed globally on our local computer.

Why do we use `python -m pip` instead of just `pip` for this command? The latter does work, but it can cause some issues. Using `python` with the `-m` flag ensures that the intended version of Python is in use, even if you have multiple versions of Python installed on your computer. For example, if you have Python 2.7 and 3.7 installed on your computer, it is possible for `pip install` to use Python 2.7 at one point but Python 3.7 later: not desired behavior. [Brett Cannon](#) has a much fuller explanation if you are curious about the underlying reasons why this is the case.

Python is now installed, we know how to use virtual environments, and we've updated our version of pip. It is time to install Django itself for the first time.

## Install Django

In the `ch1-setup` directory, reactivate the existing virtual environment and install Django.

Shell

---

```
# Windows
$ .venv\Scripts\Activate.ps1
(.venv) $ python -m pip install django~=4.2.0

# macOS
$ source .venv/bin/activate
(.venv) $ python3 -m pip install django~=4.2.0
```

---

This command uses the comparison operator, `~=`, to install the latest version of Django 4.2.x. As I type these words, the newest version is 4.2.0, but soon it will be 4.2.1 and then a month later 4.2.2. By using `~=4.2.0`, we ensure that the latest version of 4.2.x will be installed when the user executes the command.

If we did not “pin” our version number in this way—if we just installed Django using the command `python -m pip install django`—then the latest version of Django will be installed even if, in the future, it is 5.0 or 5.1 or even 6.0. There is no guarantee that all the code in this book will work perfectly on a later version of Django. By specifying the version number for each software package installed, you can update them one at a time to ensure compatibility.

**Note:** I will provide Windows and macOS commands throughout this book if they differ. However, when it comes to using `python` on Windows vs. `python3` on macOS, the default will be `python` for conciseness.

## First Django Project

To create a new Django project, use the command `django-admin startproject django_project`. A Django project can have almost any name, but we will use `django_project` in this book.

### Shell

---

```
(.venv) $ django-admin startproject django_project .
```

---

It's worth pausing here to explain why you should add a period (.) to the end of the previous command. If you just run `django-admin startproject django_project`, then by default Django will create this directory structure:

---

#### Layout

---

```
django_project/
  └── django_project
    ├── __init__.py
    ├── asgi.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
  └── manage.py
  .venv/
```

---

Do you see the multiple `django_project` directories? First, a top-level `django_project` directory is created, and then *another* one within it that contains the files we need for our Django project. Opinions differ on the “best” approach within the Django community, but it feels redundant to me to have these two directories with the same name and makes deployment easier later on, so I prefer adding a period to the end that installs Django in the current directory.

---

#### Layout

---

```
└── django_project
  ├── __init__.py
  ├── asgi.py
  ├── settings.py
  ├── urls.py
  └── wsgi.py
  └── manage.py
  .venv/
```

---

As you progress in your journey learning Django, you will bump up against more and more situations like this where there are different opinions within the Django community on the correct best practice. Django is eminently customizable, which is a great strength; however, the tradeoff is that this flexibility comes at the cost of seeming complexity.

Generally speaking, it's a good idea to research any such issues, make a decision, and then stick with it!

## The Development Server

Django includes a built-in, lightweight Web server for local development that is accessible via the [runserver](#) command. The development server automatically reloads Python code for each request so you don't need to restart the server for code changes to take effect. Be aware, however, that some actions like adding files will not automatically trigger a restart so if your code is not working as expected, doing a manual restart is always a good first step.

By default, the server runs on port 8000 on the IP address 127.0.0.1, which is known as the "loopback address" because no data is actually sent from our computer (host) to the local network or internet; instead, it is "looped back" on itself so the computer sending the data becomes the recipient.

Let's confirm everything is working correctly by starting up the development server now. We'll use `manage.py` to execute the `runserver` command.

### Shell

---

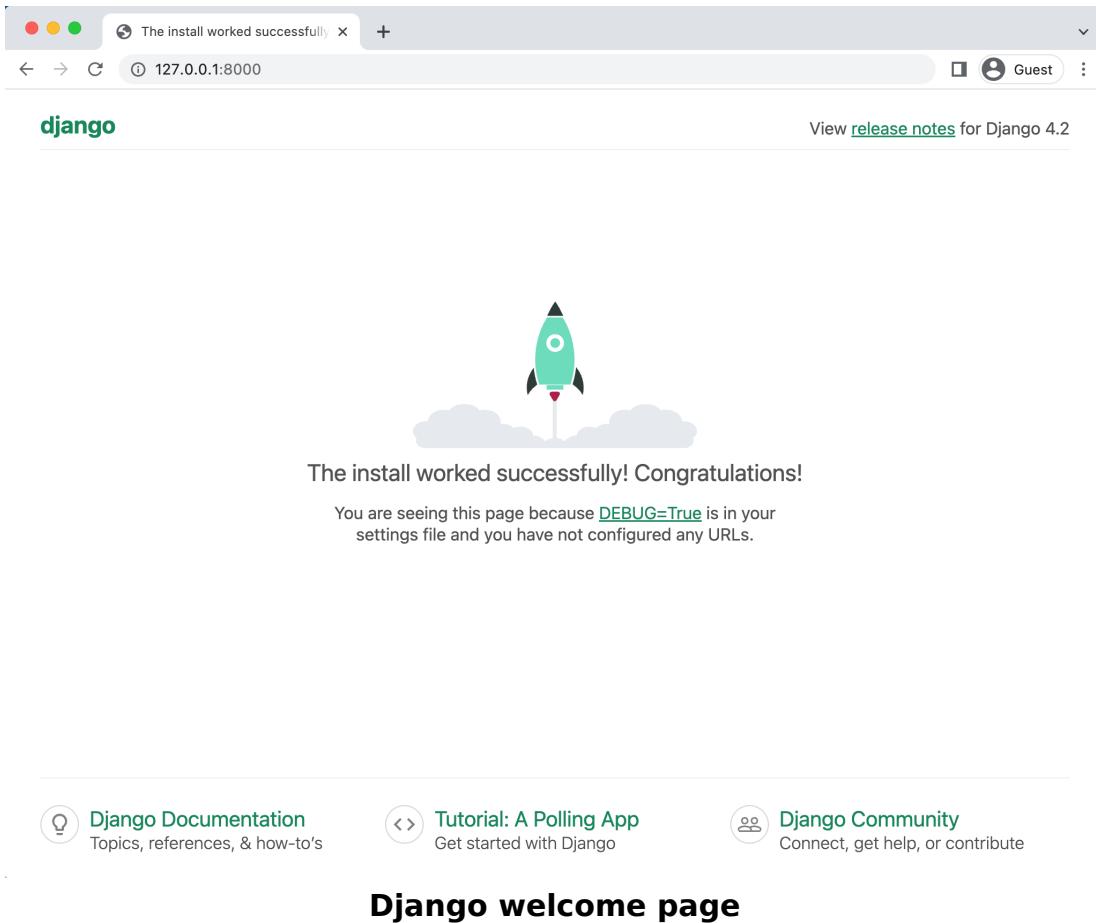
```
(.venv) $ python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until
you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
April 13, 2023 - 13:15:03
Django version 4.2, using settings 'django_project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-BREAK.
```

---

Don't worry about the text in red about 18 unapplied migrations. We'll get to that shortly. The important part, for now, is to visit <http://127.0.0.1:8000/> in your web browser and make sure the following image is visible:



If you are on Windows, you will see the final line says to use CONTROL-BREAK to quit, whereas on macOS, it is CONTROL-C. Newer Windows keyboards often do not have a Pause/Break key so using the c key usually works.

Resist the temptation to use the development server for production. It can handle only a single request at a time reliably and has not undergone any security audits or performance tests. There are dedicated web servers for that sort of thing that we will use in later chapters.

If you want to change the port number you can do so by passing in a command-line argument. For example, you could stop the server and then restart it on port 8080.

---

Shell

---

```
(.venv) $ python manage.py runserver 8080
```

---

Generally speaking, Django developers don't need to do this but as will become a running theme in this book, Django provides ultimately flexibility if needed.

For readers new to web development, be aware that is it also possible to visit `http://localhost:8000/` to see the running Django website. `localhost` is a common shorthand for `127.0.0.1`. In this book we will default to `127.0.0.1:8000` because that is what Django outputs in the terminal but either option is fine.

Go ahead and stop the local server with the appropriate command and then exit the virtual environment by typing `deactivate` and hit Return.

---

Shell

---

```
# Windows or macOS  
(.venv) $ deactivate
```

---

We'll get lots of practice with virtual environments in this book, so don't worry if it's a little confusing. The basic pattern for any new Django project is to make and activate a virtual environment, install Django, and then run `startproject`.

It's worth noting that only one virtual environment can be active in a command line tab. In future chapters, we will create a new virtual environment for each new project, so either make sure to deactivate your current environment or open up a new tab for new projects.

## **Text Editors**

The command line is where we execute commands for our programs, but a text editor is where the code is actually written. The computer doesn't care what text editor you use—the result is just code—but a good text editor can provide helpful hints and catch typos for you.

Many modern text editors are available that come with helpful extensions to make Python and Django development more accessible. [PyCharm](#) is a very popular option that has both a paid Professional and free Community version. [Visual Studio Code](#) is also free, easy to install, and enjoys widespread popularity. It does not matter what text editor you decide to use: the result is just code.

## **VSCode Configurations**

If you're not already using a text editor, download and install VSCode from the official website. There are three recommended configurations you can add to improve your developer productivity.

The first is to add the official Python extension to VSCode. On Windows, navigate to File -> Settings -> Extensions; On macOS, Code -> Settings -> Extensions to launch a search bar for the extensions marketplace. Enter “python” to bring up the Microsoft extension as the first result. Install it.

The second is adding [Black](#), a Python code formatter that has quickly become the default within the Python community. In the terminal run the command `python -m pip install black` on Windows or `python3 -m pip install black` on macOS.

---

```
(.venv) $ python -m pip install black
```

---

Next, within VSCode open up the settings by navigating to File -> Preferences -> Settings on Windows or Code -> Preferences -> Settings on macOS. Search for “python formatting provider” and select black from the dropdown options. Then search for “format on save” and enable “Editor: Format on Save”. Black will automatically format your code whenever a \*.py file is saved.

To confirm this is working, use your text editor to create and save a new file called hello.py within the ch1-setup directory located on your desktop and type in the following using single quotes:

```
hello.py  
print('Hello, World!')
```

---

On save, it should update automatically to using double quotes, which is [Black's default preference](#): print("Hello, World!"). That means everything is working correctly.

The third configuration makes it possible to open VSCode from your terminal. This is quite useful since a standard workflow will be to open up the terminal, navigate to a directory you want to work on, and then open it with VSCode.

Within VSCode simultaneously press Command + Shift + P to open the command palette, which allows us to customize our VS Code settings. In the command palette type shell: the top result will be “Shell Command: Install code command in PATH.” Then hit enter to install this shortcut. There will be a success message saying “Shell command ‘code’ successfully installed in PATH.” The [PATH variable](#), by the way, is often used to customize terminal prompts.

Now go back to your terminal and navigate to the ch1-setup directory. If you type `code .` it will open up in VS Code.

#### Shell

---

```
(.venv) $ code .
```

---

## Install Git

The final step is to install *Git*, a version control system indispensable to modern software development. With Git, you can collaborate with other developers, track all your work via commits, and revert to any previous version of your code even if you accidentally delete something important! This is not a book on Git, so all necessary commands are given and briefly explained, but there are numerous resources available for free on the internet if you'd like to learn [more about Git itself](#).

On Windows, navigate to the official website at <https://git-scm.com/> and click on the “Download” link, which should install the proper version for your computer. Save the file, open your Downloads folder, and double-click on the file to launch the Git for Windows installer. Click the “Next” button through most early defaults as they are fine and can always be updated later. Make sure that under “Choosing the default editor used by Git”, the selection is for “Use Visual Studio Codeas Git’s default editor.” And in the section on “Adjusting the name of the initial branch in new repositories,” make sure the option to “Override the default branch name for new repositories” is selected so that “main” is used.

To confirm Git is installed on Windows, close all current shell windows and then open a new one which will load the changes to our PATH variable. Type in `git --version` to display the installed version of Git.

---

## Shell

---

```
# Windows  
$ git --version  
git version 2.39.1.windows.1
```

---

On macOS, [Xcode](#) is primarily designed for building iOS apps but includes many developer features needed on macOS. Currently, installing Git via Xcode is the easiest option. To check if Git is installed on your computer, type `git --version` in a new terminal window.

---

## Shell

---

```
# macOS  
$ git --version
```

---

If you do not have Git installed, a popup message will ask if you want to install it as part of “command line developer tools.” Select “Install” which will load Xcode and its command line tools package. Or if you do not see the message, type `xcode-select --install` instead to install Xcode directly.

Be aware that Xcode is a very large package, so the initial download may take some time. Xcode is primarily designed for building iOS apps and also includes many developer features needed on macOS. Once the download is complete, close all existing terminal shells, open a new window, and type in `git --version` to confirm the install worked.

---

## Shell

---

```
# macOS  
$ git --version  
git version 2.37.1 (Apple Git-137.1)
```

---

Once Git installs on your local machine, we need to do a one-time *system* configuration by declaring the name and email address associated with all your Git commits. We will also set the default branch name to `main`. Within the

command line shell, type the following two lines. Make sure to update them with your name and email address, not the defaults of “Your Name” and “yourname@email.com”!

#### Shell

---

```
$ git config --global user.name "Your Name"  
$ git config --global user.email "yourname@email.com"  
$ git config --global init.defaultBranch main
```

---

If you desire, you can always change these configs later by retying the same commands with a new name or email address.

## Conclusion

It is a challenging task to configure a software development environment from scratch. Even experienced programmers have difficulty with the job, but it is a one-time pain that is more than worth it. We know have the ability to start up new Django projects quickly and have learned about the command line, Python interactive mode, how to install the latest version of Python and Django, configured our text editor, and installed Git. Everything is ready for our first proper Django website in the next chapter.

# Chapter 2: Hello World App

In this chapter, we will build a Django project that says “Hello, World” on the homepage, the traditional way to start a new web framework. We’ll also work with Git for the first time and deploy our code to GitHub. The complete source code for this and all future chapters is available online at [the official GitHub repo](#) for the book.

## How Websites Work

If you are new to programming and building websites, it is worth quickly reviewing the fundamentals behind the Internet and the World Wide Web. The Internet is a broad global system of interconnected computers; the World Wide Web is a subset of the Internet that refers to hypertext documents linked together via hyperlinks (in other words, webpages).

The internet relies on various “communication protocols,” which are like human languages in that they allow computers all over the world to communicate with one another via agreed-upon conventions. For example, file sharing uses the *File Transfer Protocol (FTP)*, sending email uses the *Simple Mail Transfer Protocol (SMTP)*, communicating through voice uses the *Voice over Internet Protocol (VoIP)*, and viewing webpages—our area of particular interest in this book—uses the *Hypertext Transfer Protocol (HTTP)*.

In common conversation, the terms “Internet” and “World Wide Web” are frequently used interchangeably, but as web developers it is important to realize that they refer to quite different things.

Underpinning the world wide web is the [client-server model](#). A “client” refers to any internet-connected device making service requests, such as a computer, a phone, a dishwasher, etc.; the “server” is computer hardware or software that responds to service requests. In other words, the client makes a request and the server returns a response.

The computers powering the internet are often referred to as servers, but really they’re just computers connected to the internet all the time running special software that lets them “serve” information to other computers. Your own computer can be a server, but in practice most servers exist in large data centers (aka “the cloud”).

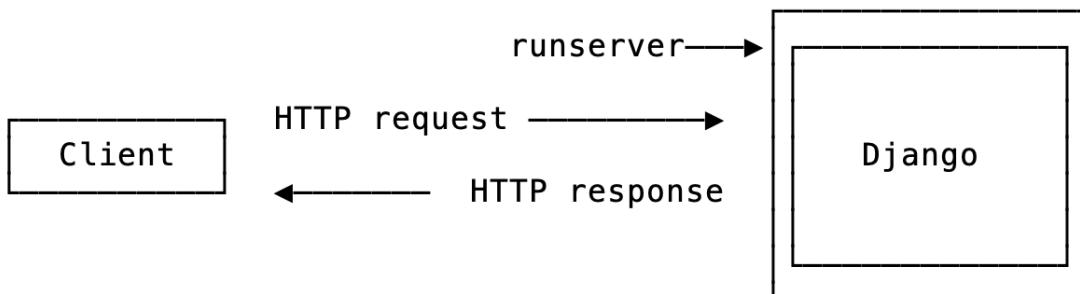
Since we are using the HTTP protocol for all of this, we can be more specific and say that a client makes an HTTP request and a server responds with an HTTP response.



The full domain name for a website like LearnDjango.com is actually `https://learndjango.com`. The `https://` at the beginning specifies that we are using HTTP as our protocol: HTTPS is the encrypted version of HTTP and now accounts for the majority of web traffic. Modern web browsers will automatically add this on for you so most regular users simply type the domain name and are unaware of the HTTP underpinnings.

Each time you type a URL address into your web browser—for example <https://learndjango.com>—an HTTP request is sent to the appropriate server which then returns an HTTP response. Your web browser then renders the data from the HTTP response to create a webpage. Every time you click on a link or request a new URL this *HTTP request/response cycle* begins again. Back and forth the communication goes.

In production, a Django website like LearnDjango.com is hosted on a physical server and automatically processes HTTP requests and responses. It relies on additional machinery that we will build out during later projects in the book. In local development, things are much simpler. Django comes with a lightweight development server, [runserver](#), that manages HTTP requests and responses, helps Django generate dynamic content from the database and serves static files (more on these later). It's quite powerful. We can therefore update our first image with a new one featuring runserver wrapped around Django.



**Django HTTP request/response cycle**

If you want to see the actual raw data included in an HTTP response, find the “View Source” option in your web browser of choice. In the Chrome web browser, at the top of the window go to View -> Developer -> View Source to take a look. It isn't very human-readable! That's why we use web

browsers to compile the responses into human-readable webpages.

## How Web Frameworks Work

There are two broad categories of websites: static and dynamic. A *static* website consists of individual HTML documents that are sent as-is over HTTP to your web browser. If your website has ten pages then there must be ten corresponding HTML files. This approach can work for very small websites but quickly falls apart when a website needs hundreds or thousands of pages. A *dynamic* website consists of a database, HTML templates, and an application server that can update the files before sending them to your browser via HTTP. Most large websites adopt this approach since it means millions of webpages can be composed of only a few HTML templates, a small amount of logic, and a big database.

Django is designed for dynamic websites and abstracts away much of the difficulty inherent in creating a website from scratch. If you think about it, most websites require the same fundamental tools:

- a way to process URL requests
- a way to connect to a database
- a way to generate dynamic content by filtering data from the database
- a way to create templates for styling HTML and adding CSS, images, etc as needed

## Model-View-Controller vs Model-View-Template

If you have built websites before you might be familiar with the **Model-View-Controller (MVC)** pattern. It is used by web frameworks including Ruby on Rails, Spring (Java),

Laravel (PHP), and ASP.NET (C#). This is a popular way to *internally* separate the data, logic, and display of an application into separate components that are easier for a developer to reason about.

In the traditional MVC pattern there are three major components:

- Model: Manages data and core business logic
- View: Renders data from the model in a particular format
- Controller: Accepts user input and performs application-specific logic

Django's approach is sometimes called **Model-View-Template (MVT)** but it is really a 4-part pattern that also incorporates URL configuration. Something like **Model-View-Template-URL (MVTU)** would be a more accurate description:

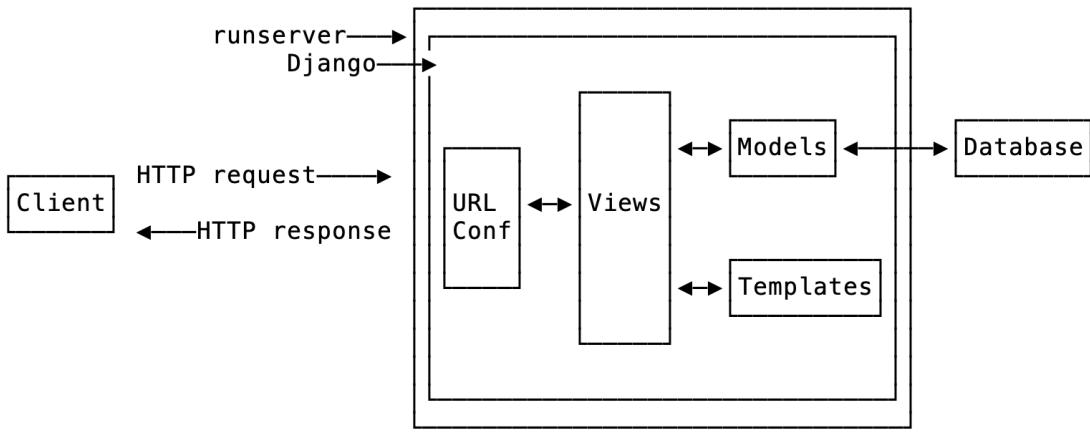
- Model: Manages data and core business logic
- View: Describes *which* data is sent to the user but not its presentation
- Template: Presents the data as HTML with optional CSS, JavaScript, and static assets
- URL Configuration: Regular expression components configured to a View

The “View” in MVC is analogous to a “Template” in Django while the “Controller” in MVC is divided into a Django “View” and “URL config.” This is understandably quite confusing to newcomers. To help, let’s map out the order of a given HTTP request/response cycle for Django.

When you type in a URL, such as <https://djangoproject.com>, the first thing that happens within our Django project is that runserver kicks into gear and helps Django look for a

matching URL pattern (contained in `urls.py`). The URL pattern is linked to a single view (contained in `views.py`) which combines the data from the model (stored in `models.py`) and the styling from a template (any file ending in `.html`). The view then returns a HTTP response to the user.

A simplified version of this complete Django flow looks like this:



**Django request/response cycle**

If you are new to web development the distinction between MVC and MVT will not matter much. This book demonstrates the Django way of doing things so there won't be confusion. However if you are a web developer with previous MVC experience, it can take a little while to shift your thinking to the "Django way" of doing things which is more loosely coupled and allows for easier modifications than the MVC approach.

## Initial Set Up

To begin our first Django website, open up a new command line shell or use the built-in terminal on VS Code. For the

latter, click on “Terminal” at the top and then “New Terminal” to bring it up on the bottom of the screen.

Make sure you are not in an existing virtual environment by checking there is nothing in parentheses before your command line prompt. You can even type deactivate to be completely sure. Then navigate to the code directory on your desktop and create a helloworld directory with the following commands.

---

#### Shell

---

```
# Windows
$ cd onedrive\desktop\code
$ mkdir helloworld
$ cd helloworld

# macOS
$ cd ~/desktop/code
$ mkdir helloworld
$ cd helloworld
```

---

Create a new virtual environment called .venv, activate it, and install Django with Pip as we did in the previous chapter. We can also install Black now, too.

---

#### Shell

---

```
# Windows
$ python -m venv .venv
$ .venv\Scripts\Activate.ps1
(.venv) $ python -m pip install django~=4.2.0
(.venv) $ python -m pip install black

# macOS
$ python3 -m venv .venv
$ source .venv/bin/activate
(.venv) $ python3 -m pip install django~=4.2.0
(.venv) $ python3 -m pip install black
```

---

Now we'll use the Django startproject command to make a new project called django\_project. Don't forget to include the period (.) at the end of the command so that it is installed in our current directory.

## Shell

---

```
(.venv) $ django-admin startproject django_project .
```

---

Let's pause for a moment to examine the default project structure Django has provided for us. You can examine this visually by opening the new directory with your mouse on the desktop. The `.venv` directory may not be initially visible because it is a "hidden file" but it is nonetheless still there and contains information about our virtual environment.

## Code

---

```
└── django_project
    ├── __init__.py
    ├── asgi.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
    └── manage.py
    └── .venv/
```

---

Django has created a `django_project` directory and a `manage.py` file. Within `django_project` are five new files:

- `__init__.py` indicates that the files in the folder are part of a Python package. Without this file, we cannot import files from another directory which we will be doing a lot of in Django!
- `asgi.py` allows for an optional [Asynchronous Server Gateway Interface](#) to be run.
- `settings.py` controls our Django project's overall settings
- `urls.py` tells Django which pages to build in response to a browser or URL request
- `wsgi.py` stands for [Web Server Gateway Interface](#), more on this in the next chapter when we do our first deployment

The `manage.py` file is not part of `django_project` but is used to execute various Django commands such as running the local web server or creating a new app.

Let's try out our new project by using Django's lightweight built-in web server for local development purposes.

#### Shell

---

```
(.venv) $ python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

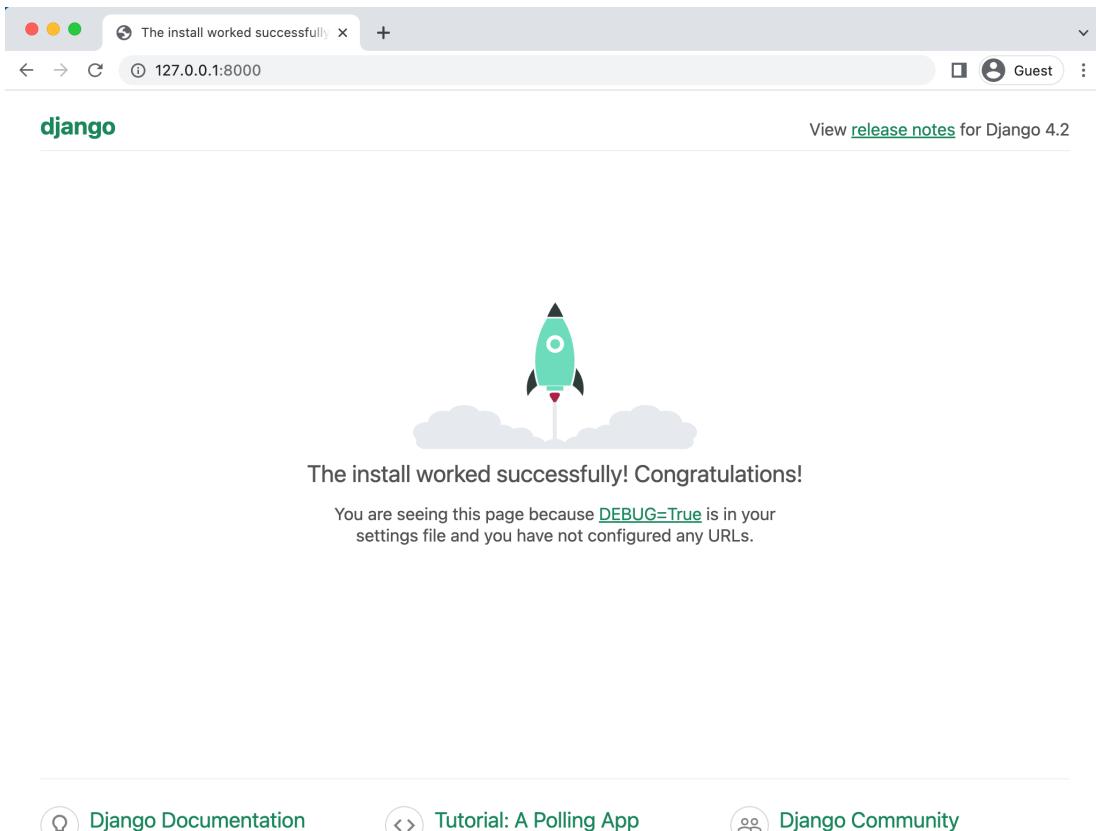
System check identified no issues (0 silenced).

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

April 13, 2023 - 13:18:12
Django version 4.2, using settings 'django_project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

---

If you visit <http://127.0.0.1:8000/> you should see the following image:



## Django welcome page

It is safe to ignore the warning about 18 unapplied migrations at this point. Django is complaining that we have not yet “migrated” our initial database. Since we won’t actually use a database in this chapter the warning won’t affect the end result. However, since warnings are annoying to see, we can remove it by first stopping the local server with the command `Control+c` and then running `python manage.py migrate`.

**Note:** Going forward when there is a common command for both Windows and macOS, `python` will be used as the default rather than referencing both `python` on Windows and `python3` on macOS.

Shell

---

```
$ python manage.py migrate

Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
```

---

What Django has done here is create a SQLite database and migrated its built-in apps provided for us. This is represented by the new db.sqlite3 file in our directory.

---

#### Code

---

```
└── django_project
    ├── __init__.py
    ├── asgi.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
    └── db.sqlite3 # new
    └── manage.py
    └── .venv/
```

---

If you execute `python manage.py runserver` again you should no longer see any warnings.

## Create An App

A single Django project can contain many “apps,” which is an organizational technique for keeping our code clean and readable. Each app should control an isolated piece of

functionality. For example, an e-commerce site might have one app for user authentication, another app for payments, and a third app to power item listing details. That's three distinct apps that all live within one top-level project.

Another example is if you build a social networking site—let's call it a Twitter clone—that has an option to email users when someone comments on a post. Initially you might create a tweet app and build the email functionality within it.

However over time the complexity of both the tweets and the emails being sent will likely grow and so splitting it into two apps—`tweet` and `emails`—could make sense. This is especially true if you want to reuse the email parts elsewhere in your larger project.

How and when you split functionality into apps is very subjective but a good rule of thumb is that when a single app feels like it's doing too much, it is time to split features into their own apps which each have a single function.

To add a new app go to the command line and quit the running server with `Control+c`. Then use the `startapp` command followed by the name of our app which will be `pages`.

---

#### Shell

---

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

---

If you look visually at the `helloworld` directory Django has created within it a new `pages` directory containing the following files:

---

#### Code

---

```
pages
├── __init__.py
├── admin.py
├── apps.py
└── migrations
    └── __init__.py
└── models.py
```

```
|   └── tests.py  
|   └── views.py
```

---

Let's review what each new pages app file does:

- `admin.py` is a configuration file for the built-in Django Admin app
- `apps.py` is a configuration file for the app itself
- `migrations/` keeps track of any changes to our `models.py` file so it stays in sync with our database
- `models.py` is where we define our database models which Django automatically translates into database tables
- `tests.py` is for app-specific tests
- `views.py` is where we handle the request/response logic for our web app

Notice that the model, view, and url from the MVT pattern are present from the beginning. The only thing missing is a template which we'll add shortly.

Even though our new app exists within the Django project, Django doesn't "know" about it until we explicitly add it to the `django_project/settings.py` file. In your text editor open the file up and scroll down to `INSTALLED_APPS` where you'll see six built-in Django apps already there. Add `pages` at the bottom.

---

Code

```
# django_project/settings.py  
INSTALLED_APPS = [  
    "django.contrib.admin",  
    "django.contrib.auth",  
    "django.contrib.contenttypes",  
    "django.contrib.sessions",  
    "django.contrib.messages",  
    "django.contrib.staticfiles",  
    "pages", # new  
]
```

---

## Your First View

For our first website we'll create a Web page that outputs the text "Hello, World!" This is a static page that does not involve a database or even a templates file. Instead, it is a good introduction to how views and URLs work within Django.

A view is a Python function that accepts a Web request and returns a Web response. The response can be the HTML contents of a Web page, a redirect, a 404 error, an image, or really anything.

When a Web page is requested, Django automatically creates an `HttpRequest` object that contains metadata about the request. Then Django loads the appropriate view, passing the `HttpRequest` in as the first argument to the view function. The view is ultimately responsible for returning an `HttpResponse` object.

In our pages app there is already a file called `views.py` which comes with the following default text:

---

Code

---

```
# pages/views.py
from django.shortcuts import render

# Create your views here.
```

---

[render\(\)](#) is a Django shortcut function that *can* be used to create views, however there is an even simpler approach possible which is to instead use the built-in [HttpResponse](#) method. That's what we'll do here.

Update the `pages/views.py` file with the following code:

---

Code

---

```
# pages/views.py
from django.http import HttpResponse
```

```
def home_page_view(request):
    return HttpResponseRedirect("Hello, World!")
```

---

Let's step through it line-by-line:

- First, we import `HttpResponse` from the `django.http` module.
- Then we define a function called `home_page_view`. The name can be almost anything you like, for example `some_other_view`, but it is best to use something descriptive. Note as well that we are using `snake_case` here for our naming style—all lowercase words separated by underscores—which is traditional in Python for function and variable names.
- Finally, the view returns an `Http Response` object with the string of text “Hello, World!”

## URLconfs

Moving along we need to configure our URLs. In your text editor, create a new file called `urls.py` within the `pages` app. Then update it with the following code:

Code

---

```
# pages/urls.py
from django.urls import path

from .views import home_page_view

urlpatterns = [
    path("", home_page_view, name="home"),
]
```

---

On the top line we import `path` from Django to power our URL pattern and on the next line we import our views. By referring to the `views.py` file as `.views` we are telling Django to look within the current directory for a `views.py` file and import the view `home_page_view` from there.

Our URL pattern here has three parts:

- the empty string, ""
- a reference to the view called `home_page_view`
- an optional [named URL pattern](#) called "home"

In other words, if the user requests the homepage represented by the empty string "", Django should use the view called `home_page_view`.

We're *almost* done at this point. The last step is to update our `django_project/urls.py` file: it is the first place all URL requests come into, but we can "include" URLs from individual apps, such as `pages`. That is what we will do here. The [Django docs](#) have a fuller description of how this process works under-the-hood.

Here is what the updated code looks like:

---

Code

---

```
# django_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
]
```

---

We've imported [django.urls.include](#) on the second line next to `path` and then created a new URL pattern for our `pages` app. Now whenever a user visits the homepage-represented by the empty string here, "", they will be routed to the `urls` file in the `pages` app. The empty string in that one is linked to the `home_page_view` view.

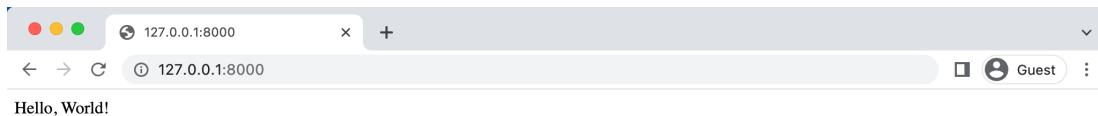
This need for two separate `urls.py` files is often confusing to beginners. Think of the top-level `django_project/urls.py` as the gateway to various URL patterns distinct to each app.

We have all the code we need now. To confirm everything works as expected, restart our Django server:

#### Shell

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

If you refresh the browser for `http://127.0.0.1:8000/` it now displays the text “Hello, World!”



## Hello World homepage

## Git

In the previous chapter, we installed the version control system Git. Let’s use it here. The first step is to initialize (or add) Git to our repository. Make sure you’ve stopped the local server with Control+c, then run the command `git init`.

#### Shell

```
(.venv) $ git init
```

If you then type `git status` you’ll see a list of changes since the last Git commit. Since this is our first commit, this list is all of our changes so far.

#### Shell

```
(.venv) $ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

  .venv
```

```
django_project/
db.sqlite3
manage.py
pages/
nothing added to commit but untracked files present (use "git add" to track)
```

---

Note that our virtual environment `.venv` is included which is *not* a best practice. It should be kept out of Git source control since secret information such as API keys and the like are often included in it. The solution is to create a new file in the project-level directory called `.gitignore` which tells Git what to ignore. The period at the beginning indicates this is a “hidden” file. The file still exists but it is a way to communicate to developers that the contents are probably meant for configuration and not source control.

Here is how your project structure should look now:

Layout

```
└── django_project
    ├── __init__.py
    ├── asgi.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
└── pages
    ├── migrations
    │   ├── __init__.py
    │   ├── admin.py
    │   ├── apps.py
    │   ├── models.py
    │   ├── tests.py
    │   ├── urls.py
    │   └── views.py
    ├── .gitignore # new
    ├── db.sqlite3
    ├── manage.py
    └── .venv/
```

---

In this new `.gitignore` file, add a single line for `.venv`.

```
.gitignore
.venv/
```

---

If you run `git status` again you will see that `.venv` is not longer there. It has been “ignored” by Git.

At the same time, we *do* want a record of packages installed in our virtual environment. The current best practice is to create a `requirements.txt` file with this information. The command `pip freeze` will output the contents of your current virtual environment and by using the `>` operator we can do all this in one step: output the contents into a new file called `requirements.txt`. If your server is still running enter `Ctrl+C` and `Enter` to exit before entering this command.

---

#### Shell

---

```
(.venv) $ pip freeze > requirements.txt
```

---

A new `requirements.txt` file will appear with all our installed packages and their dependencies. If you look *inside* this file you’ll see there are actually nine packages even though we have installed only two: Django and Black. That’s because Django and Black depend on *other* packages, too. It is often the case that when you install one Python package you’re also installing multiple dependent packages, too. Since it is difficult to keep track of all the packages a `requirements.txt` file is very important.

---

#### requirements.txt

---

```
asgiref==3.6.0
black==23.3.0
click==8.1.3
Django==4.2
mypy-extensions==1.0.0
packaging==23.1
pathspec==0.11.1
platformdirs==3.2.0
sqlparse==0.4.3
```

---

Next want to perform our first Git commit to store all the recent changes. Git comes with a [lengthy list](#) of options/flags that can be used. For example, to add *all*

recent changes we can use `git add -A`. And then to commit the changes we will use a `-m` flag (this one stands for “message”) to describe what has changed. It is **very** important to always add a message to your commits since most projects will easily have hundreds if not thousands of commits. Adding a descriptive message each time helps with debugging efforts later on since you can search through your commit history.

#### Shell

---

```
(.venv) $ git add -A  
(.venv) $ git commit -m "initial commit"
```

---

In professional projects a `.gitignore` file is typically quite lengthy. For efficiency and security reasons, there are often quite a few directories and files that should be removed from source control.

## GitHub

It's a good habit to create a remote repository of your code for each project. This way you have a backup in case anything happens to your computer and more importantly, it allows for collaboration with other software developers. Popular choices include GitHub, Bitbucket, and GitLab. When you're learning web development, it is **highly recommended** to use private rather than public repositories so you don't inadvertently post critical information such as passwords online.

We will use GitHub in this book but all three services offer similar functionality for newcomers. Sign up for a free account on GitHub's homepage and verify your email address. It is also now required to add 2FA (two-factor authentication) for increased security. Once fully signed up

navigate to the “Create a new repository” page located at <https://github.com/new>.

Enter the repository name `hello-world` and click on the radio button next to “Private” rather than “Public.” Then click on the button at the bottom for “Create Repository.”

Your first repository is now created! However there is no code in it yet. Scroll down on the page to where it says “...or push an existing repository from the command line.” That’s what we want. Copy the text immediately under this headline and paste it into your command line. Here is what it looks like for me with my GitHub username of `wsvincent`. Your username will be different.

#### Shell

---

```
$ git remote add origin https://github.com/wsvincent/hello-world.git  
$ git branch -M main  
$ git push -u origin main
```

---

The first line adds the remote repo on GitHub to our local Git configuration, the next line establishes the default branch as `main`, and the third line “pushes” the code up to GitHub’s servers. The `-u` flag creates a tracking reference for every new branch that you successfully push onto the remote repository. The next time we push commits we will only need the command `git push origin main`.

Assuming everything worked properly, you can now go back to your GitHub webpage and refresh it. Your local code is now hosted online!

## SSH Keys

Unfortunately, there is a good chance that the last command yielded an error if you are a new developer and do not have SSH keys already configured.

## Shell

---

```
ERROR: Repository not found.  
fatal: Could not read from remote repository.
```

Please make sure you have the correct access rights  
and the repository exists.

---

This cryptic message means we need to configure SSH keys.  
This is a one-time thing but a bit of a hassle to be honest.

*The Secure Shell Protocol (SSH)* is a protocol used to ensure private connections with a remote server. The process involves generating unique SSH keys and storing them on your computer so only GitHub can access them. For regular websites authentication is done via username/password powered by *Hypertext Transfer Protocol Secure (HTTPS)*, essentially encrypted HTTP. But for very important websites-and your GitHub repos which store all your code count as such-SSH is a more secure approach.

First, check whether you have existing SSH keys. Github has [a guide to this](#) that works for Mac, Windows, and Linux. If you *don't* have existing public and private keys, you'll need to generate them. GitHub, again, has [a guide on doing this](#).

Once complete you should be able to execute the `git push -u origin main` command successfully!

It's normal to feel overwhelmed and frustrated if you become stuck with SSH keys. GitHub has a lot of resources to walk you through it but the reality is that it's very intimidating the first time. If you're truly stuck, continue with the book and come back to SSH Keys and GitHub with a full night's sleep. I can't count the number of times a clear head has helped me process a difficult programming issue.

Assuming success with GitHub, go ahead and exit our virtual environment with the `deactivate` command.

## Shell

---

```
(.venv) $ deactivate
```

---

You should no longer see parentheses on your command line, indicating the virtual environment is no longer active.

## Conclusion

Congratulations! We've covered a lot of fundamental concepts in this chapter. We built our first Django application and learned about Django's project/app structure. We started to learn about views, urls, and the internal Django web server. And we worked with Git to track our changes and pushed our code into a private repo on GitHub.

If you become stuck at any point, compare your code against the [official repo](#).

Continue on to the next chapter where we'll build and deploy a more complex Django application using templates and class-based views.

# Chapter 3: Pages App

In this chapter, we will build, test, and deploy a *Pages* app containing a homepage and an about page. We are still not using the database yet—that comes in the next chapter—but we'll learn about class-based views and templates which are the building blocks for the more complex web applications built later on in the book.

## Initial Set Up

Our initial setup is similar to that in the previous chapter and contains the following steps:

- make a new directory for our code called `pages` and navigate into it
- create a new virtual environment called `.venv` and activate it
- install Django and Black
- create a new Django project called `django_project`
- create a new app called `pages`

On the command line, ensure you're not working in an existing virtual environment. If there is text before the command line prompt—either `>` on Windows or `%` on macOS—then you are! Make sure to type `deactivate` to leave it.

Within a new command line shell, navigate to the `code` folder on the desktop, create a new folder called `pages`, change directories into it, and activate a new Python virtual environment called `.venv`.

---

### Shell

```
# Windows
$ cd onedrive\desktop\code
$ mkdir pages
```

```
$ cd pages
$ python -m venv .venv
$ .venv\Scripts\Activate.ps1
(.venv) $

# macOS
$ cd ~/desktop/code
$ mkdir pages
$ cd pages
$ python3 -m venv .venv
$ source .venv/bin/activate
(.venv) $
```

---

Next, install Django and Black, create a new project called django\_project, and make a new app called pages.

#### Shell

---

```
(.venv) $ python -m pip install django~=4.2.0
(.venv) $ python -m pip install black
(.venv) $ django-admin startproject django_project .
(.venv) $ python manage.py startapp pages
```

---

Remember that even though we added a new app, Django will not recognize it until it is added to the INSTALLED\_APPS setting within django\_project/settings.py. Open your text editor and add it to the bottom now:

#### Code

---

```
# django_project/settings.py
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "pages", # new
]
```

---

Initialize the database with migrate and start the local web server with runserver.

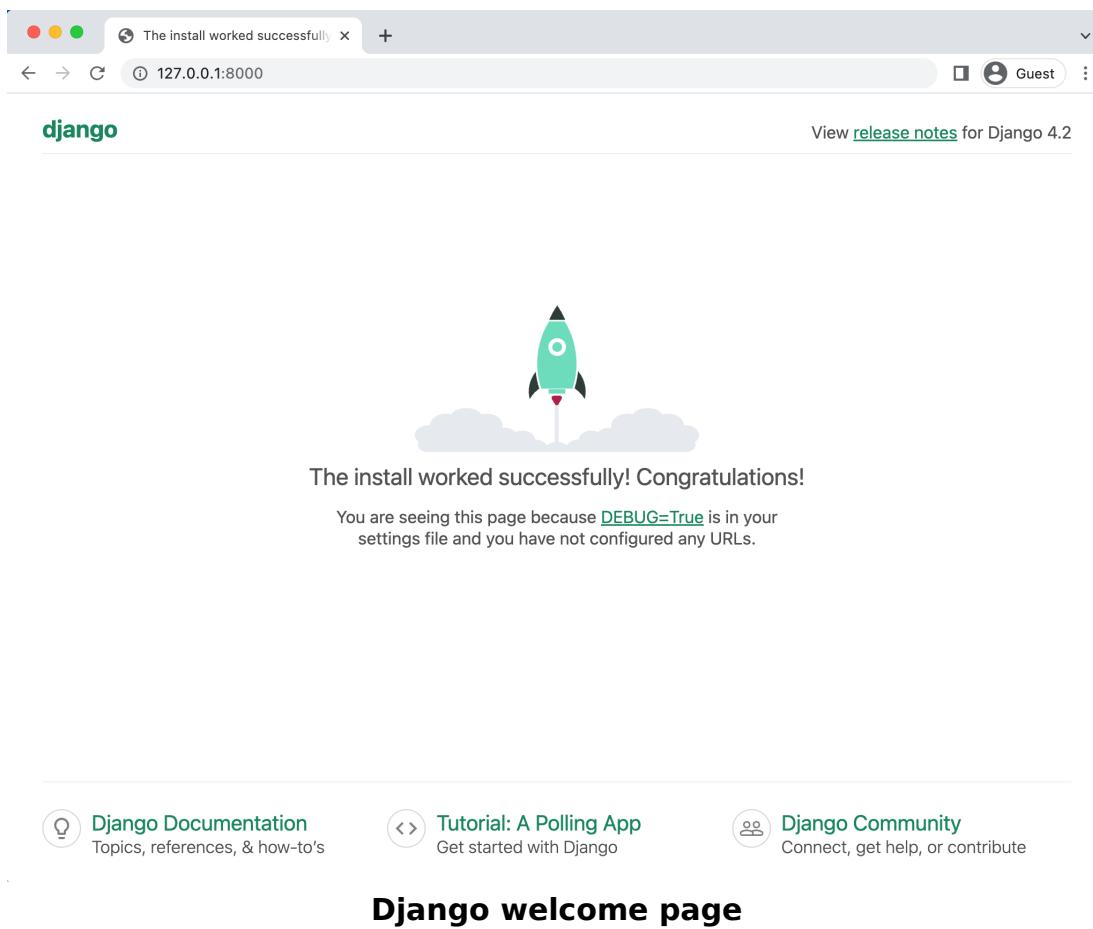
#### Shell

---

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

---

And then navigate to `http://127.0.0.1:8000/`.



## Templates

Every web framework needs a convenient way to generate HTML files, and in Django, the approach is to use *templates*: individual HTML files that can be linked together and include basic logic.

Recall that our “Hello, World” site had the phrase hardcoded into a `views.py` file in the previous chapter. That technically works but does not scale well! A better approach is to link a view to a template, thereby separating the information contained in each.

In this chapter, we'll learn how to use templates to create our desired homepage and about page. And in future chapters, the use of templates will support building websites that can support hundreds, thousands, or even millions of webpages with a minimal amount of code.

The first consideration is where to place templates within the structure of a Django project. There are two options. By default, Django's template loader will look within each app for related templates. However, the structure is somewhat confusing: each app needs a new `templates` directory, another directory with the same name as the app, and then the template file.

Therefore, in our `pages` app, Django would expect the following layout:



Why this seemingly repetitive approach? The short answer is that the Django template loader wants to be sure it finds the correct template! What happens if there are `home.html` files within two separate apps? This structure makes sure there are no such conflicts.

However, another approach is to instead create a single project-level `templates` directory and place *all* templates within it. By tweaking our `django_project/settings.py` file, we can tell Django to *also* look in this directory for templates. That is the approach we'll use.

First, quit the running server with the `Control+c` command. Then create a directory called `templates`.

## Shell

---

```
(.venv) $ mkdir templates
```

---

Next, we need to update `django_project/settings.py` to tell Django the location of our new `templates` directory. Doing so requires a one-line change to the setting "DIRS" under `TEMPLATES`.

## Code

---

```
# django_project/settings.py
TEMPLATES = [
    {
        ...
        "DIRS": [BASE_DIR / "templates"], # new
        ...
    },
]
```

---

Create a new file called `home.html` within the `templates` directory. You can do this within your text editor: in Visual Studio Code, go to the top left of your screen, click “File,” and then “New File.” Make sure to name and save the file in the correct location.

The `home.html` file will have a simple headline for now.

## Code

---

```
<!-- templates/home.html -->
<h1>Homepage</h1>
```

---

Our template is complete! The next step is to configure our URL and view files.

## Class-Based Views

Early versions of Django only shipped with function-based views, but developers soon found themselves repeating the same patterns over and over. Write a view that lists all objects in a model. Write a view that displays only one

detailed item from a model. And so on. Generic function-based views were introduced to abstract these patterns and streamline the development of common patterns. The problem with generic function-based views was that there was [no easy way to extend or customize them](#). As projects grow in size, this became more and more of an issue and, as a result, generic function-based views were [deprecated in Django 1.3](#) and removed completely in version 1.5.

To help with code reusability, Django added class-based views and generic class-based views while still retaining function-based views. Classes are a fundamental part of Python, but a thorough discussion of them is beyond the scope of this book. If you need an introduction or refresher, I suggest reviewing the [official Python docs](#), which have an excellent tutorial on classes and their usage.

There are, therefore, three different ways to write a view in Django—function-based, class-based, or generic class-based—which is very confusing to beginners. Function-based views are simpler to understand because they mimic the HTTP request/response cycle, and they are a good place to start which is what we've done in this book. Class-based views are a little harder to understand because their inheritance structure means you have to dive into the code to see everything happening; with a function-based view, it is all there. And generic class-based views are the hardest yet to understand. An entire website, [Classy Class-Based Views](#), is dedicated to helping developers decipher them.

So why bother with generic class-based views? Once you have used them for a while, they become elegant and efficient ways to write code. You can often modify a single method on one to do custom behavior rather than rewriting everything from scratch, which makes it easier to understand someone else's code. This does, however, come

at the cost of complexity and requires a leap of faith because it takes a long time to understand how they work under the hood.

That said, most Django developers first reach for generic class-based views when trying to solve a problem. It is incredible how many issues can be solved by off-the-shelf GCVBs like `TemplateView`, `ListView`, or `DetailView`. When a generic class-based view is not enough, modify it to suit your needs. And if that still isn't enough, revert to a function-based or class-based view. That is the recommended approach for handling the three different ways to write views in Django.

In our specific use case, we will use `TemplateView` to display our template. Replace the default text in the `pages/views.py` file with the following:

Code

---

```
# pages/views.py
from django.views.generic import TemplateView

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

---

Note that we've used Pascal Case to name our view, `HomePageView`, meaning the first letter of each word is capitalized. This is used for [class names in Python](#). The `TemplateView` already contains all the logic needed to display our template; we just need to specify the template's name.

## URLs

The last step is to update our URLs. Recall from Chapter 2 that we need to make updates in two locations. First, we update the `django_project/urls.py` file to point at our `pages` app, and then within `pages`, we match views to URL routes.

Let's start with the `django_project/urls.py` file.

Code

---

```
# django_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
]
```

---

The code here should look familiar at this point. We add `include` on the second line to point the existing URL to the `pages` app. Next, create a `pages/urls.py` file and add the code below. This pattern is almost identical to what we did in the last chapter with one significant difference: when using Class-Based Views, you always add `as_view()` at the end of the view name.

Code

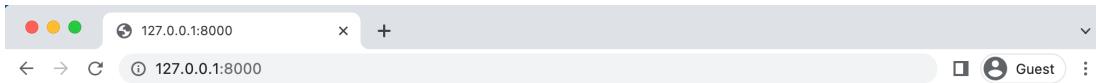
---

```
# pages/urls.py
from django.urls import path
from .views import HomePageView

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

---

And we're done! Start the local web server with the command `python manage.py runserver` and navigate to `http://127.0.0.1:8000/` to see our new homepage.



**Homepage**

**Homepage**

## About Page

The process for adding an About page is very similar to what we just did. We'll create a new template file, a new view, and a new URL route. Start by making a new template file called `templates/about.html` and populate it with a short HTML headline.

Code

---

```
<!-- templates/about.html -->
<h1>About page</h1>
```

---

Then add a new view for the page.

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 finally, import the view name and connect it to a URL at `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"),
]
```

---

Start the web server with `runserver` and navigate to `http://127.0.0.1:8000/about/`. The new About page is visible.



## About page

### About page

## Extending Templates

The real power of templates is that they can be extended. If you think about most websites, the same content appears on every page (header, footer, etc.). Wouldn't it be nice if we, as developers, could have *one canonical place* for our header code that all other templates would inherit?

Well, we can! Let's create a `base.html` file containing a header with links to our two pages. We could name this file anything, but using `base.html` is a standard convention. In your text editor, make this new file called `templates/base.html`.

Django has a minimal templating language for adding links and basic logic in our templates. You can see the complete list of built-in template tags [here in the official docs](#).

Template tags take the form of `{% something %}` where the “something” is the template tag itself. You can even create custom template tags, though we won't do that in this book.

To add URL links in our project, we can use the [built-in URL template tag](#) which takes the URL pattern name as an argument. Remember how we added optional URL names to our two routes in `pages/urls.py`? The `url` tag uses these names to create links for us automatically.

The URL route for our homepage is called `home`. To configure a link to it, we use the following syntax: `{% url 'home' %}`.

---

### Code

---

```
<!-- templates/base.html -->
<header>
  <a href="{% url 'home' %}">Home</a> |
  <a href="{% url 'about' %}">About</a>
</header>

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

---

At the bottom, we've added a block tag called content. While it's optional to name our closing endblock—you can write `{% endblock %}` if you prefer—doing so helps with readability, especially in larger template files.

Let's update our `home.html` and `about.html` files to extend the `base.html` template. That means we can reuse the same code from one template in another. The Django templating language comes with an [extends](#) method that we can use for this.

---

### Code

---

```
<!-- templates/home.html -->
{% extends "base.html" %}

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

---

---

### Code

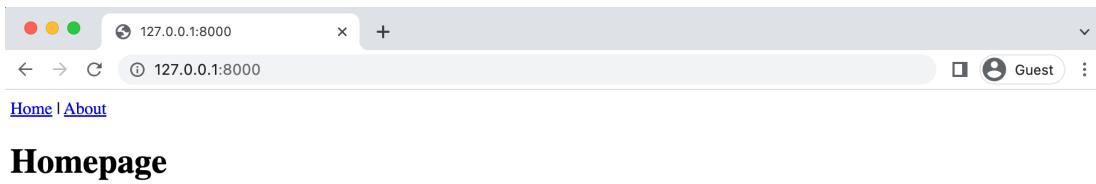
---

```
<!-- templates/about.html -->
{% extends "base.html" %}

{% block content %}
<h1>About page</h1>
{% endblock content %}
```

---

Now if you start up the server with `python manage.py runserver` and open up our webpages again at `http://127.0.0.1:8000/`, the header is included.



And it is also present on the about page at  
[http://127.0.0.1:8000/about/.](http://127.0.0.1:8000/about/)



There's a *lot* more we can do with templates, and in practice, you'll typically create a `base.html` file that can be inherited by other templates to promote code reusability. For example, if the header and footer on a website are the same on all pages, move that code into a dedicated `base.html` file and then only the code *that is different* appears in individual templates. We'll do this later on in the book.

## Tests

It's important to add automated tests and run them whenever a codebase changes. Tests require a small amount of upfront time to write but more than pay off later on. In the words of Django co-creator [Jacob Kaplan-Moss](#), "Code without tests is broken as designed."

Testing can be divided into two main categories: unit and integration. *Unit tests* check a piece of functionality in

isolation, while *Integration tests* check multiple linked pieces. Unit tests run faster and are easier to maintain since they focus on only a tiny amount of code. Integration tests are slower and harder to maintain since a failure doesn't point you in the specific direction of the cause. Most developers focus on writing many unit tests and a small number of integration tests.

The next question is, What to test? Anytime you have created new functionality, a test is necessary to confirm that it works as intended. For example, in our project, we have a home page and an about page, and we should test that both exist at the expected URLs. It may seem unnecessary now but as a project grows in size, tests make sure that you don't inadvertently break something you did previously. By writing tests and regularly running them, you can ensure this doesn't happen.

The Python standard library contains a built-in testing framework called [unittest](#) that uses [TestCase](#) instances and a long list of [assert methods](#) to check for and report failures. Django's testing framework provides several extensions on top of Python's `unittest.TestCase` base class. These include a [test client](#) for making dummy Web browser requests, several Django-specific [additional assertions](#), and four test case classes: [SimpleTestCase](#), [TestCase](#), [TransactionTestCase](#), and [LiveServerTestCase](#).

Generally speaking, `SimpleTestCase` is used when a database is unnecessary while `TestCase` is used when you want to test the database. `TransactionTestCase` is helpful to directly test [database transactions](#) while `LiveServerTestCase` launches a live server thread for testing with browser-based tools like Selenium.

Before we proceed: you may notice that the *naming* of methods in `unittest` and `django.test` are written in `camelCase` rather than the more Pythonic `snake_case` pattern. The reason is that `unittest` is based on the `jUnit` testing framework from Java, which does use `camelCase`, so when Python added `unittest` it came along with `camelCase` naming.

If you look within our `pages` app, Django already provided a `tests.py` file we can use. Since no database is involved in our project, we will import `SimpleTestCase` at the top of the file. For our first tests, we'll check that the two URLs for our website, the homepage and about page, both return [HTTP status codes](#) of 200, the standard response for a successful HTTP request.

---

#### Code

---

```
# pages/tests.py
from django.test import SimpleTestCase

class HomepageTests(SimpleTestCase):
    def test_url_exists_at_correct_location(self):
        response = self.client.get("/")
        self.assertEqual(response.status_code, 200)

class AboutpageTests(SimpleTestCase):
    def test_url_exists_at_correct_location(self):
        response = self.client.get("/about/")
        self.assertEqual(response.status_code, 200)
```

---

To run the tests, quit the server with `Control+c` and type `python manage.py test` on the command line to run them.

---

#### Shell

---

```
(.venv) $ python manage.py test
Found 2 test(s).
System check identified no issues (0 silenced).
.
-----
Ran 2 tests in 0.003s
```

---

OK

---

If you see an error such as `AssertionError: 301 != 200`, you likely forgot to add the trailing slash to `/about` above. The web browser knows to add a slash if it's not provided automatically, but that causes a 301 redirect, not a 200 success response!

What else can we test? At the moment we are testing the actual URL route for each page: `/` for the homepage and `/about` for the about page. But remember that we *also* added a URL name for each in the `pages/urls.py` file. We should check that the URL name works as well.

---

Code

---

```
# pages/urls.py
from django.urls import path
from .views import HomePageView, AboutPageView

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

---

To do that we can use the very handy Django utility function [reverse](#). Instead of going to a URL path first, it looks for the URL name. In general, it is a bad idea to hardcode URLs, especially in templates. By using `reverse` we can avoid this. See a further explanation [here in the docs](#), but we will use `reverse` more in coming chapters.

For now, we want to test the URL names for our two pages. Import `reverse` at the top of the file add then add a new unit test for each below.

---

Code

---

```
# pages/tests.py
from django.test import SimpleTestCase
from django.urls import reverse # new

class HomepageTests(SimpleTestCase):
    def test_url_exists_at_correct_location(self):
        response = self.client.get("/")
```

```
        self.assertEqual(response.status_code, 200)

    def test_url_available_by_name(self): # new
        response = self.client.get(reverse("home"))
        self.assertEqual(response.status_code, 200)

class AboutpageTests(SimpleTestCase):
    def test_url_exists_at_correct_location(self):
        response = self.client.get("/about/")
        self.assertEqual(response.status_code, 200)

    def test_url_available_by_name(self): # new
        response = self.client.get(reverse("about"))
        self.assertEqual(response.status_code, 200)
```

---

Rerun the tests to confirm that they work correctly.

#### Shell

---

```
(.venv) $ python manage.py test
Found 4 test(s).
System check identified no issues (0 silenced).
..
-----
Ran 4 tests in 0.005s
```

---

OK

---

We have tested our URL locations and names so far but not our templates. Let's make sure that the correct templates—`home.html` and `about.html`—are used on each page and that they display the expected content of "`<h1>Homepage</h1>`" and "`<h1>About page</h1>`" respectively.

We can use [assertTemplateUsed](#) and [assertContains](#) to achieve this.

#### Code

---

```
# pages/tests.py
from django.test import SimpleTestCase
from django.urls import reverse

class HomepageTests(SimpleTestCase):
    def test_url_exists_at_correct_location(self):
        response = self.client.get("/")
        self.assertEqual(response.status_code, 200)
```

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

def test_template_name_correct(self): # new
    response = self.client.get(reverse("home"))
    self.assertTemplateUsed(response, "home.html")

def test_template_content(self): # new
    response = self.client.get(reverse("home"))
    self.assertContains(response, "<h1>Homepage</h1>")

class AboutpageTests(SimpleTestCase):
    def test_url_exists_at_correct_location(self):
        response = self.client.get("/about/")
        self.assertEqual(response.status_code, 200)

    def test_url_available_by_name(self):
        response = self.client.get(reverse("about"))
        self.assertEqual(response.status_code, 200)

    def test_template_name_correct(self): # new
        response = self.client.get(reverse("about"))
        self.assertTemplateUsed(response, "about.html")

    def test_template_content(self): # new
        response = self.client.get(reverse("about"))
        self.assertContains(response, "<h1>About page</h1>")
```

---

Run the tests one last time to check our new work.  
Everything should pass.

#### Shell

---

```
(.venv) $ python manage.py test
Found 8 test(s).
System check identified no issues (0 silenced).
.
-----
Ran 8 tests in 0.006s
```

---

OK

---

Experienced programmers may look at our testing code and note that there is a lot of repetition. For example, we set the response each time for all eight tests. Generally, it is a good idea to abide by DRY (Don't Repeat Yourself) coding, but unit tests work best when they are self-contained and highly verbose. As a test suite expands, for performance reasons,

it might make more sense to combine multiple assertions into a smaller number of tests, but that is an advanced-and often subjective-topic beyond the scope of this book.

We'll do much more with testing in the future, especially once we start working with databases. For now, it's important to see how easy and important it is to add tests every time we add new functionality to our Django project.

## Git and GitHub

It's time to track our changes with Git and push them to GitHub. We'll start by initializing our directory and checking the status of our changes.

Shell

---

```
(.venv) $ git init  
(.venv) $ git status
```

---

Use `git status` to see all our code changes. Did you notice that the `.venv` directory with our virtual environment is included? We don't want that. In your text editor, create a new hidden file in the root directory, `.gitignore`, so we can specify what Git will *not* track. And while we're at it, we can tell Git to ignore the `__pycache__` directory, which contains bytecode compiled and ready to be executed.

`.gitignore`

---

```
.venv/  
__pycache__/
```

---

Run `git status` again to confirm these two files are being ignored. You may have noticed we did not yet create a `requirements.txt` file as we did in the “Hello, World!” project in the last chapter. Hang tight; we will be adding this shortly as part of the deployment process. Use `git add -A` to add the

intended files/directories and include an initial commit message.

#### Shell

---

```
(.venv) $ git status  
(.venv) $ git add -A  
(.venv) $ git commit -m "initial commit"
```

---

Over on GitHub [create a new repo](#) called pages and make sure to select the “Private” radio button. Then click on the “Create repository” button.

On the next page, scroll down to where it says “...or push an existing repository from the command line.” Copy and paste the two commands there into your terminal.

It should look like the below, albeit instead of wsvincent as the username, it will be your GitHub username.

#### Shell

---

```
(.venv) $ git remote add origin https://github.com/wsvincent/pages.git  
(.venv) $ git branch -M main  
(.venv) $ git push -u origin main
```

---

## Hosting Options

To make our site available online where everyone can see it, we need to deploy our code to an external server and database. Local code lives only on our computer; production code lives on an external server available to everyone. Therefore, deploying code online is also called, *putting our code into production*.

There are three main ways to host a website:

- **1. Dedicated Server:** a physical server sitting in a data center that belongs exclusively to you. Generally, only the largest companies adopt this approach since it

requires a lot of technical expertise to configure and maintain.

- **2. Virtual Private Server (VPS):** a server can be divided into multiple virtual machines that use the same hardware, which is much more affordable than a dedicated server. This approach also means you don't have to worry about maintaining the hardware yourself.
- **3. Platform as a Service (PaaS):** a managed VPS solution that is pre-configured and maintained, making it the fastest option to deploy and scale a website. The downside is that a developer does not have access to the same degree of customization that a VPS or a dedicated server provides.

These days most individual developers and small companies rely on PaaS's for their hosting needs since it abstracts away a challenging part of website development. That is the approach we will use in this book, too.

## Deployment Checklist

Django defaults to a local development configuration with the `startproject` command, which makes it easier to start building apps locally, but many settings are a security risk in production.

Since deployment requires many discrete steps, it is common to have a “checklist” for these since there quickly become too many to remember. At this stage, we are intentionally keeping things basic however, this list will grow in future projects as we add additional security and performance features.

Here is the current deployment checklist:

- install Gunicorn as a WSGI server

- generate a requirements.txt file
- update ALLOWED\_HOSTS in django\_project/settings.py
- add a .dockerignore file

The first step is to install [Gunicorn](#), a production-ready WSGI server for our project. Gunicorn replaces the Django local development server and-as with all Python packages-is installable via Pip.

#### Shell

---

```
(.venv) $ python -m pip install gunicorn==20.1.0
```

---

Step two is to create a requirements.txt file containing every Python package currently installed in our virtual environment. This file is necessary so that the project can be recreated on production servers and other team members (if any) can recreate the repository from scratch in the future.

We can use the command `python -m pip freeze` to output all the installed packages to the terminal. Then we *could* create a root project-level requirements.txt file and copy/paste the output over. Or we can take a more elegant approach by redirecting the output of `pip freeze` over to the requirements.txt file automatically. The greater-than symbol (>) lets us redirect the output to a file. Here is what the full command looks like. Run it.

#### Shell

---

```
(.venv) $ python -m pip freeze > requirements.txt
```

---

If you look in your text editor, a new requirements.txt file has been created listing each package in our virtual environment. You do not need to focus too much on the contents of this file, just remember that whenever you install a new Python package in your virtual environment,

you should update the requirements.txt file to reflect it. There are more complicated tools that automate this process, but since we are using venv and keeping things as simple as possible, it must be done manually.

The third step is to look within django\_project/settings.py for the [ALLOWED\\_HOSTS](#) setting, representing which host/domain names our Django site can serve. This is a security measure to prevent HTTP Host header attacks. For now, we'll use the wildcard asterisk, \*, so that *all* domains are acceptable. Later in the book, we'll adopt a more secure approach of explicitly listing allowed domains.

Code

---

```
# django_project/settings.py
ALLOWED_HOSTS = ["*"]
```

---

The fourth and final step is to create a .dockerignore file in the project root directory, next to the existing .gitignore file. [Docker](#) is used by most modern hosting companies-more on this shortly-and it is essential to tell it what to ignore. The .dockerignore file does this for us. At a minimum, we want to ignore our local virtual environment, SQLite database, and Git repo.

.dockerignore

---

```
.venv/
*.sqlite3
.git
```

---

That's it! Remember that we've taken several security shortcuts here, but the goal is to push our project into production in as few steps as possible. In future chapters, we will cover proper security around deployments in depth.

Use git status to check our changes, add the new files, and then commit them:

Shell

---

```
(.venv) $ git status  
(.venv) $ git add -A  
(.venv) $ git commit -m "New updates for deployment"
```

---

Push the code to GitHub to have an online backup of all our code changes.

Shell

---

```
(.venv) $ git push -u origin main
```

---

## Fly.io

Until recently, Heroku was the default choice for simple Django deployments because it was free for small projects and widely used. Unfortunately, Heroku announced that as of November 2022, it would cease offering free tiers. Running a hosting company requires actual costs and efforts to prevent fraud, and it is not unreasonable to expect to pay something to host a website.

For this book, we will use [Fly.io](#). Sign up for an account on their website using your email address or GitHub account. Adding a payment method is now required.

## Deploy to Fly.io

Fly has its own *Command Line Interface (CLI)* to help manage deployments. [Follow the official instructions](#) to install it. Then from the command line [sign in](#) with the command `flyctl auth login`, which will likely open up a web browser window for you to sign into Fly.

Shell

---

```
(.venv) $ flyctl auth login  
Opening https://fly.io/app/auth/cli/606daf31a9c91b62e5528e27ee891e4e ...
```

```
Waiting for session... Done  
successfully logged in as will@wsvincent.com
```

---

To configure and launch our site run the `fly launch` command and follow the wizard.

- **Choose an app name:** this will be your dedicated `fly.dev` subdomain
- **Choose the region for deployment:** select the one closest to you or [another region](#) if you prefer
- **Decline overwriting our `.dockerignore` file:** our choices are already optimized for the project
- **Decline to setup a Postgres or Redis database:** we will tackle this in the next project

---

#### Shell

```
(.venv) $ fly launch
Creating app in ~/desktop/code/pages
Scanning source code
Detected a Django app
? Choose an app name (leave blank to generate one): dfb-ch3-pages
automatically selected personal organization: Will Vincent
? Choose a region for deployment: Boston, Massachusetts (US) (bos)
App will use 'bos' region as primary
Created app dfb-ch3-pages in organization 'personal'
Admin URL: https://fly.io/apps/dfb-ch3-pages
Hostname: dfb-ch3-pages.fly.dev
? Overwrite "/Users/wsv/Desktop/dfb_42/ch3-pages/.dockerignore"? No
Set secrets on dfb-ch3-pages: SECRET_KEY
? Would you like to set up a Postgresql database now? No
? Would you like to set up an Upstash Redis database now? No
Wrote config file fly.toml
Your app is ready! Deploy with `flyctl deploy`
```

---

In your project directory, Fly created two new files: `fly.toml` and `Dockerfile`. The [fly.toml](#) is a Fly-specific configuration file while the [Dockerfile](#) contains instructions for creating a Docker image on Fly servers.

Now that everything is configured, run the command `flyctl deploy` to deploy our project on Fly servers.

---

#### Shell

```
(.venv) $ flyctl deploy
...
```

```
1 desired, 1 placed, 1 healthy, 0 unhealthy [health checks: 1 total, 1
passing]
--> v0 deployed successfully
```

---

This initial deploy will take a few seconds to run as it uploads your application, verifies the app configuration, builds the image, and then monitors to ensure it starts successfully. Once complete, visit your app with the following command to open your custom URL in a web browser:

#### Shell

---

```
(.venv) $ fly open
```

---

**Note:** If you are wondering why some commands start with `flyctl` and others with `fly`, you are not alone. Initially, all Fly commands required `flyctl`, but many work just as well with simply `fly` at this point. When in doubt, use `flyctl`, but often `fly` will suffice.

If your application didn't boot on the first deployment, run `fly logs` to see what is happening. This command shows the past few log file entries and tails your production log files. [Additional flags](#) are available for filtering.

As a brief recap of the Fly deployment process:

- our Django website is running on a Docker virtual machine created via the `Dockerfile` image
- the `fly.toml` file controls our website configuration and can be modified as needed
- `fly dashboard` opens a web browser-based view to monitor and adjust our deployment as needed

## Conclusion

Congratulations on building and deploying your second Django project! This time we used templates, class-based views, explored URLs more fully, added basic tests, and used Fly for deployment. If you feel overwhelmed by the deployment process, don't worry. Deployment *is* hard even with a tool like Fly. The good news is the steps required are the same for most projects, so you can refer to a deployment checklist to guide you each time you launch a new project.

The complete source code for this chapter is [available on GitHub](#) if you need a reference. In the next chapter, we'll move on to our first database-backed project, a *Message Board* website, and see where Django truly shines.

# Chapter 4: Message Board App

In this chapter, we will use a database for the first time to build a basic *Message Board* application where users can post and read short messages. We'll explore Django's powerful built-in admin interface, which provides a visual way to change our data. And after adding tests, we will push our code to GitHub and deploy the app on Fly.

Thanks to the powerful Django ORM (Object-Relational Mapper), there is built-in support for multiple database backends: PostgreSQL, MySQL, MariaDB, Oracle, and SQLite. As a result, developers can write the same Python code in a `models.py` file, which will automatically be translated into the correct SQL for each database. The only configuration required is to update the [Databases](#) section of our `dango_project/settings.py` file: truly an impressive feature!

For local development, Django defaults to using [SQLite](#) because it is file-based and, therefore, far more straightforward to use than the other database options that require a dedicated server to run separately from Django itself.

## Initial Set Up

Since we've already set up several Django projects in the book, we can quickly run through the standard commands to begin a new one. We need to do the following:

- make a new directory for our code called `message-board` on the desktop
- set up a new Python virtual environment and activate it

In a new command line console, enter the following commands:

---

#### Shell

---

```
# Windows
$ cd onedrive\desktop\code
$ mkdir message-board
$ cd message-board
$ python -m venv .venv
$ .venv\Scripts\Activate.ps1
$ (.venv)

# macOS
$ cd ~/desktop/code
$ mkdir message-board
$ cd message-board
$ python3 -m venv .venv
$ source .venv/bin/activate
$ (.venv)
```

---

Then finish the setup by doing the following:

- install Django and Black in the new virtual environment
- create a new project called `django_project`
- create a new app called `posts`

---

#### Shell

---

```
(.venv) $ python -m pip install django~=4.2.0
(.venv) $ python -m pip install black
(.venv) $ django-admin startproject django_project .
(.venv) $ python manage.py startapp posts
```

---

As a final step, update `django_project/settings.py` to alert Django to the new app, `posts`, by adding it to the bottom of the `INSTALLED_APPS` section.

---

#### Code

---

```
# django_project/settings.py
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
```

```
] "posts", # new
```

---

Then execute the `migrate` command to create an initial database based on Django's default settings.

Shell

---

```
(.venv) $ python manage.py migrate
```

---

A `db.sqlite3` file is created the first time you run *either* `migrate` or `runserver`. However, `migrate` will sync the database with the current state of any database models contained in the project and listed in `INSTALLED_APPS`. In other words, to ensure the database reflects the current state, you'll need to run `migrate` (and also `makemigrations`) each time you update a model. More on this shortly.

To confirm everything works correctly, spin up our local server.

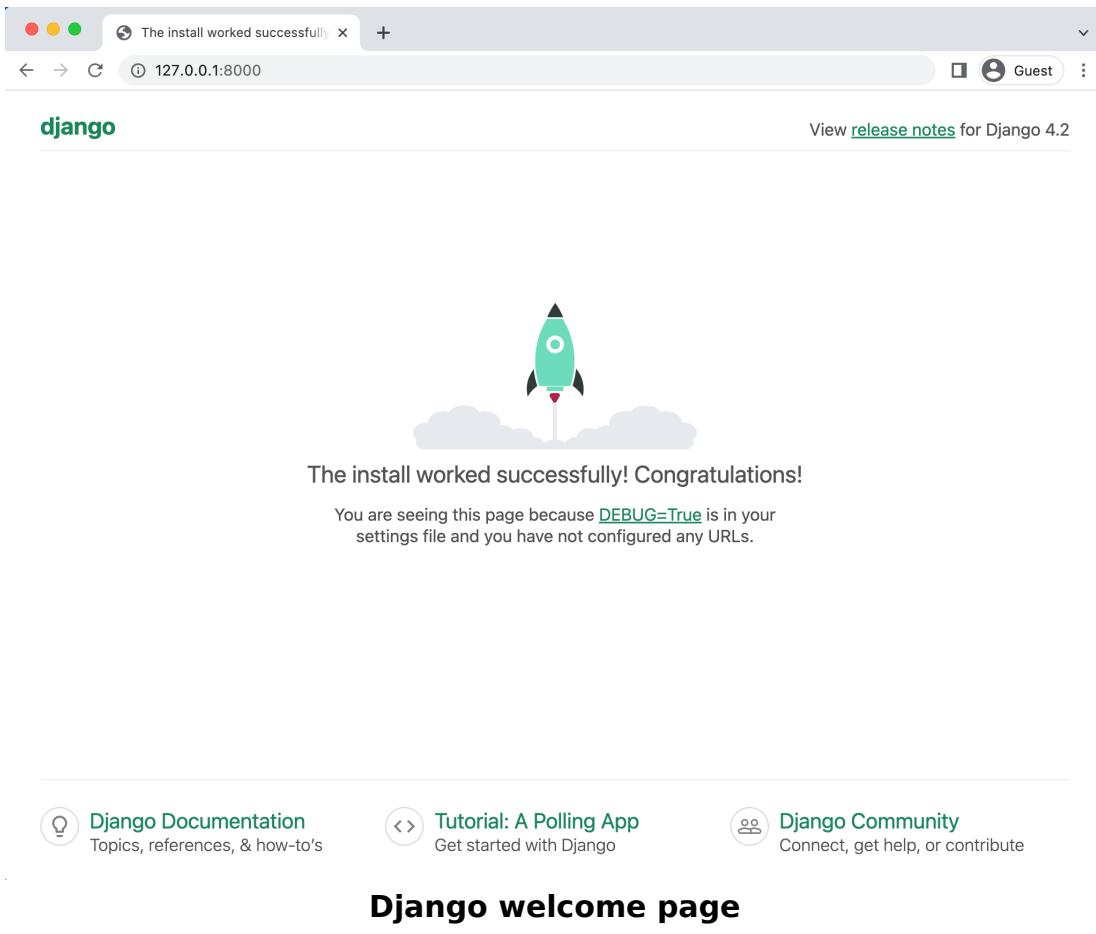
Shell

---

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

---

In your web browser, navigate to `http://127.0.0.1:8000/` to see the familiar Django welcome page.



## Create a Database Model

Our first task is to create a database model where we can store and display posts from our users. Django's ORM will automatically turn this model into a database table for us. In a real-world Django project, there are often many complex, interconnected database models, but we only need one in our simple message board app.

Open the `posts/models.py` file and look at the default code which Django provides:

Code

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

# Create your models here

---

Django imports a module, `models`, to help us build new database models which will “model” the characteristics of the data in our database. For each database model we want to create, the approach is to subclass (meaning to extend) `djangodb.models.Model` and then add our fields. To store the textual content of a message board post, we can do the following:

Code

---

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

class Post(models.Model): # new
    text = models.TextField()
```

---

Note that we’ve created a new database model called `Post`, which has the database field `text`. We’ve also specified the *type of content* it will hold, `TextField()`. Django provides many [model fields](#) supporting common types of content such as characters, dates, integers, emails, and so on.

## Activating models

Now that our new model has been created, we need to activate it. From now on, whenever we make or modify an existing model, we’ll need to update Django in a two-step process:

1. First, we create a migrations file with the `makemigrations` command. Migration files create a reference of any changes to the database models, which means we can track changes—and debug errors as necessary—over time.

2. Second, we build the database with the `migrate` command, which executes the instructions in our migrations file.

Ensure the local server is stopped by typing `Control+c` on the command line and then run the commands `python manage.py makemigrations posts` and `python manage.py migrate`.

#### Shell

---

```
(.venv) $ python manage.py makemigrations posts
Migrations for 'posts':
  posts/migrations/0001_initial.py
    - Create model Post

(.venv) $ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, posts, sessions
Running migrations:
  Applying posts.0001_initial... OK
```

---

You don't have to include a name after `makemigrations`. If you just run `makemigrations` without specifying an app, a migrations file will be created for *all* available changes throughout the Django project. That is fine in a small project like ours with only a single app, but most Django projects have more than one app! Therefore if you made model changes in multiple apps, the resulting migrations file would include *all* those changes: not ideal! Migrations files should be as small and concise as possible, making it easier to debug in the future or even roll back changes as needed. Therefore, as a best practice, adopt the habit of always including the name of an app when executing the `makemigrations` command!

## Django Admin

One of Django's killer features is its robust admin interface that visually interacts with data. It came about because [Django started off](#) as a newspaper CMS (Content

Management System). The idea was that journalists could write and edit their stories in the admin without needing to touch “code.” Over time, the built-in admin app has evolved into a fantastic, out-of-the-box tool for managing all aspects of a Django project.

To use the Django admin, we must first create a superuser who can log in. In your command line console, type `python manage.py createsuperuser` and respond to the prompts for a username, email, and password:

---

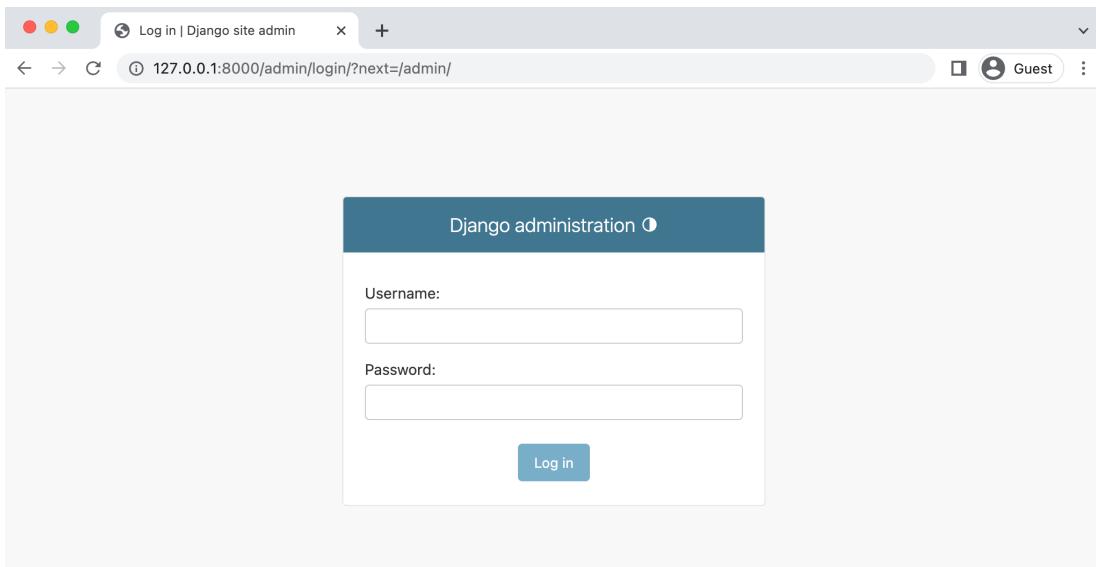
#### Shell

---

```
(.venv) $ python manage.py createsuperuser
Username (leave blank to use 'wsv'): wsv
Email: will@wsvincent.com
Password:
Password (again):
Superuser created successfully.
```

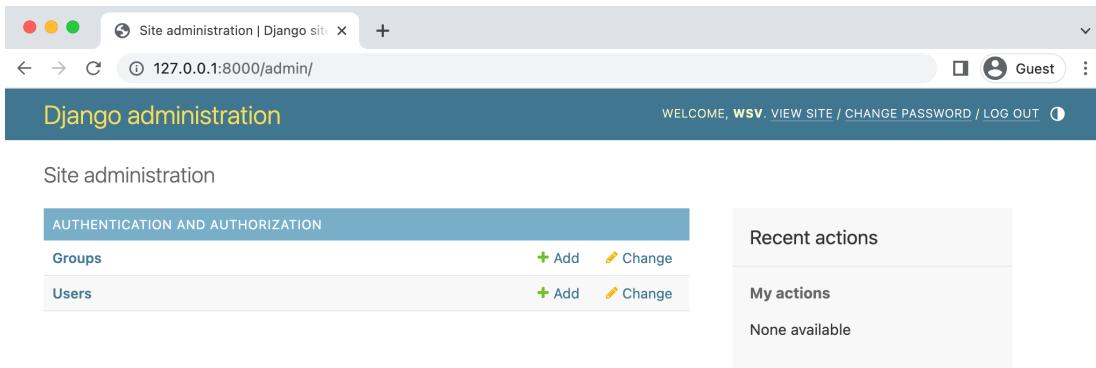
---

When you type your password, it will not appear visible in the command line console for security reasons. For local development, I often use `testpass123`. Restart the Django server with `python manage.py runserver` and, in your web browser, go to `http://127.0.0.1:8000/admin/`. You should see the login screen for the admin:



## Admin login page

Log in by entering the username and password you just created. You will see the Django admin homepage next:



## Admin homepage

Django has impressive support for multiple language, so if you'd like to see the admin, forms, and other default messages in a language other than English, try adjusting the [LANGUAGE\\_CODE](#) configuration in `django_project/settings.py` which is automatically set to American English, en-us.

But where is our posts app since it is not displayed on the main admin page? Just as we must explicitly add new apps to the `INSTALLED_APPS` config, so, too, must we update an app's `admin.py` file for it to appear in the admin.

In your text editor, open up `posts/admin.py` and add the following code to display the Post model.

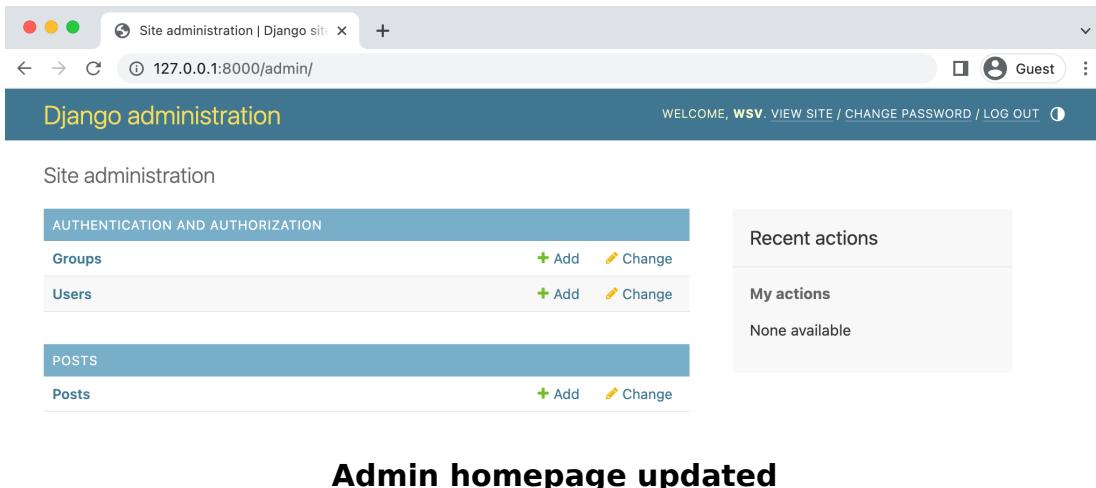
#### Code

```
# posts/admin.py
from django.contrib import admin

from .models import Post

admin.site.register(Post)
```

Django now knows it should display our posts app and its database model `Post` on the admin page. If you refresh your browser, you'll see that it appears:



The screenshot shows the Django Admin homepage. At the top, there's a header bar with the title "Site administration | Django site", the URL "127.0.0.1:8000/admin/", and a "Guest" user indicator. Below the header is a navigation bar with links for "Django administration", "WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT", and a "Guest" link. The main content area is titled "Site administration". It contains two sections: "AUTHENTICATION AND AUTHORIZATION" (Groups, Users) and "POSTS" (Posts). Each section has "Add" and "Change" buttons. To the right, there's a sidebar titled "Recent actions" which says "None available". At the bottom of the page, a banner displays the message "Admin homepage updated".

Let's create our first message board post for our database. Click the + Add button opposite Posts and enter your content in the Text form field.

Django administration

Home > Posts > Posts > Add post

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

POSTS

Posts + Add

Add post

Text:

Hello, World!

«

SAVE Save and add another Save and continue editing

### Admin new entry

Then click the “Save” button to redirect you to the main Post page. However, if you look closely, there’s a problem: our new entry is called “Post object (1)”, which isn’t very descriptive!

Select post to change

ADD POST +

Action: ----- Go 0 of 1 selected

POST

Post object (1)

1 post

### Admin posts list

Let’s change that. Within the `posts/models.py` file, add a new method called `str`, which provides a human-readable

representation of the model. In this case, we'll have it display the first 50 characters of the text field.

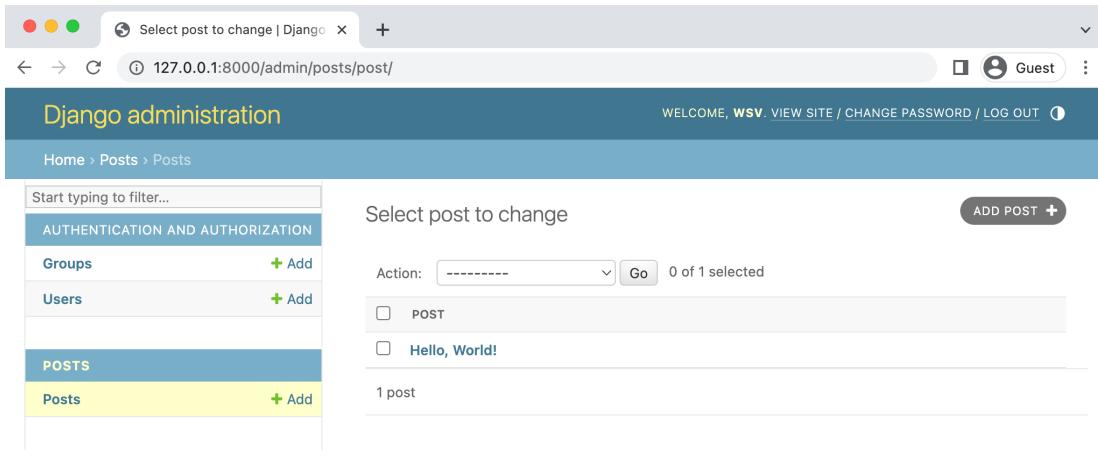
### Code

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

class Post(models.Model):
    text = models.TextField()

    def __str__(self): # new
        return self.text[:50]
```

If you refresh your Admin page in the browser, you'll see it's changed to a much more descriptive and helpful representation of our database entry.



The screenshot shows the Django Admin interface for the 'Posts' model. The title bar says 'Select post to change | Django'. The URL is 127.0.0.1:8000/admin/posts/post/. The main area has a blue header 'Django administration' and a sub-header 'Home > Posts > Posts'. On the left, there's a sidebar with 'Start typing to filter...' and sections for 'AUTHENTICATION AND AUTHORIZATION' (Groups, Users) and 'POSTS' (Posts). Under 'POSTS', 'Posts' is selected and highlighted in yellow. The main content area is titled 'Select post to change' and shows a table with one row. The table has columns for 'Action' (dropdown menu), 'POST' (checkbox), and 'Hello, World!' (post title). Below the table, it says '1 post'. There's also a 'ADD POST' button with a plus sign.

### Admin posts readable

Much better! It's a best practice to add `__str__()` methods to all your models to improve their readability.

## Views/Templates/URLs

In order to display our database content on our homepage, we have to wire up our views, templates, and URLs. This pattern should start to feel familiar now.

Let's begin with the view. Earlier in the book, we used the built-in generic [TemplateView](#) to display a template file on our homepage. Now we want to list the contents of our database model. Fortunately, this is also a common task in web development, and Django comes equipped with the generic class-based [ListView](#).

In the `posts/views.py` file, replace the default text and enter the Python code below:

Code

---

```
# posts/views.py
from django.views.generic import ListView
from .models import Post

class HomePageView(ListView):
    model = Post
    template_name = "home.html"
```

---

On the first line, we're importing `ListView` and in the second line, we import the `Post` model. In the view, `HomePageView`, we subclass `ListView` and specify the correct model and template.

Our view is complete, meaning we still need to configure our URLs and make our template. Let's start with the template. Create a new project-level directory called `templates`.

Shell

---

```
(.venv) $ mkdir templates
```

---

Then update the `DIRS` field in our `django_project/settings.py` file so that Django can look in this new `templates` directory.

Code

---

```
# django_project/settings.py
TEMPLATES = [
{
    ...
    "DIRS": [BASE_DIR / "templates"], # new
```

```
        },
    ]
```

---

In your text editor, create a new file called `templates/home.html`. ListView automatically returns to us a context variable called `<model>_list`, where `<model>` is our model name, that we can loop over via the built-in `for` template tag. We'll create a variable called `post` and can then access the desired field we want to be displayed, `text`, as `post.text`.

Code

---

```
<!-- templates/home.html -->
<h1>Message board homepage</h1>
<ul>
  {% for post in post_list %}
    <li>{{ post.text }}</li>
  {% endfor %}
</ul>
```

---

The last step is to set up our URLs. Let's start with the `django_project/urls.py` file, where we include our `posts` app and add `include` on the second line.

Code

---

```
# django_project/urls.py
from django.contrib import admin
from django.urls import path, include # new

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

---

Then in your text editor, create a new `urls.py` file within the `posts` app and update it like so:

Code

---

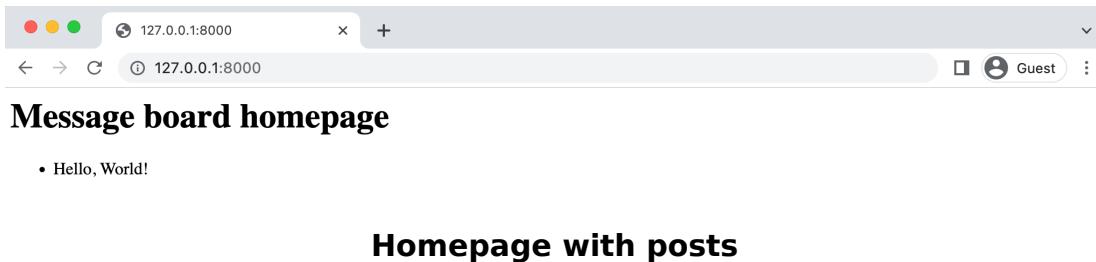
```
# posts/urls.py
from django.urls import path
from .views import HomePageView

urlpatterns = [
```

```
    path("", HomePageView.as_view(), name="home"),
]
```

---

Restart the server with `python manage.py runserver` and navigate to our homepage, which now lists our message board post.



We're basically done at this point, but let's create a few more message board posts in the Django admin to confirm that they will display correctly on the homepage.

## Adding New Posts

To add new posts to our message board, go to the admin and create two more posts. If you then return to the homepage, you'll see it automatically displays the formatted posts. Woohoo!

Everything works, so it is a good time to initialize our directory and create a `.gitignore` file. We can initialize a new Git repository from the command line with the `git init` command.

Shell

---

```
(.venv) $ git init
```

---

Then in your text editor, create a new `.gitignore` file in the root directory and add three lines so that the `.venv` directory, Python bytecode, and the `db.sqlite` file are not stored. The

local database is just for testing purposes anyway, not for production, and in addition to becoming quite large might contain sensitive information you do not want to be stored in a remote code repository. Therefore it is a best practice not to track it with Git.

---

.gitignore

---

.venv/  
\_\_pycache\_\_/  
\*.sqlite3

---

If you now run git status the .venv directory and the db.sqlite3 file are ignored. Use git add -A to add the intended files/directories and then write an initial commit message.

---

Shell

---

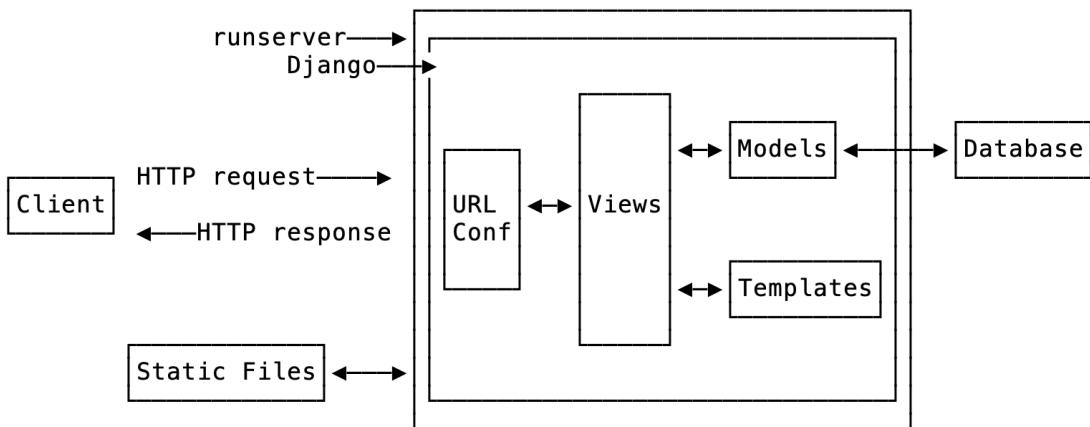
(.venv) \$ git status  
(.venv) \$ git add -A  
(.venv) \$ git commit -m "initial commit"

---

## Static Files

[Static files](#) are the Django community's term for additional files commonly served on websites such as CSS, fonts, images, and JavaScript. Even though we haven't added any yet to our project, we are already relying on core Django static files—custom CSS, fonts, images, and JavaScript—to power the look and feel of the Django admin.

We don't have to worry about static files for local development because the web server—run via the runserver command—will automatically find and serve them for us. Here is what static files look like in visual form added to our existing Django diagram:



**Django request/response cycle with static files**

In production, things are more complex, something that we will cover in the next chapter when we deploy this project. The central concept to understand right now is that it is far more efficient to combine all static files across a Django project into a single location in production. If you look near the bottom of the existing `django_project/settings.py` file, there is already a configuration for [`STATIC\_URL`](#), which refers to the *URL location* of all static files in production. In other words, if our website had the URL `example.com`, all static files would be available in `example.com/static`.

Code

---

```
# django_project/settings.py
STATIC_URL = "static/"
```

---

The built-in management command, [`collectstatic`](#), performs this task of compiling all static files into one location on our file system. The location of the static files in our file system is set via [`STATIC\_ROOT`](#). While we have the flexibility to name this new directory anything we want, traditionally, it is called `staticfiles`. Here's how to set things up in our project:

Code

---

```
# django_project/settings.py
STATIC_URL = "static/"
```

---

```
STATIC_ROOT = BASE_DIR / "staticfiles" # new
```

---

Now run `python manage.py collectstatic` on the command line, and compile all of our project's static files into a new root-level directory called `staticfiles`.

#### Shell

---

```
(.venv) $ python manage.py collectstatic  
125 static files copied to '~/desktop/code/message-board/staticfiles'.
```

---

If you inspect the new `staticfiles` directory, it contains a single directory at the moment, `admin`, from the built-in admin app powered by its own CSS, images, and Javascript.

#### Code

---

```
# staticfiles/  
└── admin  
    ├── css  
    ├── img  
    └── js
```

---

These steps mean that when we deploy our website into production, the admin will work and display as expected. Expect much more on static files in future chapters, but for now, this will suffice.

## Tests

In the previous chapter, we were only testing static pages, so we used [SimpleTestCase](#). Now that our project works with a database, we need to use [TestCase](#), which will let us create a test database. In other words, we don't need to run tests on our *actual* database but instead can make a separate test database, fill it with sample data, and then test against it, which is a much safer and more performant approach.

Our Post model contains only one field, text, so let's set up our data and then check that it is stored correctly in the database. All test methods must start with the phrase `test` so that Django knows to test them!

We will use the hook [`setUpTestData\(\)`](#) to create our test data: it is much faster than using the `setUp()` hook from Python's unittest because it creates the test data only once per test case rather than per test. It is still common, however, to see Django projects that rely on `setUp()` instead. Converting any such tests over to `setUpTestData` is a reliable way to speed up a test suite and should be done!

`setUpTestData()` is a [`classmethod`](#) which means it is a method that can transform into a class. To use it, we'll use the `@classmethod` function decorator. As [PEP 8 explains](#), in Python it is a best practice to always use `cls` as the first argument to class methods. Here is what the code looks like:

Code

---

```
# posts/tests.py
from django.test import TestCase

from .models import Post

class PostTests(TestCase):
    @classmethod
    def setUpTestData(cls):
        cls.post = Post.objects.create(text="This is a test!")

    def test_model_content(self):
        self.assertEqual(self.post.text, "This is a test!")
```

---

At the top, we import `TestCase` and our `Post` model. Then we create a test class, `PostTests`, that extends `TestCase` and uses the built-in method `setUpTestData` to develop initial data. In this instance, we only have one item stored as `cls.post` that can be referred to in subsequent tests within the class as `self.post`. Our first test, `test_model_content`, uses `assertEqual` to

check that the content of the text field matches what we expect.

Run the test on the command line with command `python manage.py test`.

#### Shell

---

```
(.venv) $ python manage.py test
Found 1 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

.
-----
Ran 1 test in 0.001s

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

---

It passed! Why does the output say only one test ran when we have two functions? Again, only functions that start with the name `test` will be run! So while we can use set-up functions and classes to help with our tests, unless a function is named correctly it won't be executed with the `python manage.py test` command.

Moving along, it is time to check our URLs, views, and templates like the previous chapter. We will want to check the following four things for our message board page:

- URL exists at / and returns a 200 HTTP status code
- URL is available by its name of “home”
- Correct template is used called “home.html”
- Homepage content matches what we expect in the database

We can include all of these tests in our existing `PostTests` class since this project has only one webpage. Make sure to import `reverse` at the top of the page and add the four tests as follows:

## Code

---

```
# posts/tests.py
from django.test import TestCase
from django.urls import reverse # new

from .models import Post

class PostTests(TestCase):
    @classmethod
    def setUpTestData(cls):
        cls.post = Post.objects.create(text="This is a test!")

    def test_model_content(self):
        self.assertEqual(self.post.text, "This is a test!")

    def test_url_exists_at_correct_location(self): # new
        response = self.client.get("/")
        self.assertEqual(response.status_code, 200)

    def test_url_available_by_name(self): # new
        response = self.client.get(reverse("home"))
        self.assertEqual(response.status_code, 200)

    def test_template_name_correct(self): # new
        response = self.client.get(reverse("home"))
        self.assertTemplateUsed(response, "home.html")

    def test_template_content(self): # new
        response = self.client.get(reverse("home"))
        self.assertContains(response, "This is a test!")
```

---

If you rerun our tests again you should see that they all pass.

## Shell

---

```
(.venv) $ python manage.py test
Found 5 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
-----
Ran 5 tests in 0.006s

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

---

In the previous chapter, we discussed how unit tests work best when they are self-contained and highly verbose. However, there is an argument to be made here that the

bottom three tests are just testing that the homepage works as expected: it uses the correct URL name, the intended template name, and contains expected content. We can combine these three tests into one unit test, `test_homepage`.

---

#### Code

---

```
# posts/tests.py
from django.test import TestCase
from django.urls import reverse # new
from .models import Post

class PostTests(TestCase):
    @classmethod
    def setUpTestData(cls):
        cls.post = Post.objects.create(text="This is a test!")

    def test_model_content(self):
        self.assertEqual(self.post.text, "This is a test!")

    def test_url_exists_at_correct_location(self):
        response = self.client.get("/")
        self.assertEqual(response.status_code, 200)

    def test_homepage(self): # new
        response = self.client.get(reverse("home"))
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, "home.html")
        self.assertContains(response, "This is a test!")
```

---

Run the tests one last time to confirm that they all pass.

---

#### Shell

---

```
(.venv) $ python manage.py test
Found 3 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.

Ran 3 tests in 0.005s

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

---

Ultimately, we want our test suite to cover as much code functionality as possible yet remain easy for us to reason about. In my view, this update is easier to read and understand.

That's enough tests for now; it's time to commit the changes to Git.

---

#### Shell

---

```
(.venv) $ git add -A  
(.venv) $ git commit -m "added tests"
```

---

## GitHub

We also need to store our code on GitHub. You should already have a GitHub account from previous chapters, so go ahead and create a new repo called `message-board`. Select the “Private” radio button.

On the next page, scroll down to where it says, “or push an existing repository from the command line.” Copy and paste the two commands there into your terminal, which should look like the below after replacing `wsvincent` (my username) with your GitHub username:

---

#### Shell

---

```
(.venv) $ git remote add origin https://github.com/wsvincent/message-board.git  
(.venv) $ git branch -M main  
(.venv) $ git push -u origin main
```

---

## Conclusion

We've now built and tested our first database-driven app and learned how to create a database model, update it with the admin panel, and display the contents on a webpage. Static files were briefly introduced and will be the subject of more coverage later in the book. In the next chapter, we will deploy the message board app and deepen our understanding by adding environment variables, connecting to a production PostgreSQL database, and addressing security concerns around `ALLOWED_HOSTS`.

# Chapter 5: Message Board Deployment

In this chapter, we will deploy our *Message Board* app. Although this is our second deployment in the book, it is our first time focusing on configuring a production database properly. We'll rely on environment variables to toggle seamlessly between local and production environments to assist in that effort while providing greater overall security. Let's begin.

## Previous Deployment Checklist

Deployment involves so many discrete steps that it is easy to lose track of them all. As a result, deployment checklists are commonly used so no step is missed. For our previous deployment checklist we only had four steps:

- install Gunicorn as a WSGI server
- generate a requirements.txt file
- update ALLOWED\_HOSTS in django\_project/settings.py
- add a .dockerignore file

We will need to perform all four again, so let's start by working through the initial deployment checklist.

Gunicorn is a production replacement for Django's local WSGI server. Since it is just a Python third-party package, we can install it with pip.

Shell

---

```
(.venv) $ python -m pip install gunicorn==20.1.0
```

---

A requirements.txt file is a snapshot of our virtual environment. In one elegant command, we can output the contents of our virtual environment to a dedicated requirements.txt file in the root project-level directory.

Shell

---

```
(.venv) $ python -m pip freeze > requirements.txt
```

---

ALLOWED\_HOSTS is a list of the host/domain names that our Django website can serve. Eventually, we will lock this down and restrict access, but for now, it is ok to repeat our past step of using a wildcard asterisk, \*, meaning all domains are acceptable.

Code

---

```
# django_project/settings.py
ALLOWED_HOSTS = ["*"]
```

---

The fourth step is adding a .dockerignore file in the root project-level directory (next to requirements.txt and manage.py) that tells Docker what to ignore, notably the local virtual environment, any Python bytecode, the SQLite database, and our Git repository. Create this new file in your text editor and add the content below.

.dockerignore

---

```
.venv/
__pycache__/
*.sqlite3
.git
```

---

Are we all done now? Not quite! Remember, we are dealing with a database now, and so somehow, we must figure out how to use SQLite locally but PostgreSQL (or other Django-supported databases) in production. But before configuring our database, we need to be introduced to environment variables.

# Environment Variables

An environment variable is a variable whose value is set outside the current program and can be loaded in at runtime. In other words, we can store these variables in a secure location but load them into our Django project, which helps us in two major ways. First, it keeps information secure such as our passwords, API keys, and so on. Second, it allows us to rely on a single `settings.py` file to load in either local or production environment variables. An older Django technique was to have multiple `settings.py` files for each configuration, but with environment variables we only need one.

Let's see this in action. There are multiple third-party packages that provide environment variables, but we will use [environs](#) because it is well-maintained and comes with additional Django support. Go ahead and install it now: include the double quotes, "", to install the Django extension.

---

## Shell

```
(.venv) $ python -m pip install "environs[django]==9.5.0"
```

---

Then add three new lines to the top of our `django_project/settings.py` file.

---

## Code

```
# django_project/settings.py
from pathlib import Path
from environs import Env # new

env = Env() # new
env.read_env() # new
```

---

# DATABASES

The only environment variable we'll use right now is for the DATABASES configuration in the django\_project/settings.py file. Remember, we want a way to use SQLite locally but PostgreSQL in production. This is what currently exists: the ENGINE says to use SQLite, and the NAME points to its location.

Code

---

```
# django_project/settings.py
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.sqlite3",
        "NAME": BASE_DIR / "db.sqlite3",
    }
}
```

---

It turns out that most PaaS, including Fly, will automatically set a DATABASE\_URL environment variable inspired by the [Twelve-Factor App](#) approach that contains all the parameters needed to connect to a database in the format. In raw form, for PostgreSQL, it looks something like this:

Code

---

```
postgres://USER:PASSWORD@HOST:PORT/NAME
```

---

In other words, use postgres and here are custom values for USER, PASSWORD, HOST, PORT/NAME.

While we *could* manage this manually ourselves, this pattern is so well established in the Django community that a dedicated third-party package, [dj-database-url](#), exists to manage this for us. dj-database-url is already installed since it is one of the helper packages added by environs[django].

All of which is to say, we can solve all these problems with a single line of code. Here is the brief update to make to django\_project/settings.py so that our project will try to access a DATABASE\_URL environment variable but, in its absence (aka when we are working locally), default to SQLite.

## Code

---

```
# django_project/settings.py
DATABASES = {
    "default": env.dj_db_url("DATABASE_URL", default="sqlite:///db.sqlite3"),
}
```

---

That's it! Isn't that elegant?

## Pyscopg

One more package related to databases needs to be installed now. It is [Pyscopg](#), a database adapter that lets Python apps like ours talk to PostgreSQL databases. On Windows, you can install it with pip, but if you are on macOS, it is necessary to install PostgreSQL first via [Homebrew](#).

## Shell

---

```
# Windows
(.venv) $ python -m pip install "psycopg[binary]==3.1.8"

# macOS
(.venv) $ brew install postgresql
(.venv) $ python3 -m pip install "psycopg[binary]==3.1.8"
```

---

We are using the [binary version](#) because it is the quickest way to start working with Psycopg.

## **CSRF\_TRUSTED\_ORIGINS**

Our message board app requires that we log into the admin on the production website to create, read, update, or delete posts. We must make sure that [CSRF\\_TRUSTED\\_ORIGINS](#) is correctly configured since it is a list of trusted origins for unsafe HTTP requests like POST. For now, we can set it to [https://\\*.fly.dev](https://*.fly.dev) since that will match our eventual production URL. This approach is also slightly insecure—it would be better to enter the exact production URL address—

but it is fine for now. Later on, in the book, we will lock down `CSRF_TRUSTED_ORIGINS` fully.

At the bottom of the `django_project/settings.py` file add a new line for `CSRF_TRUSTED_ORIGINS`.

#### Code

---

```
# django_project/settings.py
CSRF_TRUSTED_ORIGINS = ["https://*.fly.dev"] # new
```

---

## Updated Deployment Checklist

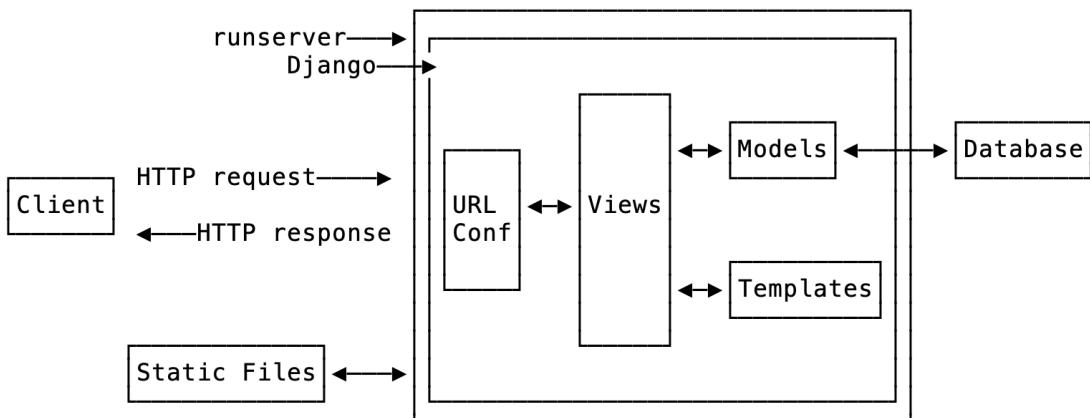
Let's pause for a moment to review what our updated deployment checklist looks like:

- install Gunicorn as a WSGI server
- install Psycopg to connect with a PostgreSQL database
- install environs for environment variables
- update DATABASES in `django_project/settings.py`
- generate a `requirements.txt` file
- update ALLOWED\_HOSTS and `CSRF_TRUSTED_ORIGINS` in `django_project/settings.py`
- add a `.dockerignore` file

Step by step, we are switching from local Django development to eventual deployment in production. We still have one more big step to undertake-configuring static files for production-but before that, we need to step back for a moment and review how web servers, WSGI, and ASGI work.

## Web Servers

The local server provided by Django and run via `runserver` is not intended for use in production. As a quick recap, here is our current visual diagram of Django:



**Django request/response cycle with static files**

A *web server* is the software that sits in front of our Django application to process HTTP requests and responses. It *also* manages static file requests. Before Platforms-as-a-Service became available, it was up to the web developer to install, configure, and maintain a dedicated web server like [Nginx](#) or [Apache](#): no small task and requiring a completely different skill set than web development. Fortunately, Fly.io knows we are deploying a website and automatically bundles in Nginx, so we don't have to install or manage it ourselves.

## WSGI

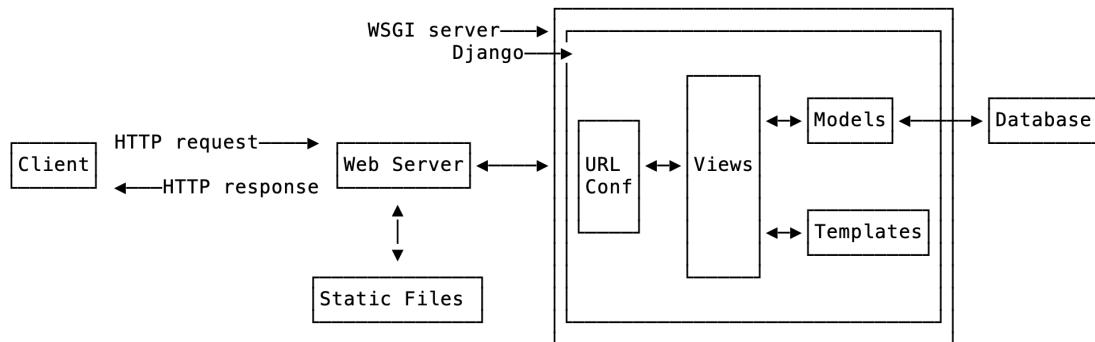
The other role that `runserver` has provided us is acting as an application server to help Django generate dynamic content. When a request comes in, it is `runserver` that powers that request through URLs, views, models, the database, templates and then generates an HTTP response. In other words, `runserver` really acts like two separate things: a web server *and* an application server.

In the early days of web development, web frameworks didn't always work easily with web servers. For Python web

frameworks, this led to the creation of the *Web Server Gateway Interface (WSGI)* in 2003. WSGI is not a server or framework but a set of rules that standardizes how web servers should connect to *any* Python web app. By abstracting away this headache, it opened the door to newer Python web frameworks-like Django, which was first released in 2005—that could work on any web server and did not have to worry about this step in the process.

Application servers are colloquially referred to as a “WSGI server” because they use WSGI to connect Python web apps to a server. We have already installed Gunicorn because it acts as our production WSGI server, replacing runserver.

Here is what the updated production diagram looks like adding a web server and replacing runserver with the more general WSGI server.



**Django request/response cycle with WSGI server**

## ASGI

Traditionally, Python was a *synchronous* programming language: code executed sequentially, meaning each piece of code had to complete before another piece of code could begin. As a result, complex tasks might take a while. Starting in 2012 with Python 3.3, *asynchronous*

programming was added to Python via the [asyncio](#) module. While synchronous processing is done sequentially in a specific order, asynchronous processing occurs in parallel. Tasks that are not dependent on others can be offloaded and executed at the same time as the main operation and then report back the result when complete.

Starting in 2019 with Django 3.0, [asynchronous support](#) has been gradually added to Django as well. One layer is the introduction of the *Asynchronous Server Gateway Interface (ASGI)*, which, as the name suggests, standardizes how servers should connect to Python web apps that support both synchronous and asynchronous communication. ASGI is intended as the eventual successor to WSGI.

ASGI and WSGI are both included in new Django projects now. When you run the `startproject` command, Django generates a `wsgi.py` file and an `asgi.py` file in the project-level directory, `django_project`.

As of Django 4.1, you can write `async` views, and full `async` support for the entire Django stack is still in the works. Given that this book is for beginners, it is important to recognize asynchronous developments rather than dwell on them. While they are exciting from a technical perspective, they are also challenging to reason about and are still in the early stages of development.

## WhiteNoise

Now that we better understand what a production Django project looks like, we can return to our deployment process and focus on static files.

For local development, the built-in Django web server performs many functions, including serving static files, but

this approach does not work in production. To view static files properly in production, we need to install the [WhiteNoise](#) package.

Static files will be stored on Fly servers for the projects in this book, an approach that works quite well for smaller projects and is easier to configure. But for truly large websites, static files are often stored on an external service like Amazon S3. If you are interested in configuring that yourself, look at the [django-storages](#) project.

Go ahead and install the latest version of WhiteNoise.

---

#### Shell

---

```
(.venv) $ python -m pip install whitenoise==6.4.0
```

---

Then in the `django_project/settings.py` file, make three changes:

- add `whitenoise` to `INSTALLED_APPS` above the built-in `staticfiles` app
- under `MIDDLEWARE` add `WhiteNoiseMiddleware` on the third line
- update the [`STORAGES`](#) setting, which is implicitly set by Django, to change the `staticfiles` alias to use `WhiteNoise`

---

#### Code

---

```
# django_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",
    "posts",
]

MIDDLEWARE = [
    "django.middleware.security.SecurityMiddleware",
    "django.contrib.sessions.middleware.SessionMiddleware",
    "whitenoise.middleware.WhiteNoiseMiddleware", # new
    "django.middleware.common.CommonMiddleware",
```

```
[ ...  
  STATIC_URL = "static/"  
  STATIC_ROOT = BASE_DIR / "staticfiles"  
  STORAGES = {  
    "default": {  
      "BACKEND": "django.core.files.storage.FileSystemStorage",  
    },  
    "staticfiles": {  
      "BACKEND": "whitenoise.storage.CompressedManifestStaticFilesStorage",  
    },  
  # new  
  },  
}
```

---

The ordering of WhiteNoiseMiddleware in the MIDDLEWARE is very important: it should be added just below SessionMiddleware, so it is third in the list.

## Middleware

Adding WhiteNoise is the first time we've updated the Django [middleware](#), a framework of hooks into Django's request/response processing. It is a way to add functionality such as authentication, security, sessions, and more. During the HTTP request phase, middleware is applied in the order it is defined in MIDDLEWARE\_CLASSES **top-down**. That means SecurityMiddleware comes first, then SessionMiddleware, and so on.

### Code

---

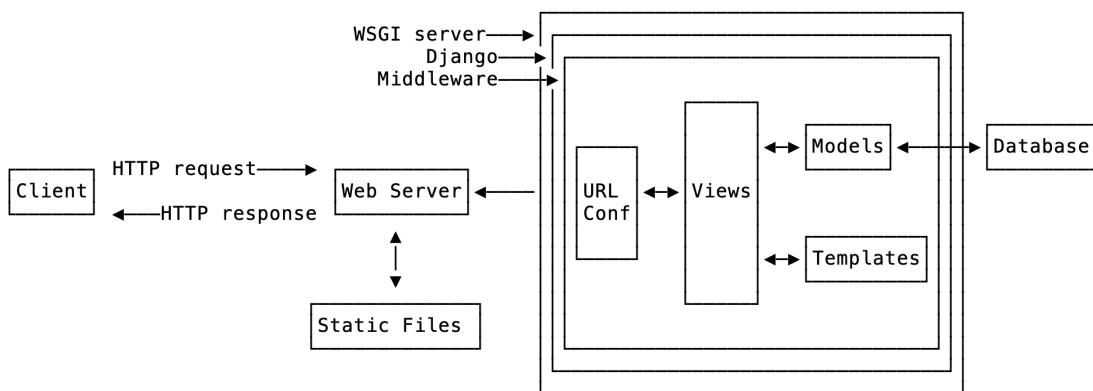
```
# django_project/settings.py  
MIDDLEWARE = [  
  "django.middleware.security.SecurityMiddleware",  
  "django.contrib.sessions.middleware.SessionMiddleware",  
  "whitenoise.middleware.WhiteNoiseMiddleware", # new  
  "django.middleware.common.CommonMiddleware",  
  "django.middleware.csrf.CsrfViewMiddleware",  
  "django.contrib.auth.middleware.AuthenticationMiddleware",  
  "django.contrib.messages.middleware.MessageMiddleware",  
  "django.middleware.clickjacking.XFrameOptionsMiddleware",  
]  
v
```

---

During the HTTP response phase, after the view is called, middleware are applied in reverse order, from the bottom

up, starting with `xFrameOptionsMiddleware`, then `MessageMiddleware`, and so on. The traditional way of describing middleware is like an onion, where each middleware class is a “layer” that wraps the view.

We can add middleware to our larger visual representation of Django as follows:



**Django request/response cycle with middleware**

Truly diving into middleware is an advanced topic beyond the scope of this book. It is important, however, to be conceptually aware of how all the pieces in the Django machine fit together.

Now that `WhiteNoise` is installed and configured, run the `collectstatic` command again to store our static files using `WhiteNoise` rather than the previous local Django defaults. Type `yes` to confirm you want to overwrite the existing files.

---

### Shell

```
(.venv) $ python manage.py collectstatic
You have requested to collect static files at the destination
location as specified in your settings:
```

```
/Users/wsv/Desktop/code/message-board/staticfiles
```

```
This will overwrite existing files!
Are you sure you want to do this?
```

```
Type 'yes' to continue, or 'no' to cancel: yes  
0 static files copied to '/Users/wsv/Desktop/code/message-board/staticfiles',  
130 unmodified, 384 post-processed.
```

---

And that's it! Our static files are compiled into a separate staticfiles directory that we set via the STATIC\_ROOT setting in the last chapter.

Code

---

```
# django_project/settings.py  
STATIC_URL = "static/"  
STATIC_ROOT = BASE_DIR / "staticfiles"
```

---

## requirements.txt

We are almost done at this point. The last step is to update the contents of the requirements.txt file since we have installed several additional packages.

Shell

---

```
(.venv) $ python -m pip freeze > requirements.txt
```

---

Your contents should look similar to this:

Code

---

```
# requirements.txt  
asgiref==3.6.0  
black==23.3.0  
click==8.1.3  
dj-database-url==1.3.0  
dj-email-url==1.0.6  
Django==4.2  
django-cache-url==3.4.4  
environs==9.5.0  
gunicorn==20.1.0  
marshmallow==3.19.0  
mypy-extensions==1.0.0  
packaging==23.1  
pathspec==0.11.1  
platformdirs==3.2.0  
psycopg==3.1.8  
psycopg-binary==3.1.8  
python-dotenv==1.0.0  
sqlparse==0.4.3
```

```
typing_extensions==4.5.0  
whitenoise==6.4.0
```

---

## Final Deployment Checklist

Here now is our final deployment checklist for the chapter.

- install Gunicorn as a WSGI server
- install Psycopg to connect with a PostgreSQL database
- install environs for environment variables
- update DATABASES in django\_project/settings.py
- install WhiteNoise for static files
- generate a requirements.txt file
- add a .dockerignore file
- update ALLOWED\_HOSTS and CSRF\_TRUSTED\_ORIGINS

Before turning our attention to Fly, please take a moment to add and commit our code changes, then push them to GitHub.

### Shell

---

```
(.venv) $ git status  
(.venv) $ git add -A  
(.venv) $ git commit -m "New updates for Fly deployment"  
(.venv) $ git push origin main
```

---

## Fly Deployment

You should already have a Fly account set up and installed from before. Make sure you are logged in: use the command `flyctl auth login` to confirm this. Then run the `fly launch` command to configure our website for deployment. Follow the wizard and make sure to say “Yes” to adding a Postgresql database:

- **Choose an app name:** this will be your dedicated `fly.dev` subdomain

- **Choose the region for deployment:** select the one closest to you or [another region](#) if you prefer
- **Decline overwriting our .dockerignore file:** our choices are already optimized for the project
- **Setup a Postgres database cluster:** the “Development” option is appropriate for this project. [Fly Postgres is a regular app deployed to Fly.io, not a managed database.](#)
- **Select “Yes” to scale a single node pg to zero after one hour:** this will save money for toy projects
- **Decline to setup a Redis database:** we don’t need one for this project

## Shell

---

```
(.venv) $ fly launch
Creating app in ~/desktop/code/message-board
Scanning source code
Detected a Django app
? Choose an app name (leave blank to generate one): dfb-ch5-mb
automatically selected personal organization: Will Vincent
? Choose a region for deployment: Boston, Massachusetts (US) (bos)
App will use 'bos' region as primary
Created app dfb-ch5-mb in organization personal
Admin URL: https://fly.io/apps/dfb-ch5-mb
Hostname: dfb-ch5-mb.fly.dev
? Overwrite "/Users/wsv/Desktop/code/message-board/.dockerignore"? No
Set secrets on dfb-ch5-mb: SECRET_KEY
? Would you like to set up a Postgresql database now? Yes
? Select configuration: Development - Single node, 1x shared CPU, 256MB RAM,
1GB
? Scale single node pg to zero after one hour? Yes
Creating postgres cluster in organization personal
Creating app...
...
? Would you like to set up an Upstash Redis database now? No
Wrote config file fly.toml
Your app is ready! Deploy with `flyctl deploy`
```

---

Notice now that we have added a Postgresql database, there is quite a bit more information provided to us, including a username, password, hostname, and so on, but this is all bundled into the `DATABASE_URL` environment variable and loaded correctly into our project thanks to the `dj-database-url` package.

The `fly launch` command has again created two new files for us: `fly.toml` and `Dockerfile`. Now we can run the command `flyctl deploy` to deploy our project on Fly servers

---

#### Shell

---

```
(.venv) $ flyctl deploy
  1 desired, 1 placed, 1 healthy, 0 unhealthy [health checks: 1 total, 1
passing]
--> v1 deployed successfully
```

---

And then `fly open` to open the hosted URL in your web browser.

---

#### Shell

---

```
(.venv) $ fly open
```

---

## Production Database

You will notice, however, that none of the message board posts you created are there. What happened?

Those posts were created locally in our file-based SQLite database, which we did not deploy. Instead, we are now using a brand-new PostgreSQL database. SSH into our deployed website to fix this and set up a superuser account. The `--pty` flag is necessary for prompts that ask for a password, like this one.

---

#### Shell

---

```
(.venv) $ fly ssh console --pty -C "python /code/manage.py createsuperuser"
```

---

Then visit the `/admin` page, which in my case, is located at `https://dfb-ch5-mb.fly.dev/admin/`. After logging in, create new posts, refresh the main production homepage, and the posts should appear.

If there are any issues with your deployment, you can use `fly logs` to see recent log file entries or go to `fly.io` to see

the full dashboard.

## Conclusion

We've now built, tested, and deployed our first database-driven app. While it's deliberately quite basic, we learned how to create a database model, update it with the admin panel, and then display the contents on a webpage. We also deepened our understanding of deployment by adding environment variables, connecting to a production PostgreSQL database, configuring WhiteNoise, and addressing security concerns around `ALLOWED_HOSTS` and `CSRF_TRUSTED_ORIGINS`.

In the next section, we will build a more complex *Blog* application that has user accounts for signup and login, allows users to create/edit/delete their posts, adds CSS for styling, and makes fuller use of environment variables to make our deployment even more secure.

# Chapter 6: Blog App

In this chapter, we will build a *Blog* application that allows users to create, edit, and delete posts. The homepage will list all blog posts, and each blog post will have a dedicated page. We'll also introduce CSS for styling and learn how Django works with static files.

## Initial Set Up

As covered in previous chapters, our steps for setting up a new Django project are as follows:

- make a new directory for our code called `blog`
- install Django in a new virtual environment called `.venv`
- create a new Django project called `django_project`
- create a new app `blog`
- perform a migration to set up the database
- update `django_project/settings.py`

Let's implement them now in a new command line terminal. Start with the new directory, a new virtual environment, and activate it.

### Shell

---

```
# Windows
$ cd onedrive\desktop\code
$ mkdir blog
$ cd blog
$ python -m venv .venv
$ .venv\Scripts\Activate.ps1
(.venv) $
```

```
# macOS
$ cd ~/desktop/code
$ mkdir blog
$ cd blog
$ python3 -m venv .venv
$ source .venv/bin/activate
(.venv) $
```

---

Then install Django and Black, create a new project called django\_project, a new app called blog, and migrate the initial database.

---

#### Shell

---

```
(.venv) $ python -m pip install django~=4.2.0
(.venv) $ python -m pip install black
(.venv) $ django-admin startproject django_project .
(.venv) $ python manage.py startapp blog
(.venv) $ python manage.py migrate
```

---

To ensure Django knows about our new app, open your text editor and add the new app to INSTALLED\_APPS in the django\_project/settings.py file:

---

#### Code

---

```
# django_project/settings.py
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "blog", # new
]
```

---

Spin up the local server using the runserver command.

---

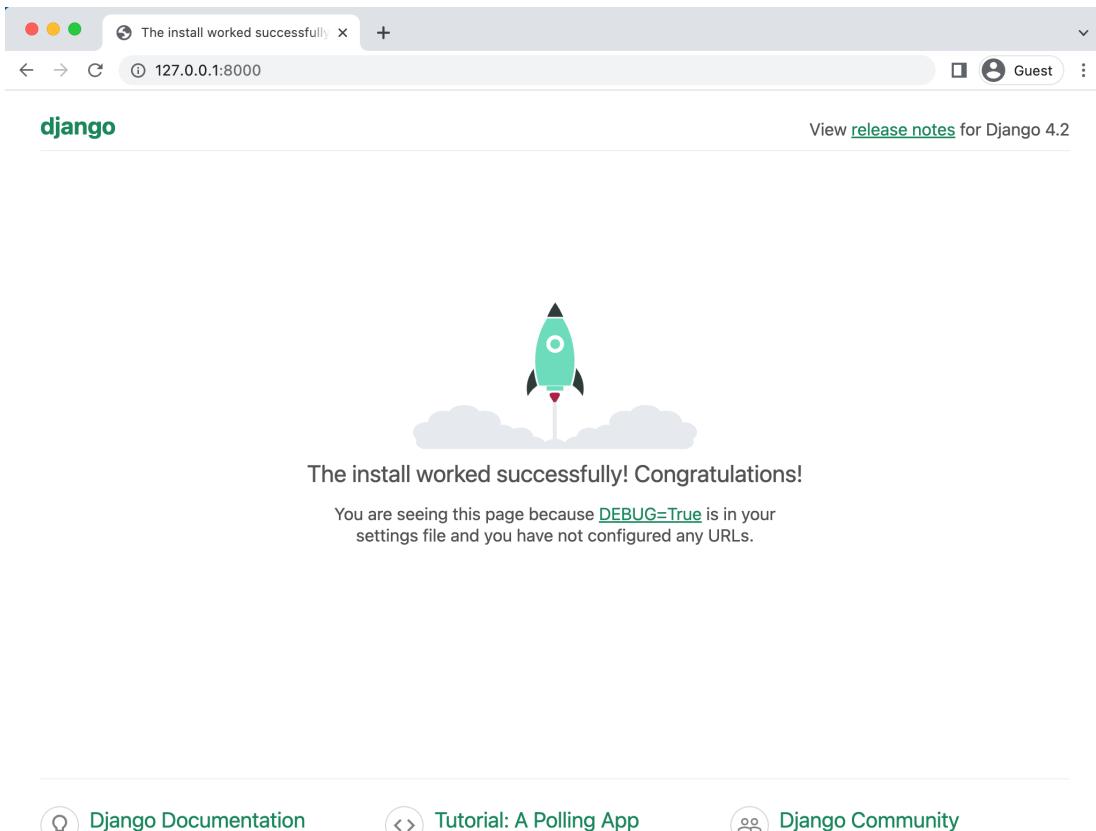
#### Shell

---

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

---

If you navigate to <http://127.0.0.1:8000/> in your web browser, you should see the friendly Django welcome page.



## Django welcome page

Ok, the initial installation is complete! Next, we'll learn more about databases and Django's ORM, then create our database models for a blog application.

## Databases

Before we dive into implementing our Blog post models, it is worth taking a step back to review how databases, ORMs, and Django work together. A database is a place to store and access different types of data, and there are two main types of databases: relational and non-relational.

A relational database stores information in tables containing columns and rows, roughly analogous to an Excel

spreadsheet. The columns define what information can be stored; the rows contain the actual data. Frequently, data in separate tables have some relation to each other, hence the term “relational database” to describe databases with this structure of tables, columns, and rows.

SQL (Structured Query Language) is typically used to interact with relational databases to perform basic CRUD (Create, Read, Update, Delete) operations and define the type of relationship (like a *many-to-one* relationship, for example. We’ll learn more about these shortly.).

A non-relational database is any database that doesn’t use the tables, fields, rows, and columns inherent in relational databases to structure its data: examples include document-oriented, key-value, graph, and wide-column.

Relational databases are best when data is consistent, structured, and relationships between entities are essential. Non-relational databases have advantages when data is not structured, needs to be flexible in size or shape, and must be open to change in the future. Relational databases have been around far longer and are more widely used, while many non-relational databases were designed recently for specific use in the cloud.

Database design and implementation is an entire field of computer engineering that is very deep and quite interesting but far beyond the scope of this book. The important takeaway for our purposes is that these two types of databases exist. Still, Django only has built-in support for relational databases, so that is what we will focus on going forward.

## Django’s ORM

An ORM (Object-Relational Mapper) is a powerful programming technique that makes working with data and relational databases much easier. In the case of Django, its ORM means we can write Python code to define database models; we don't have to write raw SQL ourselves. And we don't have to worry about subtle differences in how each database interprets SQL. Instead, the Django ORM supports five relational databases: SQLite, PostgreSQL, MySQL, MariaDB, and Oracle. It also comes with support for migrations which provides a way to track and sync database changes over time. In sum, the Django ORM saves developers a tremendous amount of time and is one of the major reasons why Django is so efficient.

While the ORM abstracts much of the work, we still need a basic understanding of relational databases if we want to implement them correctly. For example, before writing any actual code, let's look at structuring the data in our Blog database.

Recall that we create a table by adding columns to define different “fields” of data. So, for example, we could start with a table called “Post” with columns for the title, author, and body text. If we drew this out as a simple schema, it would look something like this:

Post Schema		
<hr/>		
TITLE	AUTHOR	BODY

And the actual database table with columns and rows would look like this:

Post Database Table		
<hr/>		
TITLE	AUTHOR	BODY

Hello, World!	wsv	My first blog post. Woohoo!
Goals Today	wsv	Learn Django and build a blog application.
3rd Post	wsv	This is my 3rd entry.

---

## Blog Post Models

At the beginning of this chapter, we used the command `python manage.py startapp blog` to create a new `blog` app within our project, resulting in a `blog` directory containing several additional files, including `blog/models.py`. In Django, a `models.py` file is the single, definitive source of information about your data, and it contains the necessary fields and behaviors of the data being stored. We can write Python in a `models.py` file, and the Django ORM will translate it into SQL for us.

To mimic the previous `Post` table using the Django ORM, add the following code to the `blog/models.py` file.

Code

---

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

class Post(models.Model):
    title = models.CharField(max_length=200)
    author = models.CharField(max_length=200)
    body = models.TextField()

    def __str__(self):
        return self.title
```

---

At the top of the file, we import `models`. In Django, a [model](#) is the definitive source of information about your data. Next, we create a subclass of `models.Model` called `Post`, which provides access to everything within [`django.db.models.Model`](#). And from there, we have added three additional fields (think of them as columns) for `title`, `author`, and `body`. Each field must have an appropriate [field type](#). The first two use `CharField` with a maximum character

length of 200, while the third uses `TextField`, which is intended for a large amount of text.

Adding the `__str__` method is technically optional, but as we saw in the last chapter, it is a best practice to ensure a human-readable version of our model object in the Django admin. In this case, it will display the title field of any blog post.

Now that our new database model exists, we need to create a new migration record for it and migrate the change so it is applied to our database. Stop the server with `Control+c`. You can complete this two-step process with the commands below:

---

#### Shell

---

```
(.venv) $ python manage.py makemigrations blog  
(.venv) $ python manage.py migrate
```

---

The database is now configured, and there is a new `migrations` directory within the `blog` app directory containing our changes.

## Primary Keys and Foreign Keys

We could now hop over to the Django admin and add data to our blog post model. But there are two more important concepts to cover before we build out the rest of the Blog app: primary and foreign keys.

Because relational databases have relationships between tables, there needs to be an easy way for them to communicate. The solution is adding a column-known as a *primary key* that contains unique values. When there is a relationship between two tables, the primary key is the link maintaining a consistent relationship. If we look back at our

simple Post schema, it should therefore include another field for “Primary Key”:

---

#### Post Schema

---

```
Post
-----
primary_key
title
author
body
```

---

Primary keys are a standard part of relational database design. As a result, Django automatically adds an [auto-incrementing primary key](#) to our database models. Its value starts at 1 and increases sequentially to 2, 3, and so on. The naming convention is <table\_id>, meaning that for a model called Post the primary key column is named post\_id.

---

#### Post Schema

---

```
Post
-----
post_id
title
author
body
```

---

As a result, under the hood, our existing Post database table has four columns/fields.

---

#### Post Database Table

---

```
Post
-----
POST_ID  TITLE        AUTHOR      BODY
1        Hello, World!  wsv        My first blog post. Woohoo!
2        Goals Today   wsv        Learn Django and build a blog application.
3        3rd Post       wsv        This is my 3rd entry.
```

---

Now that we know about primary keys, it’s time to see how they are used to link tables. When you have more than one table, each will contain a column of primary keys starting with 1 and increasing sequentially, just like in our Post model example. In our blog model, consider that we have a field

for author, but in the actual Blog app, we want users to be able to log in and create blog posts. That means we'll need a second table for users to link to our existing table for blog posts. Fortunately, authentication is such a common-and challenging to implement well-feature on websites that Django has an entire built-in [authentication system](#) that we can use. In a later chapter, we will use it to add signup, login, logout, password reset, and other functionality. But for now, we can use the Django [auth user model](#), which comes with various fields. If we visualized the schema of our Post and User models now, it would look like this:

Post and User Schema	
Post	User
-----	-----
post_id	user_id
title	username
author	first_name
body	last_name
	email
	password
	groups
	user_permissions
	is_staff
	is_active
	is_superuser
	last_login
	date_joined

How do we link these two tables together so they have a relationship? We want the author field in Post to link to the User model so that each post has an author that corresponds to a user. And we can do this by linking the User model primary key, user\_id, to the Post.author field. A link like this is known as a *foreign key* relationship. Foreign keys in one table always correspond to the primary keys of a different table. So establishing a foreign key relationship for authors and users in our blog app means that the author fields of the Post model will have the primary key of the corresponding user in the User model who authored that specific post. In our example, wsv, whose primary key in the

User model is 1, authored all three of the posts, so that same primary key, 1, is listed as the foreign key in the Author column for each of the three posts in our Post model.

Here is how it looks in the code. We only need to change the author field in our Post model.

---

Code

---

```
# blog/models.py
from django.db import models
from django.urls import reverse

class Post(models.Model):
    title = models.CharField(max_length=200)
    author = models.ForeignKey(
        "auth.User",
        on_delete=models.CASCADE,
    ) # new
    body = models.TextField()

    def __str__(self):
        return self.title

    def get_absolute_url(self): # new
        return reverse("post_detail", kwargs={"pk": self.pk})
```

---

The [ForeignKey](#) field defaults to a *many-to-one* relationship, meaning one user can be the author of many different blog posts but not the other way around.

It is worth mentioning that there are three types of foreign relationships: *many-to-one*, *many-to-many*, and *\_one-to-one*. A many-to-one relationship, as we have in our Post model, is the most common occurrence. A many-to-many relationship would exist if there were a database tracking authors and books: each author can write multiple books, and each book can have multiple authors. A one-to-one relationship would exist in a database tracking people and passports: only one person can have one passport.

Note that when an object referenced by a ForeignKey is deleted, an additional [on\\_delete](#) argument must be set.

Understanding `on_delete` fully is an advanced topic but typically, choosing CASCADE is a safe choice, as we do here.

We also add a `get_absolute_url()` method, which we haven't seen before, that tells Django how to calculate the canonical URL for our model object. It says to use the URL named `post_detail` and pass in the `pk`. More on this later as we build out our blog app.

Since we have updated our database models again, we should create a new migrations file and then migrate the database to apply it.

---

#### Shell

---

```
(.venv) $ python manage.py makemigrations blog  
(.venv) $ python manage.py migrate
```

---

A second migrations file will now appear in the `blog/migrations` directory that documents this change.

## Admin

We need a way to access our data. Enter the Django admin! First, create a superuser account by typing the command below and following the prompts to set up an email and password. Note that typing your password it will not appear on the screen for security reasons.

---

#### Shell

---

```
(.venv) $ python manage.py createsuperuser  
Username (leave blank to use 'wsv'): wsv  
Email:  
Password:  
Password (again):  
Superuser created successfully.
```

---

Now rerun the Django server with the command `python manage.py runserver` and navigate to the admin at

127.0.0.1:8000/admin/. Log in with your new superuser account.

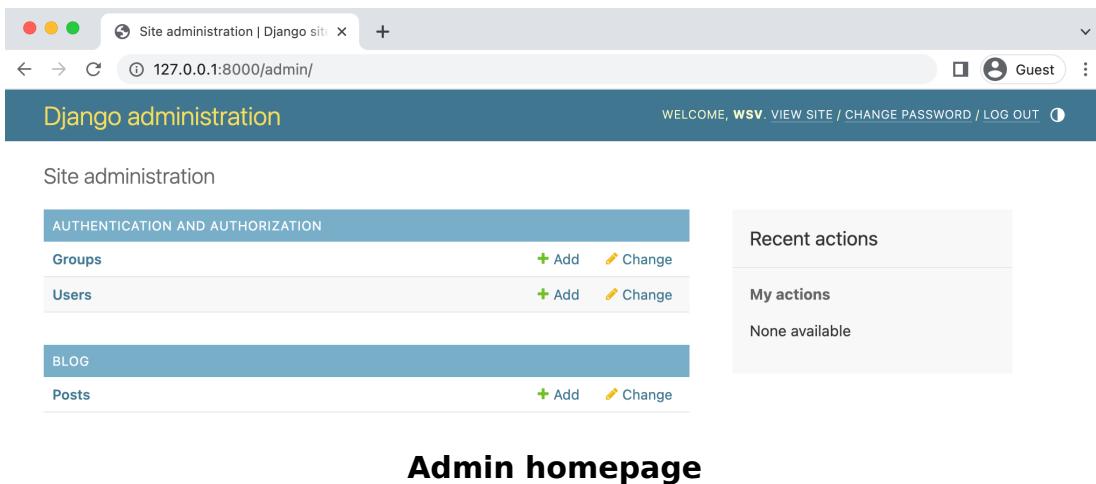
Oops! Where's our new Post model? We forgot to update blog/admin.py so let's do that now.

#### Code

```
# blog/admin.py
from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

If you refresh the page, you'll see the update.



The screenshot shows the Django Admin homepage. At the top, there's a header bar with the title "Django administration". Below it, the main content area has a sidebar on the left labeled "Site administration". Under "AUTHENTICATION AND AUTHORIZATION", there are links for "Groups" and "Users", each with "+ Add" and "Change" buttons. Under the "BLOG" heading, there is a link for "Posts", also with "+ Add" and "Change" buttons. To the right of the sidebar, there are two boxes: "Recent actions" (empty) and "My actions" (also empty). At the bottom center, the text "Admin homepage" is displayed.

Let's add two blog posts, so we have some sample data. Click on the + Add button next to Posts to create a new entry. Make sure to add an "author" to each post, too, since all model fields are required by default.

You will see an error if you try to enter a post without an author. If we wanted to change this, we could add [field options](#) to our model to make a given field optional or default to a specified value.

The screenshot shows the Django admin interface for the 'Posts' model. On the left, there's a sidebar with 'Groups' and 'Users' under 'AUTHENTICATION AND AUTHORIZATION', and 'Posts' under 'BLOG'. The main area shows a list of posts with checkboxes for selecting them. A success message at the top right says 'The post "Goals today" was added successfully.' Below it, the list includes 'POST' and 'Goals today'. An 'ADD POST' button is at the top right of the list area.

### Admin homepage with two posts

Now that our database model is complete, we must create the necessary views, URLs, and templates to display the information on our web application.

## URLs

We want to display our blog posts on the homepage, so we'll first configure our app-level `blog/urls.py` file and then our project-level `django_project/urls.py` file to achieve this.

In your text editor, create a new file called `urls.py` within the `blog` app and update it with the code below.

### Code

```
# blog/urls.py
from django.urls import path
from .views import BlogListView

urlpatterns = [
    path("", BlogListView.as_view(), name="home"),
]
```

We're importing our soon-to-be-created views at the top. The empty string, "", tells Python to match all values, and

we make it a named URL, `home`, which we can refer to in our views later on. While it's optional to add a [named URL](#), it's a best practice you should adopt as it helps keep things organized as your number of URLs grows.

We also should update our `django_project/urls.py` file so that it knows to forward all requests directly to the `blog` app.

Code

---

```
# django_project/urls.py
from django.contrib import admin
from django.urls import path, include # new

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

---

We've added `include` on the second line and a URL pattern using an empty string regular expression, `" "`, indicating that URL requests should be redirected as is to `blog`'s URLs for further instructions.

## Views

We will use class-based views, but if you want to see a function-based way to build a blog application, I highly recommend the [Django Girls Tutorial](#). It is excellent.

In our `views` file, add the code below to display the contents of our `Post` model using `ListView`. It is quite rare that we use the default `views.py` code of `from django.shortcuts import render` code that Django provides.

Code

---

```
# blog/views.py
from django.views.generic import ListView
from .models import Post

class BlogListView(ListView):
```

```
model = Post
template_name = "home.html"
```

---

On the top two lines, we import [ListView](#) and our database model Post. Then we subclass ListView and add links to our model and template, saving us a lot of code versus implementing it all from scratch.

## Templates

With our URLs and views now complete, we're only missing the third piece of the puzzle: templates. As we already saw in **Chapter 4**, we can inherit from other templates to keep our code clean. Thus we'll start with a `base.html` file and a `home.html` file that inherits from it. Then later, when we add templates for creating and editing blog posts, they too can inherit from `base.html`.

Start by creating our new templates directory.

Shell

```
(.venv) $ mkdir templates
```

---

Create two new templates in your text editor: `templates/base.html` and `templates/home.html`. Then update `django_project/settings.py` so Django knows to look there for our templates.

Code

```
# django_project/settings.py
TEMPLATES = [
    {
        ...
        "DIRS": [BASE_DIR / "templates"], # new
        ...
    },
]
```

---

And update the `base.html` template as follows.

## Code

---

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

<head>
    <title>Django blog</title>
</head>

<body>
    <header>
        <h1><a href="{% url 'home' %}">Django blog</a></h1>
    </header>
    <div>
        {% block content %}
        {% endblock content %}
    </div>
</body>

</html>
```

---

Note that code between `{% block content %}` and `{% endblock content %}` can be filled by other templates. Speaking of which, here is the code for `home.html`.

## Code

---

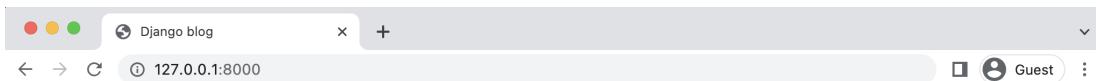
```
<!-- templates/home.html -->
{% extends "base.html" %}

{% block content %}
{% for post in post_list %}
<div class="post-entry">
    <h2><a href="">{{ post.title }}</a></h2>
    <p>{{ post.body }}</p>
</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 blog post. Note that `post_list` comes from `ListView` and contains all the objects in our view of the model `post`.

If you start the Django server again with `python manage.py runserver` and refresh the homepage, we can see it is working.



## Blog homepage with two posts

But it looks terrible. Let's fix that!

## Static Files

We need to add some CSS to our project to improve the styling. CSS, JavaScript, and images are a core piece of any modern web application and within the Django world, are referred to as “static files.” Django provides tremendous flexibility around *how* these files are used, but this can lead to quite a lot of confusion for newcomers.

By default, Django will look within each app for a folder called “static”; in other words, a folder called `blog/static/`. If you recall, this is similar to how templates are treated.

As Django projects grow in complexity over time and have multiple apps, it is often simpler to reason about static files if they are stored in a single, project-level directory instead. That is the approach we will take here.

Quit the local server with `Control+c` and create a new static directory in the same folder as the `manage.py` file.

Shell

---

```
(.venv) $ mkdir static
```

---

Then we must tell Django to look for this new folder when loading static files. If you look at the bottom of the `django_project/settings.py` file, there is already a single line of configuration:

Code

---

```
# django_project/settings.py
STATIC_URL = "static/"
```

---

STATIC\_URL is the *URL location* of static files in our project, aka at `static/`.

STATICFILES\_DIRS defines *additional locations* the built-in `staticfiles` app will traverse looking for static files beyond an `app/static` folder. We need to set it to have a project-level static folder instead, and it is also necessary for local static file viewing. Later in the book, we will add settings to configure our static files for production, but this is enough for local development purposes.

Code

---

```
# django_project/settings.py
STATIC_URL = "static/"
STATICFILES_DIRS = [BASE_DIR / "static"] # new
```

---

Next, create a `css` directory within `static`.

Shell

---

```
(.venv) $ mkdir static/css
```

---

In your text editor, create a new file within this directory called `static/css/base.css`.

What should we put in our file? How about changing the title to red?

Code

---

```
/* static/css/base.css */
header h1 a {
```

```
    color: red;  
}
```

---

Last step now. We need to add the static files to our templates by adding `{% load static %}` to the top of `base.html`. Because our other templates inherit from `base.html`, we only have to add this once. Include a new line at the bottom of the `<head></head>` code that explicitly references our new `base.css` file.

#### Code

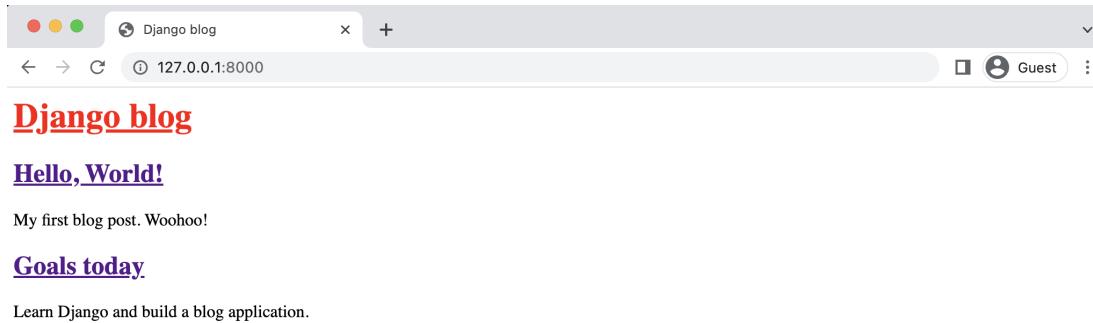
---

```
<!-- templates/base.html -->  
<html>  
  
<head>  
  <title>Django blog</title>  
  <link rel="stylesheet" href="{% static 'css/base.css' %}">  
</head>  
...
```

---

Phew! That was a pain but it's a one-time hassle. Now we can add static files to our `static` directory, which will automatically appear in all our templates.

Start the server again with `python manage.py runserver` and look at our updated homepage at `http://127.0.0.1:8000/`.



#### Blog homepage with red title

If you see an error, `TemplateSyntaxError` at `/`, you forgot to add the `{% load static %}` line at the top. Even after all my

years of using Django, I still make this mistake all the time! Fortunately, Django's error message says, "Invalid block tag on line 4: 'static'. Did you forget to register or load this tag?". A pretty accurate description of what happened, no?

Even with this new styling, we can still do a little better. Let's add a custom font and some more CSS. Since this book is not on CSS, we can simply insert the following between `<head></head>` tags to add [Source Sans Pro](#), a free font from Google.

Code

---

```
<!-- templates/base.html -->
{% load static %}
<html>

<head>
    <title>Django blog</title>
    <link href="https://fonts.googleapis.com/css?family=Source+Sans+Pro:400"
        rel="stylesheet">
    <link href="{% static 'css/base.css' %}" rel="stylesheet">
</head>
...
```

---

Then update our CSS file by copying and pasting the following code:

Code

---

```
/* static/css/base.css */
body {
    font-family: 'Source Sans Pro', sans-serif;
    font-size: 18px;
}

header {
    border-bottom: 1px solid #999;
    margin-bottom: 2rem;
    display: flex;
}

header h1 a {
    color: red;
    text-decoration: none;
}

.nav-left {
    margin-right: auto;
```

```
}

.nav-right {
    display: flex;
    padding-top: 2rem;
}

.post-entry {
    margin-bottom: 2rem;
}

.post-entry h2 {
    margin: 0.5rem 0;
}

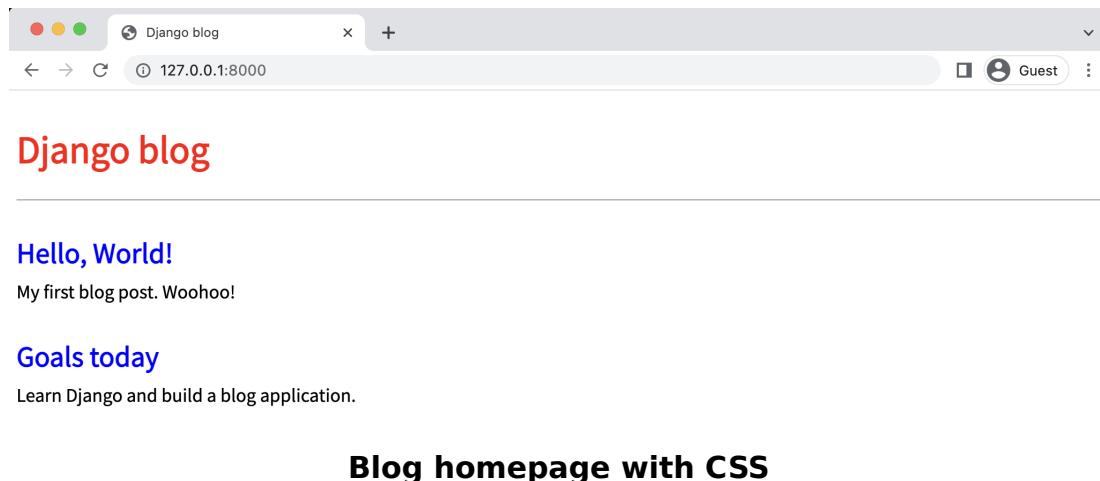
.post-entry h2 a,
.post-entry h2 a:visited {
    color: blue;
    text-decoration: none;
}

.post-entry p {
    margin: 0;
    font-weight: 400;
}

.post-entry h2 a:hover {
    color: red;
}
```

---

Refresh the homepage at `http://127.0.0.1:8000/`; you should see the following.



## Individual Blog Pages

Now we can add the functionality for individual blog pages. How do we do that? We need to create a new view, URL, and template. I hope you're noticing a pattern in development with Django now!

Start with the view. We can use the generic class-based [DetailView](#) to simplify things. At the top of the file, add `DetailView` to the list of imports and then create a new view called `BlogDetailView`.

Code

---

```
# blog/views.py
from django.views.generic import ListView, DetailView # new
from .models import Post

class BlogListView(ListView):
    model = Post
    template_name = "home.html"

class BlogDetailView(DetailView): # new
    model = Post
    template_name = "post_detail.html"
```

---

In this new view, we define the model we're using, `Post`, and the template we want it associated with, `post_detail.html`. By default, `DetailView` will provide a context object we can use in our template called either `object` or the lowercase name of our model, which would be `post`. Also, `DetailView` expects either a primary key or a slug passed to it as the identifier. More on this shortly.

Create a new template file for a post detail called `templates/post_detail.html` in your text editor. Then type in the following code:

Code

---

```
<!-- templates/post_detail.html -->
{% extends "base.html" %}

{% block content %}
<div class="post-entry">
```

```
<h2>{{ post.title }}</h2>
<p>{{ post.body }}</p>
</div>
{% endblock content %}
```

---

At the top, we specify that this template inherits from `base.html`. Then display the `title` and `body` from our context object, which `DetailView` makes accessible as `post`.

I found naming context objects in generic views extremely confusing when first learning Django. Because our context object from `DetailView` is either our model name `post` or `object`, we *could* also update our template as follows, and it would work exactly the same.

Code

---

```
<!-- templates/post_detail.html -->
{% extends "base.html" %}

{% block content %}
<div class="post-entry">
  <h2>{{ object.title }}</h2>
  <p>{{ object.body }}</p>
</div>
{% endblock content %}
```

---

If you find using `post` or `object` confusing, it is possible to explicitly name the context object in our view using [context object name](#).

The “magic” naming of the context object is a price you pay for the ease and simplicity of using generic views. It is great if you know what they’re doing, but take a little research in the official documentation to customize.

Ok, what’s next? How about adding a new URL path for our view, which we can do as follows.

Code

---

```
# blog/urls.py
from django.urls import path
from .views import BlogListView, BlogDetailView # new
```

```
urlpatterns = [
    path("post/<int:pk>/", BlogDetailView.as_view(), name="post_detail"), #
    new
    path("", BlogListView.as_view(), name="home"),
]
```

---

All blog post entries will start with post/. To represent each post entry, we can use the auto-incrementing primary key, which is represented as an integer, <int:pk>. The pk for our first “Hello, World” post is 1; for the second post, it is 2; and so on. Therefore, when we go to the individual entry page for our first post, we can expect that its URL pattern will be post/1/.

This works because we set the `get_absolute_url` method on our Post model earlier. It returns the detail view with the URL name `post_detail` and passes in the keyword argument, `kwargs, for pk`. Detail views are commonplace in Django web applications, and so is using this pattern of `DetailView` and `get_absolute_url`. Here is that code again for reference:

Code

---

```
# blog/models.py
from django.db import models
from django.urls import reverse

class Post(models.Model):
    title = models.CharField(max_length=200)
    author = models.ForeignKey(
        "auth.User",
        on_delete=models.CASCADE,
    )
    body = models.TextField()

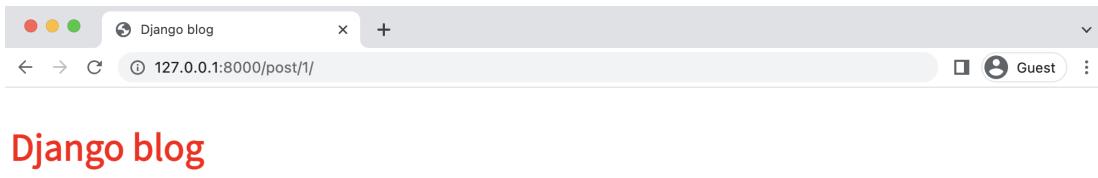
    def __str__(self):
        return self.title

    def get_absolute_url(self):
        return reverse("post_detail", kwargs={"pk": self.pk})
```

---

If you start up the server with `python manage.py runserver`, you’ll see a dedicated page for our first blog post at

<http://127.0.0.1:8000/post/1/>.



The screenshot shows a web browser window titled "Django blog". The address bar indicates the URL is "127.0.0.1:8000/post/1/". The page content includes a header "Django blog" in red, a title "Hello, World!", and a subtitle "My first blog post. Woohoo!". Below this, there is a section titled "Blog post one detail".

Woohoo! You can also go to <http://127.0.0.1:8000/post/2/> to see the second entry.

To make our life easier, we should update the link on the homepage so we can directly access individual blog posts from there. Swap out the current empty link, `<a href="">`, for `<a href="{% url 'post_detail' post.pk %}">`.

#### Code

```
<!-- templates/home.html -->
{% extends "base.html" %}

{% block content %}
{% for post in post_list %}
<div class="post-entry">
  <h2><a href="{% url 'post_detail' post.pk %}">{{ post.title }}</a></h2>
  <p>{{ post.body }}</p>
</div>
{% endfor %}
{% endblock content %}
```

We start by using Django's `url` template tag and specifying the URL pattern name of `post_detail`. If you look at `post_detail` in our URLs file, it expects to be passed an argument `pk` representing the primary key for the blog post. Fortunately, Django has already created and included this `pk` field on our `post` object, but we must pass it into the URL by adding it to the template as `post.pk`.

To confirm everything works, refresh the main page at `http://127.0.0.1:8000/` and click on the title of each blog post to confirm the new links work.

## get\_absolute\_url()

Although this approach works well, there is actually a better option available. Earlier in the chapter, we defined a `get_absolute_url()` method in our model that defined a canonical (meaning official) URL for the model. So we can take the simpler step of using `<a href="{{ post.get_absolute_url }}>{{ post.title }}</a></h2>` in the template instead.

Code

---

```
<!-- templates/home.html -->
{% extends "base.html" %}

{% block content %}
{% for post in post_list %}
<div class="post-entry">
    <h2><a href="{{ post.get_absolute_url }}>{{ post.title }}</a></h2> <!-- new
-->
    <p>{{ post.body }}</p>
</div>
{% endfor %}
{% endblock content %}
```

---

URL paths can and do change over the lifetime of a project. With the previous method, if we changed the post detail view and URL path, we'd have to go through all our HTML and templates to update the code, a very error-prone and hard-to-maintain process. By using `get_absolute_url()` instead, we have one single place, the `models.py` file, where the canonical URL is set, so our templates don't have to change.

Refresh the main page at `http://127.0.0.1:8000/` and click on the title of each blog post to confirm the links still work as expected.

## Tests

Our *Blog* project has added new functionality we have not seen or tested before this chapter. The Post model has multiple fields, we have a user for the first time, and there is a list view of all blog posts and a detailed view for each blog post. Quite a lot to test!

To begin, we can set up our test data and check the Post model's content. Here's how that might look.

Code

---

```
# blog/tests.py
from django.contrib.auth import get_user_model
from django.test import TestCase

from .models import Post

class BlogTests(TestCase):
    @classmethod
    def setUpTestData(cls):
        cls.user = get_user_model().objects.create_user(
            username="testuser", email="test@email.com", password="secret"
        )

        cls.post = Post.objects.create(
            title="A good title",
            body="Nice body content",
            author=cls.user,
        )

    def test_post_model(self):
        self.assertEqual(self.post.title, "A good title")
        self.assertEqual(self.post.body, "Nice body content")
        self.assertEqual(self.post.author.username, "testuser")
        self.assertEqual(str(self.post), "A good title")
        self.assertEqual(self.post.get_absolute_url(), "/post/1/")
```

---

At the top, we import `get_user_model()` to refer to our User and then added `TestCase` and the `Post` model. Our class `BlogTests` contains set-up data for both a test user and a test post. Currently, all the tests are focused on the `Post` model, so we name our test `test_post_model`. It checks that all three

model fields return the expected values. Our model also has new tests for the `_str_` and `get_absolute_url` methods.

Go ahead and run the tests.

---

#### Shell

---

```
(.venv) $ python manage.py test
Found 1 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

.
-----
Ran 1 test in 0.088s

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

---

What else to add? We now have two types of pages: a homepage that lists all blog posts and a detail page for each blog post containing its primary key in the URL. In the previous two chapters, we implemented tests to check that:

- expected URLs exist and return a 200 status code
- URL names work and return a 200 status code
- the correct template name is used
- the correct template content is outputted

All four tests need to be included. We *could* have eight new unit tests: four for our two pages. Or we could combine them a bit. There isn't a right or wrong answer here so long as tests are implemented to test functionality, and it is clear from their names what went wrong if an error arises.

Here is one way to add these checks to our code:

---

#### Code

---

```
# blog/tests.py
from django.contrib.auth import get_user_model
from django.test import TestCase
from django.urls import reverse # new

from .models import Post
```

```

class BlogTests(TestCase):
    @classmethod
    def setUpTestData(cls):
        cls.user = get_user_model().objects.create_user(
            username="testuser", email="test@email.com", password="secret"
        )

        cls.post = Post.objects.create(
            title="A good title",
            body="Nice body content",
            author=cls.user,
        )

    def test_post_model(self):
        self.assertEqual(self.post.title, "A good title")
        self.assertEqual(self.post.body, "Nice body content")
        self.assertEqual(self.post.author.username, "testuser")
        self.assertEqual(str(self.post), "A good title")
        self.assertEqual(self.post.get_absolute_url(), "/post/1/")

    def test_url_exists_at_correct_location_listview(self): # new
        response = self.client.get("/")
        self.assertEqual(response.status_code, 200)

    def test_url_exists_at_correct_location_detailview(self): # new
        response = self.client.get("/post/1/")
        self.assertEqual(response.status_code, 200)

    def test_post_listview(self): # new
        response = self.client.get(reverse("home"))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "Nice body content")
        self.assertTemplateUsed(response, "home.html")

    def test_post_detailview(self): # new
        response = self.client.get(reverse("post_detail",
            kwargs={"pk": self.post.pk}))
        no_response = self.client.get("/post/100000/")
        self.assertEqual(response.status_code, 200)
        self.assertEqual(no_response.status_code, 404)
        self.assertContains(response, "A good title")
        self.assertTemplateUsed(response, "post_detail.html")

```

---

First, we check that URL exists at the proper location for both views. Then we import [reverse](#) at the top and create `test_post_listview` to confirm that the named URL is used, returns a 200 status code, contains the expected content, and uses the `home.html` template. For `test_post_detailview`, we have to pass in the `pk` of our test post to the response. The same template is used, and we add new tests for what we

*don't want* to see. For example, we don't want a response at the URL `/post/100000/` because we have not created that many posts yet! And we don't want a 404 HTTP status response either. It is always good to sprinkle in examples of incorrect tests that *should* pass through failure using the `no_response` method to ensure your tests aren't all blindly passing for some reason.

Run the new tests to confirm everything is working.

---

#### Shell

---

```
(.venv) $ python manage.py test
Found 5 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

.....
-----
Ran 5 tests in 0.095s

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

---

A common gotcha when testing URLs is failing to include the preceding slash `/`. For example, if `test_url_exists_at_correct_location_detailview` is checked in the response for `"post/1/"` that would throw a 404 error. However, if you check `"/post/1/"` it will be a 200 status response.

## Git

Now is also a good time for our first Git commit. First, initialize our directory, then review all the added content by checking the status.

---

#### Shell

---

```
(.venv) $ git init
(.venv) $ git status
```

---

Oops, there is the `.venv` directory we do not want to include, and the SQLite database! There might also be a `__pycache__` directory. To remove all three, in your text editor, create a project-level `.gitignore` file-in the same top directory as `manage.py`-and add these three lines.

---

```
.gitignore
.venv/
__pycache__/
*.sqlite3
```

---

Run `git status` again to confirm the `.venv` directory is no longer included. Then add the rest of our work along with a commit message.

---

#### Shell

---

```
(.venv) $ git status
(.venv) $ git add -A
(.venv) $ git commit -m "initial commit"
```

---

## Conclusion

We've now built a basic blog application from scratch! We can create, edit, or delete the content using the Django admin. And we used `DetailView` for the first time to make a detailed individual view of each blog post entry.

In the next section, we'll add forms so we don't have to use the Django admin for these changes.

# Chapter 7: Forms

In this chapter, we'll continue working on our *Blog* application by adding forms so a user can create, edit, or delete any of their blog entries. HTML forms are one of the more complicated and error-prone aspects of web development because any time you accept user input, there are security concerns. All forms must be properly rendered, validated, and saved to the database.

Writing this code by hand would be time-consuming and difficult, so Django comes with powerful [built-in Forms](#) that abstract away much of the difficulty for us. Django also comes with [generic editing views](#) for common tasks like displaying, creating, updating, or deleting a form.

## CreateView

To start, update our base template to display a link to a page for entering new blog posts. It will take the form `<a href="{% url 'post_new' %}"></a>` where `post_new` is the name for our URL.

Your updated file should look as follows:

Code

---

```
<!-- templates/base.html -->
{% load static %}
<html>

<head>
    <title>Django blog</title>
    <link href="https://fonts.googleapis.com/css?family=\
        Source+Sans+Pro:400" rel="stylesheet">
    <link href="{% static 'css/base.css' %}" rel="stylesheet">
</head>

<body>
    <div>
        <header>
```

```
<!-- start new HTML... -->
<div class="nav-left">
    <h1><a href="{% url 'home' %}">Django blog</a></h1>
</div>
<div class="nav-right">
    <a href="{% url 'post_new' %}">+ New Blog Post</a>
</div>
<!-- end new HTML... -->
</header>
{% block content %}
{% endblock content %}
</div>
</body>

</html>
```

---

Let's add a new URL for `post_new` now. Import `BlogCreateView` (which has not been created yet) at the top and then add a URL path for `post/new/`. We will give it the URL name `post_new` to refer to it later in our templates.

Code

---

```
# blog/urls.py
from django.urls import path
from .views import BlogListView, BlogDetailView, BlogCreateView # new

urlpatterns = [
    path("post/new/", BlogCreateView.as_view(), name="post_new"), # new
    path("post/<int:pk>/", BlogDetailView.as_view(), name="post_detail"),
    path("", BlogListView.as_view(), name="home"),
]
```

---

Simple, right? It's the same URL, views, and template pattern we've seen before. Now let's create our view by importing a generic class-based view called [CreateView](#) at the top and then subclass it to create a new view called `BlogCreateView`.

Code

---

```
# blog/views.py
from django.views.generic import ListView, DetailView
from django.views.generic.edit import CreateView # new

from .models import Post

class BlogListView(ListView):
    model = Post
```

```
template_name = "home.html"

class BlogDetailView(DetailView):
    model = Post
    template_name = "post_detail.html"

class BlogCreateView(CreateView):  # new
    model = Post
    template_name = "post_new.html"
    fields = ["title", "author", "body"]
```

---

Within BlogCreateView, we specify our database model, Post, the name of our template, post\_new.html, and explicitly set the database fields we want to expose, which are title, author, and body.

The last step is creating our template, templates/post\_new.html, in the text editor. Then add the following code:

#### Code

---

```
<!-- templates/post_new.html -->
{% extends "base.html" %}

{% block content %}
<h1>New post</h1>
<form action="" method="post">{% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Save">
</form>
{% endblock content %}
```

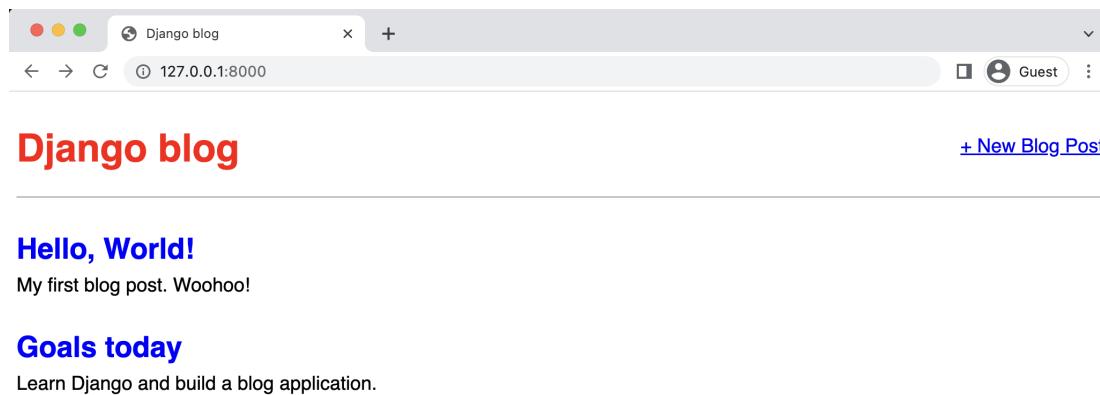
---

Let's break down what we've done:

- on the top line we extended our base template.
- use HTML <form> tags with the *POST* method since we're *sending* data. If we were receiving data from a form, for example, in a search box, we would use GET.
- add a `{% csrf_token %}`, which Django provides to protect our form from cross-site request forgery. **You should use it for all your Django forms.**

- then, to output our form data use `{{ form.as_p }}`, which renders the specified fields within paragraph `<p>` tags.
- finally, specify an input type of submit and assign the value “Save”.

To view our work, start the server with `python manage.py runserver` and go to the homepage at `http://127.0.0.1:8000/`.



The screenshot shows a web browser window titled "Django blog". The address bar displays "127.0.0.1:8000". The page content includes a red header "Django blog" and a blue link "+ New Blog Post". Below the header, there are two entries: "Hello, World!" with the subtitle "My first blog post. Woohoo!" and "Goals today" with the subtitle "Learn Django and build a blog application.".

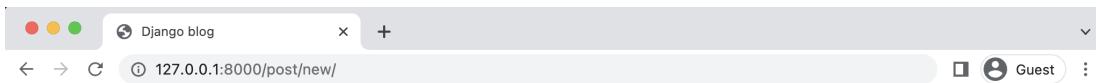
**Django blog** [+ New Blog Post](#)

**Hello, World!**  
My first blog post. Woohoo!

**Goals today**  
Learn Django and build a blog application.

### Homepage with new button

Click the “+ New Blog Post” link in the upper right-hand corner. It will redirect to the webpage at `http://127.0.0.1:8000/post/new/`.



## Django blog

[+ New Blog Post](#)

### New post

Title:

Author:

Body:

### Blog new page

Try to create a new blog post and submit it by clicking the "Save" button.



## Django blog

[+ New Blog Post](#)

### New post

Title:

Author:

Body:

I wonder if this will work?

### Blog third post save

Upon completion, it will redirect to a detail page at <http://127.0.0.1:8000/post/3/> with the post itself. Success!



The screenshot shows a web browser window titled "Django blog". The address bar displays "127.0.0.1:8000/post/3/". The main content area shows a blog post with the title "3rd post" and the body text "I wonder if this will work?". At the bottom of the post, there is a link labeled "Blog third post page". A "Guest" user is logged in.

## UpdateView

Creating an update form so users can edit blog posts should feel familiar. We'll again use a built-in Django class-based generic view, [UpdateView](#), and create the requisite template, URL, and view.

To start, let's add a new link to `post_detail.html` so that the option to edit a blog post appears on an individual blog page.

Code

```
<!-- templates/post_detail.html -->
{% extends "base.html" %}

{% block content %}
<div class="post-entry">
  <h2>{{ post.title }}</h2>
  <p>{{ post.body }}</p>
</div>
<!-- start new HTML... -->
<a href="{% url 'post_edit' post.pk %}">+ Edit Blog Post</a>
<!-- end new HTML... -->
{% endblock content %}
```

We've added a link using `<a href>...</a>` and the Django template engine's `{% url ... %}` tag. Within it, we've

specified the target name of our URL, which will be called `post_edit`, and also passed the parameter needed, which is the primary key of the post: `post.pk`.

Next, we create the template file for our edit page called `templates/post_edit.html` and add the following code:

Code

---

```
<!-- templates/post_edit.html -->
{% extends "base.html" %}

{% block content %}
<h1>Edit post</h1>
<form action="" method="post">{% csrf_token %}
{{ form.as_p }}
<input type="submit" value="Update">
</form>
{% endblock content %}
```

---

We again use HTML `<form></form>` tags, Django's `csrf_token` for security, `form.as_p` to display our form fields with paragraph tags, and finally give it the value "Update" on the submit button.

Now to our view. We must import `UpdateView` on the second-from-the-top line and then subclass it in our new view `BlogUpdateView`.

Code

---

```
# blog/views.py
from django.views.generic import ListView, DetailView
from django.views.generic.edit import CreateView, UpdateView # new

from .models import Post

class BlogListView(ListView):
    model = Post
    template_name = "home.html"

class BlogDetailView(DetailView):
    model = Post
    template_name = "post_detail.html"
```

```
class BlogCreateView(CreateView):
    model = Post
    template_name = "post_new.html"
    fields = ["title", "author", "body"]

class BlogUpdateView(UpdateView): # new
    model = Post
    template_name = "post_edit.html"
    fields = ["title", "body"]
```

---

Notice in BlogUpdateView we assume that the author of the post is not changing; we only want the title and body text to be editable, hence ["title", "body"] but not author as is the case for BlogCreateView.

The final step is to update our urls.py file as follows: add BlogUpdateView at the top and then create the new route to the existing urlpatterns, giving the new URL a pattern of /post/pk/edit and name of post\_edit.

---

#### Code

---

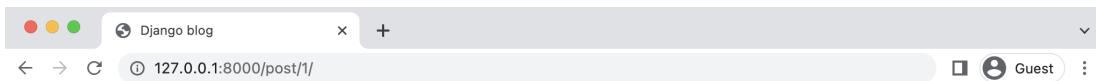
```
# blog/urls.py
from django.urls import path
from .views import (
    BlogListView,
    BlogDetailView,
    BlogCreateView,
    BlogUpdateView, # new
)

urlpatterns = [
    path("post/new/", BlogCreateView.as_view(), name="post_new"),
    path("post/<int:pk>/", BlogDetailView.as_view(), name="post_detail"),
    path("post/<int:pk>/edit/", BlogUpdateView.as_view(), name="post_edit"),
# new
    path("", BlogListView.as_view(), name="home"),
]
```

---

At the top, we added our view BlogUpdateView to the list of imported views, then creat a new URL pattern for /post/pk/edit and give it the name post\_edit.

Now if you click on a blog entry, you'll see our new + Edit Blog Post hyperlink.



## Django blog

[+ New Blog Post](#)

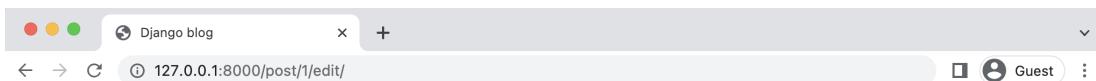
### Hello, World!

My first blog post. Woohoo!

[+ Edit Blog Post](#)

### Blog page with edit button

If you click on “+ Edit Blog Post” you’ll be redirected to /post/1/edit/ if it is your first blog post, hence the 1 in the URL. Note that the form is pre-filled with our database’s existing data for the post. Let’s make a change...



## Django blog

[+ New Blog Post](#)

### Edit post

Title:

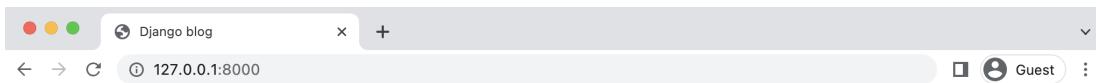
Body:

My first blog post. Woohoo!

Body:

### Blog edit page

After clicking the “Update” button, we are redirected to the detail view of the post where the change is visible. Navigate to the homepage to see the change next to all the other entries.



## Django blog

[+ New Blog Post](#)

### Hello, World! (EDITED)

My first blog post. Woohoo!

### Goals today

Learn Django and build a blog application.

### 3rd post

I wonder if this will work?

## Blog homepage with edited post

# DeleteView

The process for creating a form to delete blog posts is very similar to that for updating a post. We'll use yet another generic class-based view, [DeleteView](#), create the necessary view, URL, and template.

Let's start by adding a link to delete blog posts on our individual blog page, `post_detail.html`.

### Code

```
<!-- templates/post_detail.html -->
{% extends "base.html" %}

{% block content %}
<div class="post-entry">
    <h2>{{ post.title }}</h2>
    <p>{{ post.body }}</p>
</div>
<div><!-- start new HTML... -->
    <p><a href="{% url 'post_edit' post.pk %}">+ Edit Blog Post</a></p>
    <p><a href="{% url 'post_delete' post.pk %}">+ Delete Blog Post</a></p>
</div><!-- end new HTML... -->
{% endblock content %}
```

Then create a new file for our delete page template. It will be called `templates/post_delete.html` and contain the following code:

Code

---

```
<!-- templates/post_delete.html -->
{% extends "base.html" %}

{% block content %}
<h1>Delete post</h1>
<form action="" method="post">{% csrf_token %}
    <p>Are you sure you want to delete "{{ post.title }}"?</p>
    <input type="submit" value="Confirm">
</form>
{% endblock content %}
```

---

Note we are using `post.title` here to display the title of our blog post. We could also just use `object.title` as it too is provided by `DetailView`.

Now update the `blog/views.py` file by importing `DeleteView` and `reverse_lazy` at the top and then create a new view that subclasses `DeleteView`.

Code

---

```
# blog/views.py
from django.views.generic import ListView, DetailView
from django.views.generic.edit import CreateView, UpdateView, DeleteView # new
from django.urls import reverse_lazy # new

from .models import Post

class BlogListView(ListView):
    model = Post
    template_name = "home.html"

class BlogDetailView(DetailView):
    model = Post
    template_name = "post_detail.html"

class BlogCreateView(CreateView):
    model = Post
    template_name = "post_new.html"
    fields = ["title", "author", "body"]
```

```
class BlogUpdateView(UpdateView):
    model = Post
    template_name = "post_edit.html"
    fields = ["title", "body"]

class BlogDeleteView(DeleteView):  # new
    model = Post
    template_name = "post_delete.html"
    success_url = reverse_lazy("home")
```

---

The DeleteView specifies a model which is Post, a template which is post\_delete.html, and a third field called success\_url. What does this do? Well, after a blog post is deleted, we want to *redirect* the user to another page which is, in our case, the homepage at home.

You may have noticed we are using reverse\_lazy here and not reverse. Why is that? Both reverse and reverse\_lazy perform the same task: generating a URL based on an input like the URL name. The difference is when they are evaluated. reverse executes right away, so when BlogDeleteView is executed, immediately the model, template\_name, and success\_url methods are loaded. But the success\_url needs to find out what the resulting URL path is associated with the URL name “home.” It can’t always do that in time. That’s why we use reverse\_lazy in this example: it delays the actual call to the URLConf until the moment it is needed, not when our class BlogDeleteView is being evaluated.

The moment BlogDeleteView is called, reverse needs to have the information from the URLconf about what the proper route is for the URL name “home.” But it might not have that information in time for the success\_url. If we use reverse\_lazy instead,

An astute reader might notice that both CreateView and UpdateView also have redirects. Yet, we did not have to specify a `success_url` because Django will automatically use `get_absolute_url()` on the model object if available. And the only way to know about this trait is to read and remember the docs, which [talk about model forms](#) and `success_url`. Or the more likely situation—is that an error crops up, and you need to backtrack to sort out this internal Django behavior.

As a final step, create a URL by importing our view `BlogDeleteView` and adding a new pattern:

---

Code

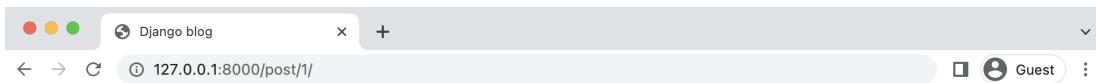
---

```
# blog/urls.py
from django.urls import path
from .views import (
    BlogListView,
    BlogDetailView,
    BlogCreateView,
    BlogUpdateView,
    BlogDeleteView, # new
)

urlpatterns = [
    path("post/new/", BlogCreateView.as_view(), name="post_new"),
    path("post/<int:pk>/", BlogDetailView.as_view(), name="post_detail"),
    path("post/<int:pk>/edit/", BlogUpdateView.as_view(),
         name="post_edit"),
    path("post/<int:pk>/delete/", BlogDeleteView.as_view(),
         name="post_delete"), # new
    path("", BlogListView.as_view(), name="home"),
]
```

---

If you start the server again with the command `python manage.py runserver` and refresh any individual post page, you'll see our "Delete Blog Post" link.



## Django blog

[+ New Blog Post](#)

### Hello, World! (EDITED)

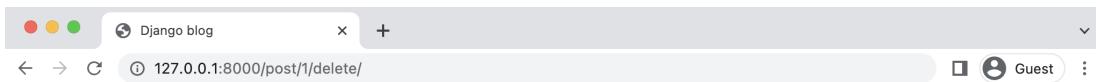
My first blog post. Woohoo!

[+ Edit Blog Post](#)

[+ Delete Blog Post](#)

### Blog delete post

Clicking on the link takes us to the delete page for the blog post, which displays the blog post's title.



## Django blog

[+ New Blog Post](#)

### Delete post

Are you sure you want to delete "Hello, World! (EDITED)"?

### Blog delete post page

If you click the “Confirm” button, it redirects you to the homepage where the blog post has been deleted!



## Django blog

[+ New Blog Post](#)

### Goals today

Learn Django and build a blog application.

### 3rd post

I wonder if this will work?

### Homepage with post deleted

So it works!

## Tests

Time for tests to make sure everything works now-and in the future-as expected. We've added new views for create, update, and delete, so that means three new tests:

- def test\_post\_createview
- def test\_post\_updateview
- def test\_post\_deleteview

Update your existing `tests.py` file with new tests below `test_post_detailview` as follows.

Code

```
# blog/tests.py
...
def test_post_createview(self): # new
    response = self.client.post(
        reverse("post_new"),
        {
            "title": "New title",
            "body": "New text",
            "author": self.user.id,
        },
    )
    self.assertEqual(response.status_code, 302)
    self.assertEqual(Post.objects.last().title, "New title")
    self.assertEqual(Post.objects.last().body, "New text")
```

```
def test_post_updateview(self): # new
    response = self.client.post(
        reverse("post_edit", args="1"),
        {
            "title": "Updated title",
            "body": "Updated text",
        },
    )
    self.assertEqual(response.status_code, 302)
    self.assertEqual(Post.objects.last().title, "Updated title")
    self.assertEqual(Post.objects.last().body, "Updated text")

def test_post_deleteview(self): # new
    response = self.client.post(reverse("post_delete", args="1"))
    self.assertEqual(response.status_code, 302)
```

---

For `test_post_createview`, we create a new response and check that the page has a 302 redirect status code and that the `last()` object created on our model matches the new response. Then `test_post_updateview` sees if we can update the initial post created in `setUpTestData` since that data is available throughout our entire test class. The last new test, `test_post_deleteview`, confirms that a 302 redirect occurs when deleting a post.

More tests can always be added later, but at least we have some coverage on all our new functionality. Stop the local web server with `Control+c` and run these tests now. They should all pass.

#### Shell

---

```
(.venv) $ python manage.py test
Found 8 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.....
-----
Ran 8 tests in 0.101s

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

---

## Conclusion

With a small amount of code, we've built a blog application that allows for creating, reading, updating, and deleting blog posts. This core functionality is known by the acronym *CRUD*: *Create-Read-Update-Delete*. The majority of websites in the world consist of this core functionality. While there are multiple ways to achieve this same functionality—we could have used function-based views or written our own class-based views—we've demonstrated how little code it takes in Django to make this happen.

Note, however, a potential security concern: currently *any* user can update or delete blog entries, not just the creator! Fortunately, Django has built-in features to restrict access based on permissions, which we'll cover later in the book.

But for now, our blog application is working, and in the next chapter, we'll add user accounts so users can sign up, log in, and log out of the web app.

# Chapter 8: User Accounts

So far, we've built a working *Blog* application with forms but still need a major piece of most web applications: user authentication.

Implementing proper user authentication is famously hard; there are many security gotchas along the way, so you don't want to implement this yourself. Fortunately, Django has a powerful, built-in [user authentication system](#) that we can use and customize as needed.

Whenever you create a new project, by default, Django installs the `auth` app, which provides us with a [User object](#) containing:

- `username`
- `password`
- `email`
- `first_name`
- `last_name`

We will use this `User` object to implement login, logout, and signup in our blog application.

## Log In

Django provides us with a default view for a login page via [LoginView](#). All we need to add are a URL pattern for the auth system, a login template, and a minor update to our `django_project/settings.py` file.

First, update the `django_project/urls.py` file. We'll place our login and logout pages at the `accounts/` URL: a one-line addition to the next-to-last line.

---

### Code

---

```
# django_project/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path("admin/", admin.site.urls),
    path("accounts/", include("django.contrib.auth.urls")), # new
    path("", include("blog.urls")),
]
```

---

As the [LoginView](#) documentation notes, by default Django will look within a templates directory called `registration` for a file called `login.html` for a login form. So we need to create a new directory called `registration` and the requisite file within it. From the command line, type `Control+c` to quit our local server. Then create the new directory.

---

### Shell

---

```
(.venv) $ mkdir templates/registration
```

---

And then, with your text editor, create a new template file, `templates/registration/login.html`, filled with the following code:

---

### Code

---

```
<!-- templates/registration/login.html -->
{% extends "base.html" %}

{% block content %}
<h2>Log In</h2>
<form method="post">{% csrf_token %}
  {{ form.as_p }}
  <button type="submit">Log In</button>
</form>
{% endblock content %}
```

---

We're using HTML `<form></form>` tags and specifying the POST method since we're sending data to the server (we'd use GET if we were requesting data, such as in a search engine form). We add `{% csrf_token %}` for security concerns to prevent a CSRF Attack. The form's contents are outputted

between paragraph tags thanks to `{{ form.as_p }}` and then we add a “submit” button.

The final step is to specify *where* to redirect the user upon successful login. We can set this with the `LOGIN_REDIRECT_URL` setting. At the bottom of the `django_project/settings.py` file, add the following:

---

Code

```
# django_project/settings.py
LOGIN_REDIRECT_URL = "home" # new
```

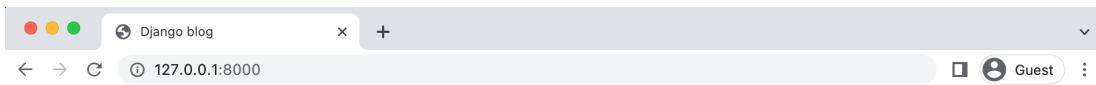
---

Now the user will be redirected to the ‘`home`’ template, which is our homepage. And we’re actually done at this point! If you now start up the Django server again with `python manage.py runserver` and navigate to our login page at `http://127.0.0.1:8000/accounts/login/`, you’ll see the following:



**Login page**

Upon entering the login info for our superuser account, we are redirected to the homepage. Notice that we didn’t add any *view* logic or create a database model because the Django auth system provided both for us automatically. Thanks, Django!



## Django blog

[+ New Blog Post](#)

### Goals today

Learn Django and build a blog application.

### 3rd post

I wonder if this will work?

## Homepage redirect from login

# Updated Homepage

Let's update our `base.html` template so we display a message to users whether they are logged in or not. We can use the [is\\_authenticated](#) attribute for this.

For now, we can place this code in a prominent position. Later on, we can style it more appropriately. Update the `base.html` file with new code starting beneath the closing `</header>` tag.

### Code

```
<!-- templates/base.html -->
{% load static %}
<html>

<head>
    <title>Django blog</title>
    <link href="https://fonts.googleapis.com/css?family=Source+Sans+Pro:400"
        rel="stylesheet">
    <link href="{% static 'css/base.css' %}" rel="stylesheet" s>
</head>

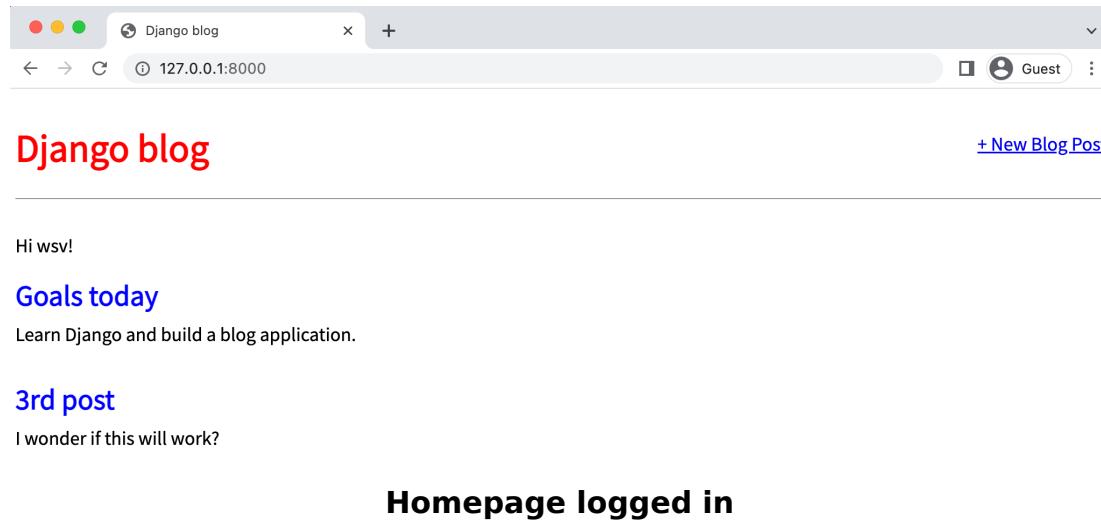
<body>
    <div>
        <header>
            <div class="nav-left">
                <h1><a href="{% url 'home' %}">Django blog</a></h1>
            </div>
            <div class="nav-right">
```

```
        <a href="{% url 'post_new' %}">+ New Blog Post</a>
    </div>
</header>
<!-- start new HTML... -->
{% if user.is_authenticated %}
<p>Hi {{ user.username }}!</p>
{% else %}
<p>You are not logged in.</p>
<a href="{% url 'login' %}">Log In</a>
{% endif %}
<!-- end new HTML... -->
{% block content %}
    {% endblock content %}
</div>
</body>

</html>
```

---

If the user is logged in, we say hello to them by name; if not, we provide a link to our newly created login page.



It worked! My superuser name is `wsv`, which I see on the page.

## Log Out Link

We added template page logic for logged-out users, but how do we log out now? We could go into the Admin panel and do it manually, but there's a better way. Let's add a logout

link instead that redirects to the homepage. Thanks to the Django auth system, this is dead simple to achieve.

In our `base.html` file, add a one-line `{% url 'logout' %}` link for logging out just below our user greeting.

---

#### Shell

---

```
<!-- templates/base.html-->
...
{%
if user.is_authenticated %}
<p>Hi {{ user.username }}!</p>
<p><a href="{% url 'logout' %}">Log out</a></p>
{%
else %}
...
-->
```

---

The Django auth app provides us with the necessary view so all we need to do is specify where to redirect a user upon logging out. Update `django_project/settings.py` to provide a redirect link called, appropriately, `LOGOUT_REDIRECT_URL`. We can add it right next to our login redirect, so the bottom of the file should look as follows:

---

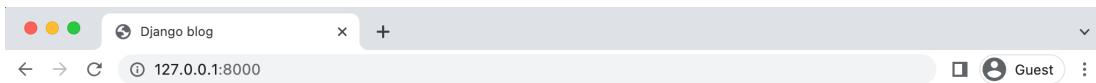
#### Code

---

```
# django_project/settings.py
LOGIN_REDIRECT_URL = "home"
LOGOUT_REDIRECT_URL = "home" # new
```

---

If you refresh the homepage, you'll see it now has a "log out" link for logged-in users.



## Django blog

[+ New Blog Post](#)

Hi wsv!

[Log out](#)

### Goals today

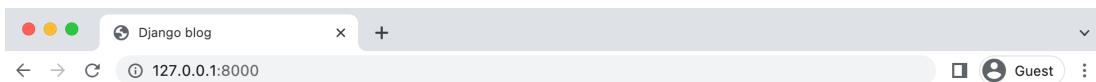
Learn Django and build a blog application.

### 3rd post

I wonder if this will work?

### Homepage log out link

And clicking it takes you back to the homepage with a `login` link.



## Django blog

[+ New Blog Post](#)

You are not logged in.

[Log in](#)

### Goals today

Learn Django and build a blog application.

### 3rd post

I wonder if this will work?

### Homepage logged out

Try logging in and out several times with your user account.

## Sign Up

We need to write our own view for a signup page to register new users, but Django does provide us with a form class, [UserCreationForm](#), to make things easier. By default, it comes with three fields: username, password1, and password2.

There are many ways to organize your code and URL structure for a robust user authentication system. Stop the local server with `Control+c` and create a dedicated new app, `accounts`, for our signup page.

---

#### Shell

---

```
(.venv) $ python manage.py startapp accounts
```

---

Add the new app to the `INSTALLED_APPS` setting in our `django_project/settings.py` file.

---

#### Code

---

```
# django_project/settings.py
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "blog",
    "accounts", # new
]
```

---

Next, add a new URL path in `django_project/urls.py` pointing to this new app directly **below** where we include the built-in auth app.

---

#### Code

---

```
# django_project/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path("admin/", admin.site.urls),
    path("accounts/", include("django.contrib.auth.urls")),
    path("accounts/", include("accounts.urls")), # new
    path("", include("blog.urls")),
]
```

---

The order of our `urls` matters here because Django reads this file from top to bottom. Therefore when we request the `/accounts/signup` url, Django will first look in `auth`, not find it, and **then** proceed to the `accounts` app.

In your text editor, create a file called `accounts/urls.py` and add the following code:

---

Code

```
# accounts/urls.py
from django.urls import path
from .views import SignUpView

urlpatterns = [
    path("signup/", SignUpView.as_view(), name="signup"),
]
```

---

We're using a not-yet-created view called `SignUpView`, which we already know is class-based since it is capitalized and has the `as_view()` suffix. Its path is just `signup/`, so the complete URL path will be `accounts/signup/`.

Now for the view, which uses the built-in `UserCreationForm` and generic `CreateView`. Replace the default `accounts/views.py` code with the following:

---

Code

```
# accounts/views.py
from django.contrib.auth.forms import UserCreationForm
from django.urls import reverse_lazy
from django.views.generic import CreateView

class SignUpView(CreateView):
    form_class = UserCreationForm
    success_url = reverse_lazy("login")
    template_name = "registration/signup.html"
```

---

We're subclassing the generic class-based view `CreateView` in our `SignUpView` class and specify `UserCreationForm` and the not-yet-created template `registration/signup.html`. We also use

`reverse_lazy` to redirect the user to the login page upon successful registration.

Why use `reverse_lazy` here instead of `reverse`? The reason is that the URLs are not loaded when the file is imported for generic class-based views, so we have to use the lazy form of `reverse` to load them later when they're available.

In your text editor, create the file `signup.html` within the `templates/registration/` directory. Populate it with the code below.

---

Code

---

```
<!-- templates/registration/signup.html -->
{% extends "base.html" %}

{% block content %}
<h2>Sign Up</h2>
<form method="post">{% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Sign Up</button>
</form>
{% endblock content %}
```

---

This format is very similar to what we've done before. We extend our base template at the top, place our logic between `<form></form>` tags, use the `csrf_token` for security, display the form's content in paragraph tags with `form.as_p`, and include a submit button.

We're done! To test it out, start the local server with the `python manage.py runserver` command and navigate to `http://127.0.0.1:8000/accounts/signup/`.



## Django signup page

Notice how there is a lot of extra text that Django includes by default. We can customize this using something like the built-in [messages framework](#), but for now, try out the form.

I've created a new user called "william" and, upon submission, was redirected to the login page. Then after logging in successfully with my new username and password, I was redirected to the homepage with our personalized "Hi username" greeting.

Hi william!

[Log out](#)

**Goals today**

Learn Django and build a blog application.

**3rd post**

I wonder if this will work?

**Homepage for user william**

Our ultimate flow is, therefore: Signup -> Login -> Homepage. And, of course, we can tweak this however we want. The SignUpView redirects to login because we set success\_url = reverse\_lazy('login'). The Login page redirects to the homepage because in our django\_project/settings.py file, we set LOGIN\_REDIRECT\_URL = 'home'.

It may initially be overwhelming to keep track of all the various parts of a Django project: but that's normal. With time, they'll start to make more sense.

## Sign Up Link

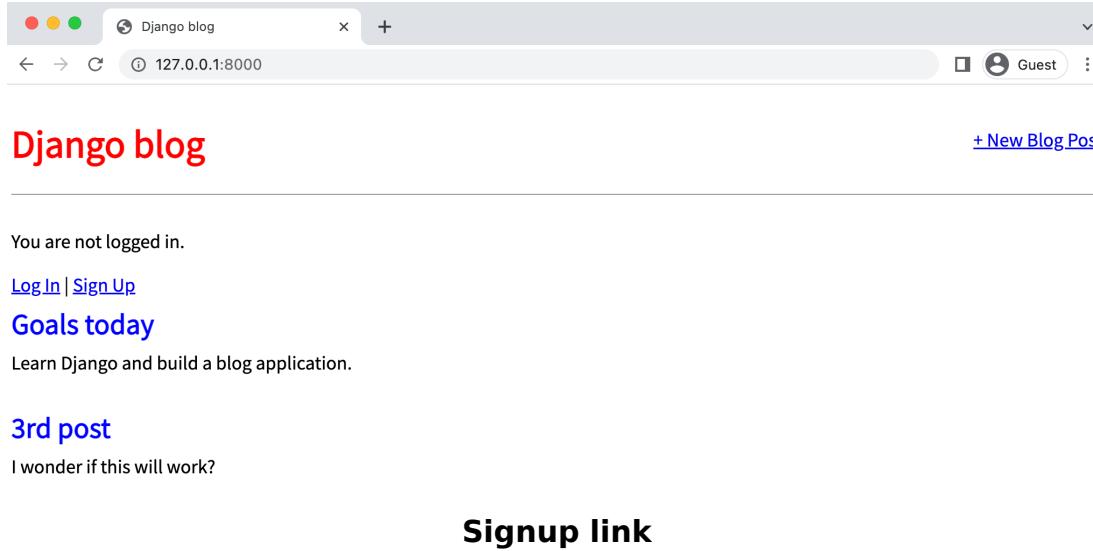
One last improvement we can make is to add a signup link to the logged-out homepage. We can't expect our users to know the correct URL after all! How do we do this? Well, we need to figure out the URL name, and then we can pop it into our template. In accounts/urls.py, we provided it the name of signup, so that's all we need to add to our base.html template with the [url template tag](#) just as we've done for our other links.

Add the link for “Sign Up” just below the existing link for “Log In” as follows:

### Shell

```
<!-- templates/base.html -->  
...  
<p>You are not logged in.</p>  
<a href="{% url 'login' %}">Log In</a> |  
<a href="{% url 'signup' %}">Sign Up</a>  
...
```

If you refresh the logged-out homepage, the signup link is now visible. Much better!



## Static Files

We saw in the *Message Board* project that the Django admin has its own static files—images, CSS, fonts, JavaScript—that must be properly prepped for production. For this *Blog* website we are adding our own static file: `static/css/base.css` containing our custom CSS.

When we started the *Blog* website we created a `static` directory for our static files and added the `STATICFILES_DIRS`

setting, which defines *additional locations* beyond an app's static folder where the staticfiles app should look within when running collectstatic. This is what the settings look like currently:

Code

---

```
# django_project/settings.py
STATIC_URL = "static/"
STATICFILES_DIRS = [BASE_DIR / "static"]
```

---

To fully ready our project for production, we must add two more settings. The first is STATIC\_ROOT, the directory location of all static files compiled when collectstatic is run. We will set it to the standard default of staticfiles meaning there will be a project-level folder with that naming containing all the static files ready for production.

The second is the static file storage engine used by collectstatic. STORAGES is implicitly set right now so let's make it explicit:

Code

---

```
# django_project/settings.py
STATIC_URL = "static/"
STATICFILES_DIRS = [BASE_DIR / "static"]
STATIC_ROOT = BASE_DIR / "staticfiles" # new
STORAGES = {
    "default": {
        "BACKEND": "django.core.files.storage.FileSystemStorage",
    },
    "staticfiles": {
        "BACKEND": "django.contrib.staticfiles.storage.StaticFilesStorage",
    },
}
```

---

When we shortly install WhiteNoise we will update STORAGES to change the staticfiles alias. For now, run the command collectstatic to compile all our static files.

Shell

---

```
(.venv) $ python manage.py collectstatic
126 static files copied to '~/desktop/code/blog/staticfiles'.
```

---

The newly created `staticfiles` directory contains both `admin` and `css` folders.

---

#### Code

---

```
# staticfiles/
└── admin
    ├── css
    ├── img
    └── js
└── css
    └── base.css
```

---

## GitHub

It's been a while since we made a git commit. Let's do that and then push a copy of our code onto GitHub. First, check all the new work we've done with `git status`.

---

#### Shell

---

```
(.venv) $ git status
```

---

Then add the new content and enter a commit message.

---

#### Shell

---

```
(.venv) $ git add -A
(.venv) $ git commit -m "forms, user accounts, and static files"
```

---

[Create a new repo](#) on GitHub and follow the recommended steps. I've chosen the name `blog` here and my username is `wsvincent`. Make sure to use your own GitHub username and repo name for the command setting up a remote origin.

---

#### Shell

---

```
(.venv) $ git remote add origin https://github.com/wsvincent/blog.git
(.venv) $ git branch -M main
(.venv) $ git push -u origin main
```

---

All set!

## Conclusion

With minimal code, we have added login, logout, and signup functionality to our blog website. Under the hood, Django has taken care of the many security gotchas that can crop up if you create a user authentication flow from scratch. We also configured static files fully and are ready to deploy the project in the next chapter.

# Chapter 9: Blog Deployment

It's time for our third deployment in this book. We will go deeper with environment variables and add improved security throughout our production environment. We can reuse the deployment checklist from earlier in the book but with three major additions: add a `.env` file to store environment variables and toggle both `DEBUG` and `SECRET_KEY`. Here is the full list to transform our Django project from a local development setup into one ready for production:

- install Gunicorn as a WSGI server
- install Psycopg to connect with a PostgreSQL database
- install environs for environment variables
- update DATABASES in `django_project/settings.py`
- install WhiteNoise for static files
- create a `requirements.txt` file
- add a `.dockerignore` file
- create a `.env` file
- update a `.gitignore` file
- update ALLOWED\_HOSTS, CSRF\_TRUSTED\_ORIGINS, DEBUG, and SECRET\_KEY

## Gunicorn, Psycopg, and environs

Let's start by installing Gunicorn, Psycopg, and environs.

---

### Shell

```
(.venv) $ python -m pip install gunicorn==20.1.0
(.venv) $ python -m pip install "psycopg[binary]"==3.1.8
(.venv) $ python -m pip install "environs[django]"==9.5.0
```

---

Then update `django_project/settings.py` with three new lines so environment variables can be loaded in.

---

### Code

```
# django_project/settings.py
from pathlib import Path
from environs import Env # new

env = Env() # new
env.read_env() # new
```

---

## DATABASES

Next, update the DATABASES setting in the django\_project/settings.py file so that SQLite is used locally but PostgreSQL in production.

Code

---

```
# django_project/settings.py
DATABASES = {
    "default": env.dj_db_url("DATABASE_URL", default="sqlite:///db.sqlite3"),
}
```

---

## WhiteNoise

WhiteNoise is needed to serve static files in production. Install the latest version using pip:

Shell

---

```
(.venv) $ python -m pip install whitenoise==6.4.0
```

---

Then in django\_project/settings.py, there are three updates to make:

- add whitenoise to the INSTALLED\_APPS **above** the built-in staticfiles app
- under MIDDLEWARE add a new line for WhiteNoiseMiddleware after SessionMiddleware
- configure staticfiles within STORAGE to use WhiteNoise

The updated file should look as follows:

Code

---

```
# django_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",
    "blog",
    "accounts",
]
MIDDLEWARE = [
    "django.middleware.security.SecurityMiddleware",
    "django.contrib.sessions.middleware.SessionMiddleware",
    "whitenoise.middleware.WhiteNoiseMiddleware", # new
    "django.middleware.common.CommonMiddleware",
    ...
]
STATIC_URL = "static/"
STATICFILES_DIRS = [BASE_DIR / "static"]
STATIC_ROOT = BASE_DIR / "staticfiles"
STORAGES = {
    "default": {
        "BACKEND": "django.core.files.storage.FileSystemStorage",
    },
    "staticfiles": {
        "BACKEND": "whitenoise.storage.CompressedManifestStaticFilesStorage",
    },
# new
},
}
```

---

We have updated the default staticfiles storage engine to use WhiteNoise when running the collectstatic management command. Run it one more time now:

Shell

---

```
(.venv) $ python manage.py collectstatic
```

---

There will be a short warning, This will overwrite existing files! Are you sure you want to do this? Type “yes” and hit Enter. The collected static files are now regenerated in the same staticfiles folder using WhiteNoise.

## requirements.txt

Now that all additional third-party packages are installed, we can generate a requirements.txt file containing the contents of our virtual environment.

#### Shell

---

```
(.venv) $ python -m pip freeze > requirements.txt
```

---

My requirements.txt file looks like this:

#### Code

---

```
# requirements.txt
asgiref==3.6.0
black==23.3.0
click==8.1.3
dj-database-url==1.3.0
dj-email-url==1.0.6
Django==4.2
django-cache-url==3.4.4
environs==9.5.0
gunicorn==20.1.0
marshmallow==3.19.0
mypy-extensions==1.0.0
packaging==23.1
pathspec==0.11.1
platformdirs==3.2.0
psycopg==3.1.8
psycopg-binary==3.1.8
python-dotenv==1.0.0
sqlparse==0.4.3
typing_extensions==4.5.0
whitenoise==6.4.0
```

---

## .dockerignore and .env

It's time to create two hidden files: .dockerignore and a new one, .env. Both should exist in the root directory next to the existing .gitignore file. Go ahead and create them now.

The .dockerignore file should include our local virtual environment, pycache, SQLite database, git directory, and the new .env file.

#### .dockerignore

---

```
.venv/
__pycache__/
```

```
*.sqlite3  
.git  
.env
```

---

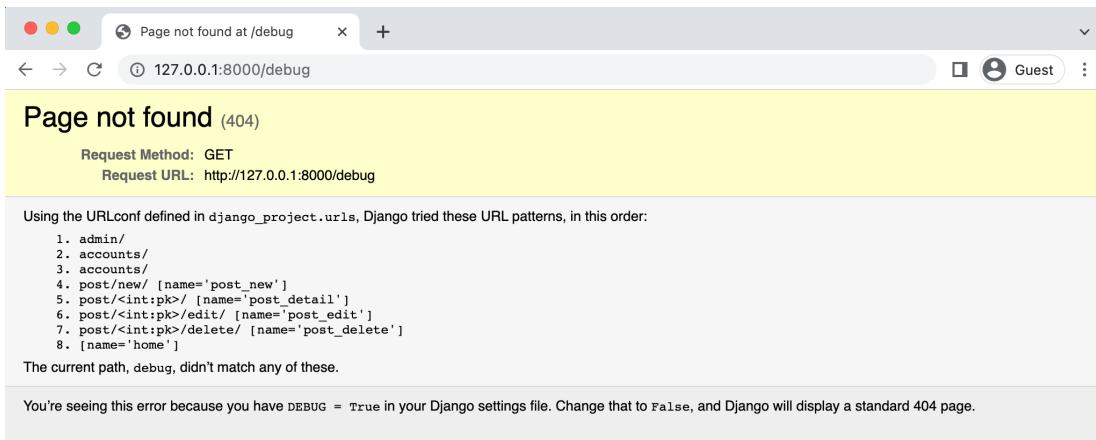
The `.env` file stores environment variables, most of which are secret, so it should be added to `.gitignore` immediately so that Git does not track it. That is also why adding `.env` to the `.dockerignore` file is important: we do not want the `.env` file accidentally loaded into your production server. That would not be secure!

```
.gitignore  
_____  
.venv/  
__pycache__/  
*.sqlite3  
.env # new
```

---

## DEBUG

The [`DEBUG`](#) setting is a boolean that turns debug mood on and off. The official documentation notes that you should **never deploy a site into production with DEBUG turned on**. To understand why, start up the local server with `python manage.py runserver` and navigate to a page that does not exist, like `http://127.0.0.1:8000/debug`. You will see the following:



## Page not found

This page lists all the URLs tried and apps loaded, which is a treasure map for any hacker attempting to break into your site. You'll even see that on the bottom of the error page, it says that Django will display a standard 404 page if DEBUG=False. This is no good, so let's fix that. Within the django\_project/settings.py file, change DEBUG to False.

### Code

```
# django_project/settings.py
DEBUG = False
```

If you look in the terminal, the local web server has automatically been stopped and there is an error message: CommandError: You must set settings.ALLOWED\_HOSTS if DEBUG is False. You can't even run the local web server when DEBUG is set to False!

Our goal is for DEBUG to be True for local development but set to False in production. The two-step process for adding any environment variable is first adding it to the .env file and then to django\_project/settings.py. Within the .env file, create a new environment variable called DEBUG; we will set its value to True. It is important *not to have* any spaces here.

.env

---

DEBUG=True

---

Then in `django_project/settings.py`, change the DEBUG setting to read the variable "DEBUG" from the `.env` file.

Code

---

```
# django_project/settings.py
DEBUG = env.bool("DEBUG")
```

---

Try to run the local web server again with `runserver` and it will work as before.

Shell

---

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

---

It's easy to be confused here! Our environment variable is named `DEBUG`, the same as the setting it replaces. But we could have named our environment variable `ANYTHING` instead, and that would have looked like this:

.env

---

ANYTHING=True

---

Code

---

```
# django_project/settings.py
DEBUG = env.bool("ANYTHING")
```

---

`ANYTHING` is a variable, so it can have almost any name we desire. In practice, however, most developers will name the environment variable to match the name of the setting it replaces, and we will do the same so `DEBUG=True`.

One more best practice we will adopt is to set a default value, in this case, `False`, meaning that if an environment variable can't be found, our production setting will be used. It's a best practice to default to production settings since

they are more secure, and if something goes wrong in our code, we won't default to exposing all our secrets out in the open.

The final `django_project/settings.py` line, therefore, looks as follows:

---

Code

---

```
# django_project/settings.py
DEBUG = env.bool("DEBUG", default=False)
```

---

If you refresh the webpage at `http://127.0.0.1:8000/debug`, you'll see the full error page is back again, which means `DEBUG=TRUE` because the local environment has looked for a `.env` file, found `DEBUG` and imported its value into `django_project/settings.py`.

When we deploy into production, the `.env` is ignored by both Docker and Git, so it won't be present. That means our project will default to `False`, the secure choice.

## **SECRET\_KEY**

The [SECRET\\_KEY](#) is a random 50-character string generated each time `startproject` is run. It provides [cryptographic signing](#) in our project and is very important to keep secure. If you look at the value in `django_project/settings.py`, the comment clearly shows that the existing key is not secret.

---

Code

---

```
# django_project/settings.py
# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = "django-insecure-^qi19(+oo-ere5b&$@275chw)k@7ob1)74aol5d$(k*)5kk5"
```

---

Copy and paste this value into the `.env` file.

.env

---

```
DEBUG=True  
SECRET_KEY=django-insecure-^qi19(+0o-ere5b&$@275chw)k@7ob1)74aol5d$(k*)5kk5
```

---

Then update `django_project/settings.py` so that `SECRET_KEY` points to this new environment variable.

Code

---

```
# django_project/settings.py  
SECRET_KEY = env.str("SECRET_KEY")
```

---

Our `SECRET_KEY` is now out of the `settings.py` file and is safe, right? Actually *no!* Because we made an earlier Git commit, the `SECRET_KEY` value is stored in our Git history no matter what we do. Anyone who can access our source code and Git history can see it. Uh oh.

The solution is to create a new `SECRET_KEY` and add it to the `.env` file. Remember that we added `.env` to our `.gitignore` file, so Git will not track it. Even if someone malicious has access to our codebase, they cannot see the `.env` file and the `SECRET_KEY` within.

One way to generate a new one is by invoking Python's built-in [`secrets`](#) module by running `python -c 'import secrets; print(secrets.token_urlsafe())'` on the command line.

Shell

---

```
(.venv) $ python -c "import secrets; print(secrets.token_urlsafe())"  
imDnfLxy-8Y-YozfJmP2Rw_81YA_qx1XKl5FeY0mXyY
```

---

Check that there are no quotations (" or ') around strings nor spaces around the =. If there are, update them to a different value. Copy and paste this new value into the `.env` file.

.env

---

```
DEBUG=True  
SECRET_KEY=imDnfLxy-8Y-YozfJmP2Rw_81YA_qx1XKl5FeY0mXyY
```

---

Now restart the local server with `python manage.py runserver` and refresh your website. It will work with the new `SECRET_KEY` that is loaded from the `.env` file and not tracked by Git.

## **ALLOWED\_HOSTS and CSRF\_TRUSTED\_ORIGINS**

In previous examples, we were somewhat lax with security, but now we will tighten things up properly using environment variables. There are four big settings that need to be toggled for production: `DEBUG`, `SECRET_KEY`, `ALLOWED_HOSTS` and `CSRF_TRUSTED_ORIGINS`.

Let's start with `ALLOWED_HOSTS` and `CSRF_TRUSTED_ORIGINS`, which we loosely set in the Chapter 5 Message Board deployment chapter to the following:

Code

---

```
# django_project/settings.py
ALLOWED_HOSTS = ["*"]
CSRF_TRUSTED_ORIGINS = ["https://*.fly.dev"]
```

---

These settings are insecure. A better approach is to restrict `ALLOWED_HOSTS` access to either a local port (`localhost` and `127.0.0.1`) or our actual production URL. The `CSRF_TRUSTED_ORIGINS` setting should only be set to the production URL. Since we won't know that URL until after running `fly launch`, let's put in a placeholder for now.

Code

---

```
# django_project/settings.py
ALLOWED_HOSTS = [".fly.dev", "localhost", "127.0.0.1"] # new
CSRF_TRUSTED_ORIGINS = ["https://*.fly.dev"] # new
```

---

## **Fly Launch**

Fly allows us to deploy our Django website once it's packaged in a Docker image. If we were to do this ourselves, we'd need to manually create three files:

- Dockerfile with commands to build the image
- fly.toml with configuration specific to Fly.io
- .dockerignore with files and directories Docker should ignore

The `fly launch` command detects our Django project and automatically generates all three files needed for deployment. Since we already created a `.dockerignore` file earlier, we will tell Fly not to create one for us.

Make sure you're logged into your Fly account. You can use `flyctl auth login` from the command line. Then run the `fly launch` command to configure our website for deployment.

- **Choose an app name:** this will be your dedicated `fly.dev` subdomain
- **Choose the region for deployment:** select the one closest to you or [another region](#) if you prefer
- **Decline overwriting our `.dockerignore` file:** our choices are already optimized for the project
- **Setup a Postgres database cluster:** the "Development" option is appropriate for this project. [Fly Postgres is a regular app deployed to Fly.io, not a managed database.](#)
- **Select “Yes” to scale a single node pg to zero after one hour:** this will save money for toy projects
- **Decline to setup a Redis database:** we don't need one for this project

#### Shell

---

```
(.venv) $ fly launch
Creating app in ~/desktop/code/blog
Scanning source code
Detected a Django app
? Choose an app name (leave blank to generate one): dfb-ch9-blog
automatically selected personal organization: Will Vincent
? Choose a region for deployment: Boston, Massachusetts (US) (bos)
App will use 'bos' region as primary
Created app dfb-ch9-blog in organization 'personal'
Admin URL: https://fly.io/apps/dfb-ch9-blog
```

```
Hostname: dfb-ch9-blog.fly.dev
? Overwrite "/Users/wsv/Desktop/code/blog/.dockerignore"? No
Set secrets on dfb-ch9-blog: SECRET_KEY
? Would you like to set up a Postgresql database now? Yes
? Select configuration: Development - Single node, 1x shared CPU, 256MB RAM,
1GB
? Scale single node pg to zero after one hour? Yes
Creating postgres cluster in organization personal
Creating app...
...
? Would you like to set up an Upstash Redis database now? No
Wrote config file fly.toml
...
Your app is ready! Deploy with `flyctl deploy`
```

---

The fly launch wizard has automatically configured two new files for us: `fly.toml` and `Dockerignore`. Because we added the `.env` file to our `.dockerignore` file, there was no record of an environment variable for `SECRET_KEY`. Therefore, Fly automatically created one for us in the `Dockerfile` that can be used to build the Docker image.

It's important to be clear right now about how `SECRET_KEYS` are being managed. We are using one locally that is loaded in via the `.env` file. There is a second one present in the `Dockerfile` that is used just for building an initial Docker image. And then there is a third one present on Fly servers that Fly also creates for us; this one is secure and loaded into the production environment.

To confirm that Fly has set to production environment variables for us now—`SECRET_KEY` and `DATABASE_URL`—you can list them with the command `fly secrets list`:

#### Shell

---

(.venv) \$ fly secrets list	NAME	DIGEST	CREATED AT
<hr/>			
	DATABASE_URL	9233fb6f0aa3f3ee	1m59s ago
	SECRET_KEY	f52a879c981352db	3m12s ago

---

## Updated ALLOWED\_HOSTS and CSRF\_TRUSTED\_ORIGINS

The last step before actual deployment is updating our ALLOWED\_HOSTS and CSRF\_TRUSTED\_ORIGINS settings to use a production URL. The fly launch wizard generated one for us. In my case it was Hostname: dfb-ch9-blog.fly.dev, but yours will be different.

Code

---

```
# django_project/settings.py
ALLOWED_HOSTS = ["dfb-ch9-blog.fly.dev", "localhost", "127.0.0.1"] # new
CSRF_TRUSTED_ORIGINS = ["https://dfb-ch9-blog.fly.dev"] # new
```

---

## Fly Deployment

Now that everything is configured, run the command `flyctl deploy` to deploy our Django project on Fly servers.

Shell

---

```
(.venv) $ flyctl deploy
...
1 desired, 1 placed, 1 healthy, 0 unhealthy [health checks: 1 total, 1
passing]
--> v0 deployed successfully
```

---

If there are any issues with your deployment, you can use `fly logs` to see recent log file entries or go to `fly.io` to see the full dashboard. But assuming the deployment was successful, you can now run `fly open` or visit the production URL in a web browser

Shell

---

```
(.venv) $ fly open
```

---

The website is there but it looks empty since we have not created a superuser or added any blog content on the PostgreSQL production server. Let's fix that by SSHing in to

create a superuser account. This command prompts for a password and requires adding the `--pty` flag to `fly ssh` console. The full command for `createsuperuser` is therefore:

---

#### Shell

---

```
(.venv) $ fly ssh console --pty -C "python /code/manage.py createsuperuser"
```

---

Create a username, email address, and password. Then visit the `/admin` page and create new blog posts. Refresh the main homepage and they should appear on the live website.

## Conclusion

Deployment is complex even when using a Platform-as-a-Service provider like Fly. But that's why we rely on a checklist to make sure all the steps are covered.

You may have noticed that on the current live site anyone can create, read, update, delete blog posts without being logged in. That is not ideal, but is easily fixed with built-in Django features that will be covered soon.

For the rest of the book, we will embark on one single project, a *Newspaper* site that uses a custom user model, advanced user registration flow, enhanced styling via Bootstrap, and email configuration. It also uses proper permissions, authorizations, and improved security for our deployment process.

# Chapter 10: Custom User Model

Django's built-in [User model](#) allows us to start working with users right away, as we just did with our *Blog* app in the previous chapters. However, most large projects need a way to add information related to users, such as age or any number of additional fields. There are two popular approaches.

The first-and older approach-is called the “User Profile” approach and [extends the existing User model](#) by creating a [OneToOneField](#) to a separate model containing fields with additional information. The idea is to keep authentication reserved for User and not bundled with non-authentication-related user information.

The second approach, to create a custom user model, is recommended in the [official Django documentation](#). We can extend [AbstractUser](#) to create a custom user model that behaves identically to the default User model but provides the option for customization in the future.

A third and more advanced approach is to create a custom user model with [AbstractBaseUser](#) that provides total control. This should only be done if you are an advanced user of Django.

This chapter will use AbstractUser to create a custom user model for a new *Newspaper* website project. The choice of a newspaper app pays homage to Django's roots as a web framework built for editors and journalists at the Lawrence Journal-World.

## Initial Set Up

The first step is to create a new Django project from the command line. We need to do our familiar steps of creating and navigating to a new directory called news and installing and activating a new virtual environment called .venv.

---

#### Shell

---

```
# Windows
$ cd onedrive\desktop\code
$ mkdir news
$ cd news
$ python -m venv .venv
$ .venv\Scripts\Activate.ps1
(.venv) $

# macOS
$ cd ~/desktop/code
$ mkdir news
$ cd news
$ python3 -m venv .venv
$ source .venv/bin/activate
(.venv) $
```

---

Next, install Django and Black, create a new Django project called django\_project, and make a new app called accounts.

---

#### Shell

---

```
(.venv) $ python -m pip install django~=4.2.0
(.venv) $ python -m pip install black
(.venv) $ django-admin startproject django_project .
(.venv) $ python manage.py startapp accounts
```

---

Note that we **did not** run `migrate` to configure our database. It's important to wait until **after** we've created our new custom user model before doing so, given how tightly connected the user model is to the rest of Django.

In your web browser, navigate to `http://127.0.0.1:8000`, and the familiar Django welcome screen will be visible.

## Git

The start of a new project is an excellent time to initialize Git and create a repo on GitHub. We've done this several

times before so we can use the same commands to initialize a new local Git repo and check its status.

---

#### Shell

---

```
(.venv) $ git init  
(.venv) $ git status
```

---

The `.venv` directory and the SQLite database should not be included in Git, so create a project-level `.gitignore` file in your text editor. We can also add the `__pycache__` directory while at it.

---

#### .gitignore

---

```
.venv/  
__pycache__/  
*.sqlite3
```

---

Run `git status` again to confirm the `.venv` directory and SQLite database are not included. Then add the rest of our work along with a commit message.

---

#### Shell

---

```
(.venv) $ git status  
(.venv) $ git add -A  
(.venv) $ git commit -m "initial commit"
```

---

[Create a new repo](#) on GitHub and provide a name. I've chosen `news`, and my username is `wsvincent`. Make sure to use your repo name and username with the command below.

---

#### Shell

---

```
(.venv) $ git remote add origin https://github.com/wsvincent/blog.git  
(.venv) $ git branch -M main  
(.venv) $ git push -u origin main
```

---

All done!

## Custom User Model

Creating our custom user model requires four steps:

- update `django_project/settings.py`
- create a new `CustomUser` model
- create new forms for `UserCreationForm` and `UserChangeForm`
- update `accounts/admin.py`

In `django_project/settings.py`, we'll add the `accounts` app to our `INSTALLED_APPS`. Then at the bottom of the file, use the `AUTH_USER_MODEL` config to tell Django to use our new custom user model instead of the built-in User model. We'll call our custom user model `CustomUser`. Since it will exist within our `accounts` app, we should refer to it as `accounts.CustomUser`.

Code

---

```
# django_project/settings.py
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "accounts", # new
]
...
AUTH_USER_MODEL = "accounts.CustomUser" # new
```

---

Now update `accounts/models.py` with a new User model called `CustomUser` that extends the existing `AbstractUser`. We also include a custom field for age here.

Code

---

```
# accounts/models.py
from django.contrib.auth.models import AbstractUser
from django.db import models

class CustomUser(AbstractUser):
    age = models.PositiveIntegerField(null=True, blank=True)
```

---

If you read the [documentation on custom user models](#), it recommends using `AbstractBaseUser`, not `AbstractUser`, which

complicates things for beginners. Working with Django is far simpler and remains customizable if we use `AbstractUser` instead.

So why use `AbstractBaseUser` at all? If you want a fine level of control and customization, `AbstractBaseUser` *can* be justified. But it requires rewriting a core part of Django. If we want a custom user model that can be updated with additional fields, the better choice is `AbstractUser`, which subclasses `AbstractBaseUser`. In other words, we write much less code and have less opportunity to mess things up. It's the better choice unless you really know what you're doing with Django!

Note that we use both `null` and `blank` with our age field. These two terms are easy to confuse but quite distinct:

- `null` is **database-related**. When a field has `null=True`, it can store a database entry as `NULL`, meaning no value.
- `blank` is **validation-related**. If `blank=True`, then a form will allow an empty value, whereas if `blank=False` then a value is required.

In practice, `null` and `blank` are commonly used together in this fashion so that a form allows an empty value, and the database stores that value as `NULL`.

A common gotcha to be aware of is that the **field type** dictates how to use these values. Whenever you have a string-based field like `CharField` or `TextField`, setting both `null` and `blank` as we've done will result in two possible values for "no data" in the database, which is a bad idea. Instead, the Django convention is to use the empty string "", not `NULL`.

## Forms

If we step back for a moment, how would we typically interact with our new `CustomUser` model? One case is when a user signs up for a new account on our website. The other is within the `admin` app, which allows us, as superusers, to modify existing users. So we'll need to update the two built-in forms for this functionality: [UserCreationForm](#) and [UserChangeForm](#).

Create a new file called `accounts/forms.py` and update it with the following code to extend the existing `UserCreationForm` and `UserChangeForm` forms.

---

Code

---

```
# accounts/forms.py
from django.contrib.auth.forms import UserCreationForm, UserChangeForm
from .models import CustomUser

class CustomUserCreationForm(UserCreationForm):
    class Meta:
        model = CustomUser
        fields = UserCreationForm.Meta.fields + ("age",)

class CustomUserChangeForm(UserChangeForm):
    class Meta:
        model = CustomUser
        fields = UserChangeForm.Meta.fields
```

---

For both new forms, we are using the [Meta class](#) to override the default fields by setting the `model` to our `CustomUser` and using the default fields via `Meta.fields` which includes *all* default fields. To add our custom `age` field, we simply tack it on at the end, and it will display automatically on our future signup page. Pretty slick, no?

The concept of fields on a form can be confusing at first, so let's take a moment to explore it further. Our `CustomUser` model contains all the fields of the default `User` model **and** our additional `age` field, which we set.

But what are these default fields? It turns out there [are many](#) including `username`, `first_name`, `last_name`, `email`, `password`, `groups`, and more. Yet when a user signs up for a new account on Django, the default form only asks for a `username`, `email`, and `password`, which tells us that the default setting for fields on `UserCreationForm` is just `username`, `email`, and `password` even though many more fields are available.

Understanding how forms and models interact in Django takes time and repetition. Don't be discouraged if you are slightly confused right now! In the next chapter, we will create our signup, login, and logout pages to tie together our `CustomUser` model and forms more clearly.

The final step is to update our `admin.py` file since the admin is tightly coupled to the default User model. We will extend the existing [UserAdmin](#) class to use our new `CustomUser` model. To control which fields are listed, we use [list\\_display](#). But to actually edit new custom fields, like `age`, we must add [fieldsets](#). And to include a new custom field in the section for creating a new user we rely on `add_fieldsets`.

Here is what the complete code looks like:

---

Code

```
# accounts/admin.py
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin

from .forms import CustomUserCreationForm, CustomUserChangeForm
from .models import CustomUser

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

```
fieldsets = UserAdmin.fieldsets + ((None, {"fields": ("age",)}),)
add_fieldsets = UserAdmin.add_fieldsets + ((None, {"fields": ("age",)}),)

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

---

There are many ways to customize the user admin, and some developers like to add additional options such as [list\\_filter](#), [search\\_fields](#), and [ordering](#).

But for this project, we are now done. Type `Control+c` to stop the local server and go ahead and run `makemigrations` and `migrate` for the first time to create a new database that uses the custom user model.

#### Shell

---

```
(.venv) $ python manage.py makemigrations accounts
Migrations for 'accounts':
  accounts/migrations/0001_initial.py
    - Create model CustomUser
```

---

#### Shell

---

```
(.venv) $ python manage.py migrate
Operations to perform:
  Apply all migrations: accounts, admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0001_initial... OK
  Applying auth.0002.Alter_permission_name_max_length... OK
  Applying auth.0003.Alter_user_email_max_length... OK
  Applying auth.0004.Alter_user_username_opts... OK
  Applying auth.0005.Alter_user_last_login_null... OK
  Applying auth.0006.Require_contenttypes_0002... OK
  Applying auth.0007.Alter_validators_add_error_messages... OK
  Applying auth.0008.Alter_user_username_max_length... OK
  Applying auth.0009.Alter_user_last_name_max_length... OK
  Applying auth.0010.Alter_group_name_max_length... OK
  Applying auth.0011.Update_proxy_permissions... OK
  Applying auth.0012.Alter_user_first_name_max_length... OK
  Applying accounts.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying sessions.0001_initial... OK
```

---

## Superuser

Let's create a superuser account to confirm everything is working as expected. On the command line, type the following command and go through the prompts.

---

Shell

---

```
(.venv) $ python manage.py createsuperuser
```

---

Make sure your superuser email account is one that actually works. We will use it later on to verify email integration. But the fact that this flow here works is the first proof our custom user model is set up correctly. Let's view things in the admin, too, to be extra sure.

Start up the web server.

---

Shell

---

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

---

Then navigate to the admin at `http://127.0.0.1:8000/admin` and log in. If you click on the link for "Users" you should see your superuser account and the default fields of Email Address, Username, Age, and Staff Status. These were set in `list_display` in our `admin.py` file.

The screenshot shows the Django administration interface for managing users. The top navigation bar includes links for 'Home', 'Accounts', and 'Users'. The main content area is titled 'Select user to change' and displays a table with one row for a user named 'wsv'. The table columns are 'EMAIL ADDRESS' (will@wsvincent.com), 'USERNAME' (wsv), 'AGE' (empty), and 'STAFF STATUS' (Yes). A search bar and a 'Search' button are at the top of the table. To the right of the table is a 'FILTER' sidebar with three sections: 'By staff status' (All, Yes, No), 'By superuser status' (All, Yes, No), and 'By active' (All, Yes, No). A 'ADD USER +' button is located in the top right corner of the main content area.

### Admin select user to change

The age field is empty because we have yet to set it. The default prompt for creating a superuser does not ask for it; however, in the next chapter, we will see it is automatically included in our signup form.

If you wish to set the age now, that is possible because we set the `fieldsets` section. Click on the highlighted link for your superuser's email address, bringing up the edit user interface. If you scroll to the bottom, we added the age field. Go ahead and enter your age. Then click on "Save".

wsv | Change user | Django site

127.0.0.1:8000/admin/accounts/customuser/1/change/

Start typing to filter...

**ACCOUNTS**

Users [+ Add](#)

**AUTHENTICATION AND AUTHORIZATION**

Groups [+ Add](#)

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  
auth | group | Can change group  
auth | group | Can delete group  
auth | group | Can view group  
auth | permission | Can add permission

**Choose all**  Remove all

Specific permissions for this user. Hold down "Control", or "Command" on a Mac, to select more than one.

**Important dates**

Last login: Date: 2023-04-24 Today | [Calendar](#)  
Time: 13:08:38 Now | [Clock](#)

Note: You are 4 hours behind server time.

Date joined: Date: 2023-04-24 Today | [Calendar](#)  
Time: 13:08:15 Now | [Clock](#)

Note: You are 4 hours behind server time.

Age:

**Admin edit age**

**SAVE** **Save and add another** **Save and continue editing** **Delete**

It will automatically redirect back to the main Users page listing our superuser. Note that the age field is now updated.

The screenshot shows the Django admin 'Select user to change' page. The URL is 127.0.0.1:8000/admin/accounts/customuser/. A green success message at the top right says 'The user "wsv" was changed successfully.' The main area shows a table with one user entry: 'will@wsvincent.com' (username wsv, age 42, staff status checked). The sidebar on the left has 'ACCOUNTS' selected, with 'Users' highlighted. The sidebar on the right contains filter options for staff status (All, Yes, No), superuser status (All, Yes, No), and active status (All, Yes, No).

**Admin updated age**

## Tests

Every time we make code changes that alter core functionality, it is a good idea to add tests. While all our manual actions trying out the custom user worked just now, we may break something in the future. Adding tests for new code and regularly running the entire test suite helps spot errors early.

At a high level, we want to ensure that both a regular user and a superuser can be created and have the proper field permissions. Suppose you look at the [official documentation](#) on `models.User`, which our custom user model inherits from. In that case, it comes with a number of built-in fields: `username`, `first_name`, `last_name`, `email`, `password`, `groups`, `user_permissions`, `is_staff`, `is_active`, `is_superuser`, `last_login`, and `date_joined`. It is also possible to add any number of custom fields, as we have seen by adding the `age` field.

Being “staff” means a user can access the admin site and view models for which they are given permission; a “superuser” has full access to the admin and all its models. A regular user should have `is_active` set to True, `is_staff` set to False, and `is_superuser` to False. A superuser should have everything set to True.

Here is one way to add tests to our custom user model:

Code

---

```
# accounts/tests.py
from django.contrib.auth import get_user_model
from django.test import TestCase

class UsersManagersTests(TestCase):
    def test_create_user(self):
        User = get_user_model()
        user = User.objects.create_user(
            username="testuser",
            email="testuser@example.com",
            password="testpass1234",
        )
        self.assertEqual(user.username, "testuser")
        self.assertEqual(user.email, "testuser@example.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="testsuperuser",
            email="testsuperuser@example.com",
            password="testpass1234",
        )
        self.assertEqual(admin_user.username, "testsuperuser")
        self.assertEqual(admin_user.email, "testsuperuser@example.com")
        self.assertTrue(admin_user.is_active)
        self.assertTrue(admin_user.is_staff)
        self.assertTrue(admin_user.is_superuser)
```

---

At the top, we import `get_user_model()`, so we can test our user registration. Then we also import `TestCase` to run tests that touch the database and reverse so we can verify the URL and view work properly.

Our class of tests is called `UsersManagersTests` and extends `TestCase`. The first unit test, `test_create_user`, checks that a regular user displays expected behavior. The second unit test, `test_create_superuser`, does the same, albeit for a superuser account.

Now run the tests; they should pass without any issues.

---

#### Shell

---

```
(.venv) $ python manage.py test
Found 2 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

.
-----
Ran 2 tests in 0.114s

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

---

## Git

We've completed a bunch of new work so it's time to add a Git commit.

---

#### Shell

---

```
(.venv) $ git status
(.venv) $ git add -A
(.venv) $ git commit -m "custom user model"
```

---

## Conclusion

We started our new project by immediately adding a custom user model and then adding a custom `age` field. We also explored how to add tests whenever core Django functionality is changed and can now focus on building the rest of our *Newspaper* website. In the next chapter, we will implement advanced authentication and registration by customizing the signup, login, and logout pages.

# Chapter 11: User Authentication

Now that we have a working custom user model, we can add the functionality every website needs: the ability for users to sign up, log in, and log out. Django provides everything we need for users to log in and out, but we will need to create our own form to allow new users to sign up. We'll also build a basic homepage with links to all three features so users don't have to type in the URLs by hand every time.

## Templates

By default, the Django template loader looks for templates in a nested structure within each app. The structure `accounts/templates/accounts/home.html` would be needed for a `home.html` template within the `accounts` app. But a single project-level templates directory approach is cleaner and scales better, so that's what we'll use.

Let's create a new `templates` directory and, within it, a `registration` directory, as that's where Django will look for templates related to logging in and signing up.

---

### Shell

---

```
(.venv) $ mkdir templates
(.venv) $ mkdir templates/registration
```

---

We need to tell Django about this new directory by updating the configuration for "DIRS" in `django_project/settings.py`.

---

### Code

---

```
# django_project/settings.py
TEMPLATES = [
    {
        ...
        "DIRS": [BASE_DIR / "templates"], # new
```

```
    }  
]
```

---

If you think about what happens when you log in or out of a site, you are immediately redirected to a subsequent page. We need to tell Django where to send users in each case. The `LOGIN_REDIRECT_URL` and `LOGOUT_REDIRECT_URL` settings do that. We'll configure both to redirect to our homepage with the named URL of '`home`'.

Remember that when we create our URL routes, we can add a name to each one. So when we make the homepage URL, we'll call it '`home`'.

Add these two lines at the bottom of the `django_project/settings.py` file.

Code

---

```
# django_project/settings.py  
LOGIN_REDIRECT_URL = "home" # new  
LOGOUT_REDIRECT_URL = "home" # new
```

---

Now we can create four new templates within our text editor:

- `templates/base.html`
- `templates/home.html`
- `templates/registration/login.html`
- `templates/registration/signup.html`

Here's the HTML code for each file to use. The `base.html` will be inherited by every other template in our project. Using a block like `{% block content %}`, we can later override the content *just in this place* in other templates.

Code

---

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

```
<html>

<head>
    <meta charset="utf-8">
    <title>{% block title %}Newspaper App{% endblock title %}</title>
</head>

<body>
    <main>
        {% block content %}
        {% endblock content %}
    </main>
</body>

</html>
```

---

## Code

```
<!-- templates/home.html -->
{% extends "base.html" %}

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

{% block content %}
{% if user.is_authenticated %}


Hi {{ user.username }}!



Log Out


{% else %}


You are not logged in



Log In |



Sign Up


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

---

## Code

```
<!-- templates/registration/login.html -->
{% extends "base.html" %}

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

{% block content %}


## Log In

{% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Log In</button>

{% endblock content %}
```

---

## Code

```
<!-- templates/registration/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 %}
```

---

Our templates are now all set. Still to go are the related URLs and views.

## URLs

Let's start with the URL routes. In our `django_project/urls.py` file, we want our `home.html` template to appear as the homepage, but we don't want to build a dedicated pages app yet. We can use the shortcut of importing `TemplateView` and setting the `template_name` right in our URL pattern.

Next, we want to "include" the `accounts` app and the built-in auth app. The reason is that the built-in auth app already provides views and URLs for logging in and logging out. But to sign up, we must create our own view and URL. To ensure that our URL routes are consistent, we place them *both* at `accounts/` so the eventual URLs will be `/accounts/login`, `/accounts/logout`, and `/accounts/signup`.

---

### Code

---

```
# django_project/urls.py  
from django.contrib import admin  
from django.urls import path, include # new  
from django.views.generic.base import TemplateView # new  
  
urlpatterns = [  
    path("admin/", admin.site.urls),  
    path("accounts/", include("accounts.urls")), # new  
    path("accounts/", include("django.contrib.auth.urls")), # new  
    path("", TemplateView.as_view(template_name="home.html"),  
          name="home"), # new  
]
```

---

Now create a file with your text editor called accounts/urls.py and update it with the following code:

Code

---

```
# accounts/urls.py
from django.urls import path
from .views import SignUpView

urlpatterns = [
    path("signup/", SignUpView.as_view(), name="signup"),
]
```

---

The last step is our views.py file containing the logic for our signup form. We're using Django's generic CreateView here and telling it to use our CustomUserCreationForm, to redirect to login once a user signs up successfully and that our template is named signup.html.

Code

---

```
# accounts/views.py
from django.urls import reverse_lazy
from django.views.generic import CreateView

from .forms import CustomUserCreationForm

class SignUpView(CreateView):
    form_class = CustomUserCreationForm
    success_url = reverse_lazy('login')
    template_name = "registration/signup.html"
```

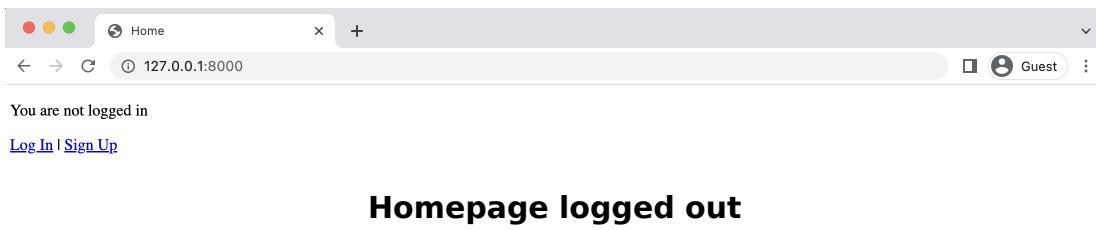
---

Ok, phew! We're done. Let's test things out. Start the server with python manage.py runserver and go to the homepage.



We logged in to the admin in the previous chapter, so you should see a personalized greeting here. Click on the “Log

Out” link.



Now we’re on the logged-out homepage. Click on *Log In* link and use your **superuser** credentials. Upon successfully logging in, you’ll be redirected to the homepage and see the same personalized greeting as before. It works!

Now use the “Log Out” link to return to the logged-out homepage, and this time, click on the “Sign Up” link. You’ll be redirected to our signup page. See that the age field is included!

Create a new user. Mine is called `testuser`, and I’ve set the age to 25.

A screenshot of a web browser window showing the "Sign Up" form at the URL "127.0.0.1:8000/accounts/signup/". The form fields include "Username" (set to "testuser"), "Age" (set to "25"), and "Password" (set to "\*\*\*\*\*"). Below the password field is a list of password requirements: "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.". A note below the password field says "Enter the same password as before, for verification." A "Sign Up" button is at the bottom.

### Signup page

After successfully submitting the form, you’ll be redirected to the login page. Log in with your new user, and you’ll again be redirected to the homepage with a personalized

greeting for the new user. But since we have the new age field, let's add that to the `home.html` template. It is a field on the `user` model, so to display it, we only need to use `{{ user.age }}`.

#### Code

---

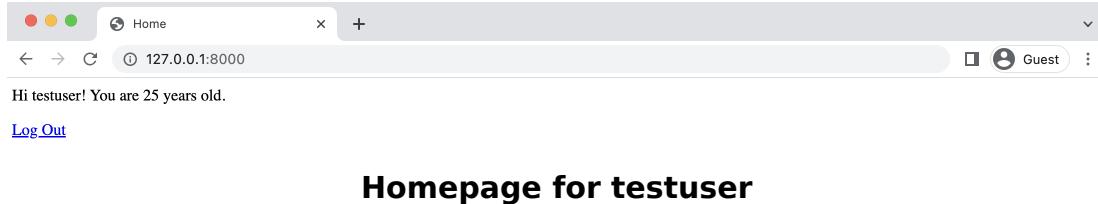
```
<!-- templates/home.html -->
{% extends "base.html" %}

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

{% block content %}
{% if user.is_authenticated %}
<!-- new code here! -->
Hi {{ user.username }}! You are {{ user.age }} years old.
<!-- end of new code -->
<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 %}
```

---

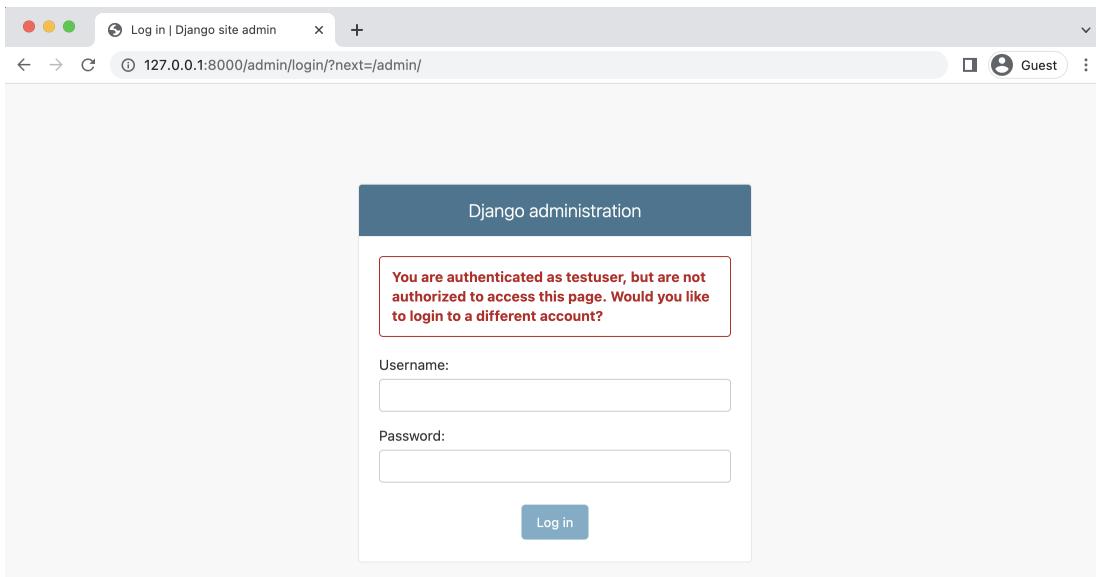
Save the file and refresh the homepage.



Everything works as expected.

## Admin

Navigate to the admin at `http://127.0.0.1:8000/admin` in your web browser and log in to view the two user accounts.



### Wrong Admin Login

What's this? Why can't we log in? We're logged in with our new testuser account, not our superuser account. Only a superuser account has permission to log in to the admin! So use your superuser account to log in instead.

After you've done that, you should see the normal admin homepage. Click on `Users` to see our two users: the testuser account we just created and your previous superuser name (mine is `wsv`).

## Users in the Admin

Everything is working, but you may notice no “email address” for our testuser. Why is that? Our signup page has no email field because it was not included in accounts/forms.py. This is an important point: just because the user model has a field does not mean it will be included in our custom signup form unless explicitly added. Let’s do so now.

Currently, in accounts/forms.py under fields, we’re using Meta.fields, which displays the default settings of username/password, and then we explicitly added our custom field, age, too. But we can also explicitly set which fields we want to be displayed, so let’s update it to ask for a username/email/age/password by setting it to ('username', 'email', 'age',). We don’t need to include the password fields because they are required! The other fields can be configured however we choose.

### Code

---

```
# accounts/forms.py
from django.contrib.auth.forms import UserCreationForm, UserChangeForm
```

```

from .models import CustomUser

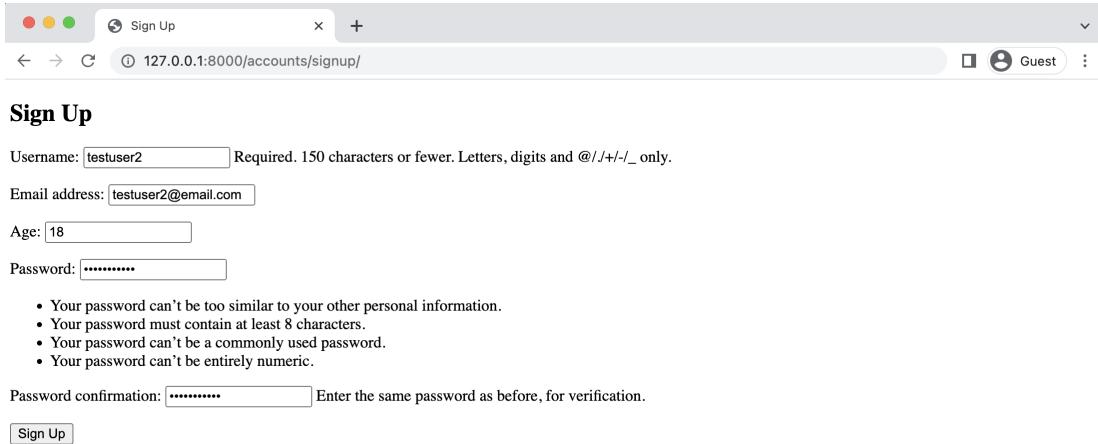
class CustomUserCreationForm(UserCreationForm):
    class Meta:
        model = CustomUser
        fields = (
            "username",
            "email",
            "age",
        ) # new

class CustomUserChangeForm(UserChangeForm):
    class Meta:
        model = CustomUser
        fields = (
            "username",
            "email",
            "age",
        ) # new

```

---

Now if you try `http://127.0.0.1:8000/accounts/signup/` again, you can see the additional “Email address” field is there. Sign up with a new user account. I’ve named mine testuser2 with an age of 18 and an email address of `testuser2@email.com`.



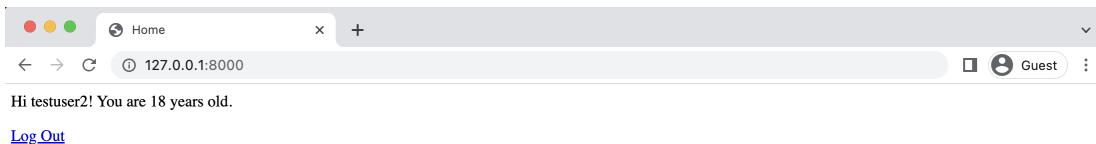
The screenshot shows a web browser window with a "Sign Up" page. The URL in the address bar is `127.0.0.1:8000/accounts/signup/`. The page has a header with three colored dots (red, yellow, green) and a "Sign Up" button. Below the header, there are input fields for "Username" (containing "testuser2"), "Email address" (containing "testuser2@email.com"), "Age" (containing "18"), and "Password". The password field contains several asterisks. Below the password field is a list of password requirements:

- 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.

At the bottom of the form, there is a "Password confirmation" field containing asterisks and a note: "Enter the same password as before, for verification." A "Sign Up" button is located at the very bottom of the form.

### New signup page

Click the “Sign Up” button and continue to log in. You’ll see a personalized greeting on the homepage.



Then switch back to the admin page-log in using our superuser account to do so-and all three users are on display. Note that I've closed the side navbar of the admin in this screenshot by clicking on the arrows <> to make it more readable.

A screenshot of the Django Admin interface. The title bar says 'Select user to change' and 'Django administration'. The main area shows a table with three users: 'testuser' (age 25, staff status red), 'testuser2' (age 18, staff status red), and 'wsv' (age 42, staff status green). A sidebar on the right is titled 'FILTER' and includes sections for 'By staff status' (All, Yes, No), 'By superuser status' (All, Yes, No), and 'By active' (All, Yes, No).

### Three users in the Admin

Django's user authentication flow requires a bit of setup. Still, you should be starting to see that it also provides us incredible flexibility to configure the signup and login process *exactly* how we want.

## Tests

The new signup page has a view, URL, and template that all should be tested. Open up the accounts/tests.py file containing code from the last chapter for UsersManagersTests. Below it, add a new class called SignupPageTests that we will review below.

#### Code

---

```
# accounts/tests.py
from django.contrib.auth import get_user_model
from django.test import TestCase
from django.urls import reverse # new

class UsersManagersTests(TestCase):
    ...

class SignupPageTests(TestCase): # new
    def test_url_exists_at_correct_location_signupview(self):
        response = self.client.get("/accounts/signup/")
        self.assertEqual(response.status_code, 200)

    def test_signup_view_name(self):
        response = self.client.get(reverse("signup"))
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, "registration/signup.html")

    def test_signup_form(self):
        response = self.client.post(
            reverse("signup"),
            {
                "username": "testuser",
                "email": "testuser@email.com",
                "password1": "testpass123",
                "password2": "testpass123",
            },
        )
        self.assertEqual(response.status_code, 302)
        self.assertEqual(get_user_model().objects.all().count(), 1)
        self.assertEqual(get_user_model().objects.all()[0].username,
"testuser")
        self.assertEqual(
            get_user_model().objects.all()[0].email, "testuser@email.com"
    )
```

---

At the top, we import [reverse](#) to verify that the URL and view work properly. Then we create a new class of tests called SignupPageTests. The first test checks that our signup page is at the correct URL and returns a 200 status code. The second test checks the view, reverses `signup`, which is

the URL name, and then confirms a 200 status code and that our `signup.html` template is being used.

The third test checks our form by sending a post request to fill it out. After the form is submitted, we confirm the expected 302 redirect and confirm that there is now one user in the test database with a matching username and email address. We do not check the password because Django automatically encrypts them by default. That is why if you look in a user's admin view, you can change a password but can't see the current one.

Run the tests with `python manage.py test` to check that everything passes as expected.

---

#### Shell

---

```
(.venv) $ python manage.py test
Found 5 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

...
-----
Ran 5 tests in 0.183s

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

---

## Git

Before moving on to the next chapter, let's record our work with Git and store it on GitHub.

---

#### Shell

---

```
(.venv) $ git status
(.venv) $ git add -A
(.venv) $ git commit -m "user authentication"
(.venv) $ git push origin main
```

---

## Conclusion

So far, our *Newspaper* app has a custom user model and working signup, login, and logout pages. But you may have noticed that our site could look better. In the next chapter, we'll add CSS styling with Bootstrap and create a dedicated pages app.

# Chapter 12: Bootstrap

Web development requires a lot of skills. Not only do you have to program the website correctly, but users expect it to look good, too. When creating everything from scratch, adding all the necessary HTML/CSS for a beautiful site can be overwhelming.

While it's possible to hand-code all the required CSS and JavaScript for a modern-looking website, in practice, most developers use a framework like [Bootstrap](#) or [TailwindCSS](#). For our project, we'll use Bootstrap, which can be extended and customized as needed.

## Pages App

In the previous chapter, we displayed our homepage by including view logic in our `urls.py` file. While this approach works, it feels hackish to me, and it certainly doesn't scale as a website grows over time; it is also confusing to Django newcomers. Instead, we can and should create a dedicated pages app for all of our static pages, such as the homepage, a future about page, etc. This will keep our code nice and organized going forward.

On the command line, use the `startapp` command to create our new pages app. If the server is still running, you may need to type `Control+c` first to quit.

Shell

---

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

---

Then immediately update our `django_project/settings.py` file. I often forget to do this, so it is a good practice to think of

creating a new app as a two-step process: run the `startapp` command, then update `INSTALLED_APPS`.

Code

---

```
# django_project/settings.py
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "accounts",
    "pages", # new
]
```

---

Now we can update our `urls.py` file inside the `django_project` directory by adding the `pages` app and removing the import of `TemplateView` and the previous URL path for the older homepage.

Code

---

```
# django_project/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path("admin/", admin.site.urls),
    path("accounts/", include("accounts.urls")),
    path("accounts/", include("django.contrib.auth.urls")),
    path("", include("pages.urls")), # new
]
```

---

It's time to add our homepage, which means Django's standard URLs/views/templates dance. We'll start with the `pages/urls.py` file. First, create it with your text editor. Then import our not-yet-created views, set the route paths, and name each URL, too.

Code

---

```
# pages/urls.py
from django.urls import path

from .views import HomePageView

urlpatterns = [
```

```
        path("", HomePageView.as_view(), name="home"),
]
```

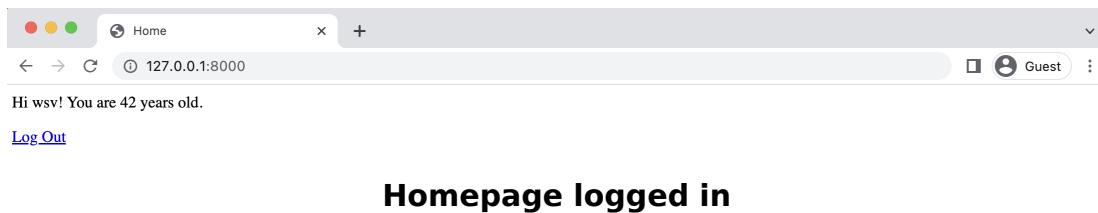
The `views.py` code should look familiar at this point. We're using Django's `TemplateView` generic class-based view, meaning we only need to specify our `template_name` to use it.

#### Code

```
# pages/views.py
from django.views.generic import TemplateView

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

We already have an existing `home.html` template. Let's confirm it still works as expected with our new URL and view. Start the local server `python manage.py runserver` and navigate to the homepage at `http://127.0.0.1:8000/` to confirm it remains unchanged.



It should show the name and age of your logged-in superuser account, which we used at the end of the last chapter.

## Tests

We've added new code and functionality, so it is time for tests. You can never have enough tests in your projects. Even though they take some upfront time to write, they always save you time down the road and provide you with confidence as a project grows in complexity.

Let's add tests to ensure our new homepage works properly. Here's what the code should look like in your pages/tests.py file.

---

Code

---

```
# pages/tests.py
from django.test import SimpleTestCase
from django.urls import reverse

class HomePageTests(SimpleTestCase):
    def test_url_exists_at_correct_location_homepageview(self):
        response = self.client.get("/")
        self.assertEqual(response.status_code, 200)

    def test_homepage_view(self):
        response = self.client.get(reverse("home"))
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, "home.html")
        self.assertContains(response, "Home")
```

---

On the top line, we import SimpleTestCase since our homepage does not rely on the database. If it did, we'd have to use TestCase instead. Then we import reverse to test our URL and view.

Our test class, HomePageTests, has two tests that check the homepage URL returns a 200 status code and that it uses our expected URL name, template, and contains “Home” in the response.

Quit the local server with `Control+c` and then run our tests to confirm everything passes.

---

Shell

---

```
(.venv) $ python manage.py test
Found 7 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
.
.
Ran 7 tests in 0.185s

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

---

## Testing Philosophy

There's really no limit to what you can test in an application. For example, we *could* also add tests now based on logged-in or logged-out behavior and whether the template displays the proper content. But the 80/20 rule of 80% of consequences coming from 20% of causes applies to testing and most other things in life. There's no sense in making as many unit tests as possible to test things that will likely never fail, at least for a web application. If we were working on a nuclear reactor, having as many tests as possible would make sense, but the stakes are a bit lower for most websites.

So while you always want to add tests around new features, it's ok to only have partial test coverage from the beginning. As errors inevitably arise on new Git branches and features, add a test for each so they don't fail again. This approach is known as *regression testing*, where tests are re-run each time there's a new change to ensure that previously developed and tested software performs as expected.

Django's testing suite is well set up for many unit tests and automatic regression tests, so developers have confidence in the consistency of their projects.

## Bootstrap

Moving along, it's time to add some style to our application. If you've never used Bootstrap, you're in for a real treat. Much like Django, it accomplishes so much in so little code.

There are two ways to add Bootstrap to a project: download and serve all the files locally or rely on a Content Delivery Network (CDN). The second approach is simpler to

implement, provided you have a consistent internet connection, so that's what we'll use here.

Our template will mimic the “Starter template” provided on the [Bootstrap introduction](#) page and involves adding the following:

- `meta name="viewport"` and content information at the top within `<head>`
- Bootstrap CSS link within `<head>`
- Bootstrap JavaScript bundle at the bottom of the `<body>` section

In general, typing out all code yourself is recommended, but adding the Bootstrap CDN is an exception since it is lengthy and easy to mistype. I recommend copying and pasting the Bootstrap CSS and JavaScript Bundle links from the Bootstrap website into the `base.html` file.

---

#### Code

---

```
<!-- templates/base.html -->
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>{% block title %}Newspaper App{% endblock title %}</title>
<meta name="viewport" content="width=device-width,
initial-scale=1, shrink-to-fit=no">

<!-- Bootstrap CSS -->
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/
bootstrap.min.css" rel="stylesheet" integrity="sha384-...
 crossorigin="anonymous">
</head>
<body>
<main>
  {% block content %}
  {% endblock content %}
</main>

<!-- Bootstrap JavaScript Bundle -->
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/js/
bootstrap.bundle.min.js" integrity="sha384-...
 crossorigin="anonymous">
</script>
</body>
</html>
```

---

This code snippet **does not** include the full links for Bootstrap CSS and JavaScript; it is abbreviated. Copy and paste the full links for Bootstrap 5.2 from the [quick start docs](#).

If you start the server again with `python manage.py runserver` and refresh the homepage at `http://127.0.0.1:8000/`, you'll see that the font size and link colors have changed.

Let's add a navigation bar at the top of the page containing our links for the homepage, login, logout, and signup pages. Notably, we can use the [if/else](#) tags in the Django templating engine to add some basic logic. We want the "log in" and "sign up" buttons to appear for a logged-out user; the "log out" and "change password" buttons when a user is logged in.

Again, it's ok to copy/paste here since this book focuses on learning Django, not HTML, CSS, and Bootstrap. You can view the [official GitHub repository](#) for reference if there are any formatting issues.

#### Code

---

```
<!-- templates/base.html -->
...
<body>
  <header class="p-3 mb-3 border-bottom">
    <div class="d-flex flex-wrap align-items-center justify-content-center
      justify-content-lg-start">
      <a class="navbar-brand" href="{% url 'home' %}">Newspaper</a>
      <ul class="nav col-12 col-lg-auto me-lg-auto mb-2 justify-content-center
        mb-md-0">
        {% if user.is_authenticated %}
          <li><a href="#" class="nav-link px-2 link-dark">+ New</a></li>
        </ul>
        <div class="dropdown text-end">
          <a href="#" class="d-block link-dark text-decoration-none dropdown-
            toggle"
            id="dropdownUser1" data-bs-toggle="dropdown" aria-expanded="false">
            {{ user.username }}
          </a>
          <ul class="dropdown-menu text-small" aria-labelledby="dropdownUser1">
            <li><a class="dropdown-item" href="{% url 'password_change' %}">
              Change password</a></li>
```

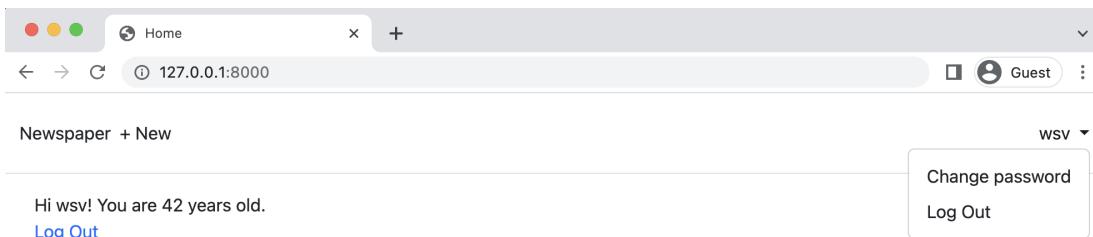
```

</li>
</ul>
</div>
{% else %}
</ul>
<div class="text-end">
    <a href="{% url 'login' %}" class="btn btn-outline-primary me-2">
        Log In</a>
    <a href="{% url 'signup' %}" class="btn btn-primary">Sign Up</a>
</div>
{% endif %}
</div>
</div>
</header>
<main>
    <div class="container">
        {% block content %}
        {% endblock content %}
    </div>
</main>
...

```

---

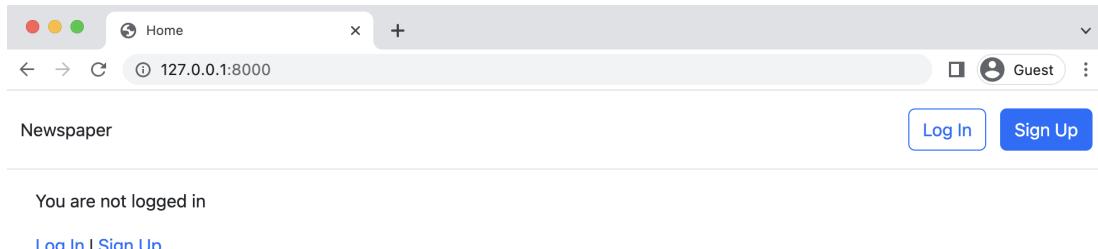
If you refresh the homepage at <http://127.0.0.1:8000/>, our new navbar has magically appeared! Note that there is no actual yet for new articles “+ New”; there is just a placeholder represented in the code by href="#". We will add that later on. Also, note that our logged-in username is now in the upper right corner, along with a dropdown arrow. If you click on it, there are links for “Change password” and “Log Out.”



**Homepage with Bootstrap nav logged in**

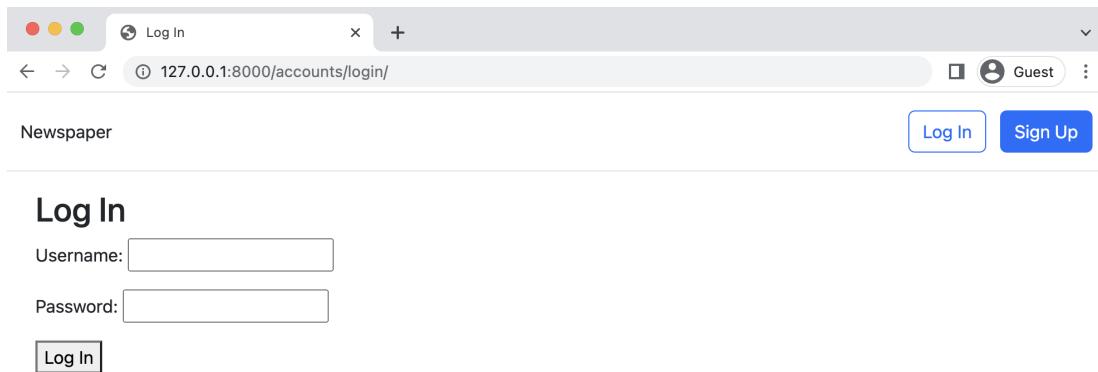
If you click “Log Out” in the dropdown, the navbar changes to button links for either “Log In” or “Sign Up” and the “+

New” link disappears. There is no sense in letting logged out users create articles.



### Homepage with Bootstrap nav logged out

If you click on the “Log In” button in the top nav, you can also see that our login page at <http://127.0.0.1:8000/accounts/login> looks better too.



### Bootstrap login

The only thing that looks off is our gray “Log In” button. We can use Bootstrap to add some nice styling, such as making it green and inviting. Change the “button” line as follows in the `templates/registration/login.html` file.

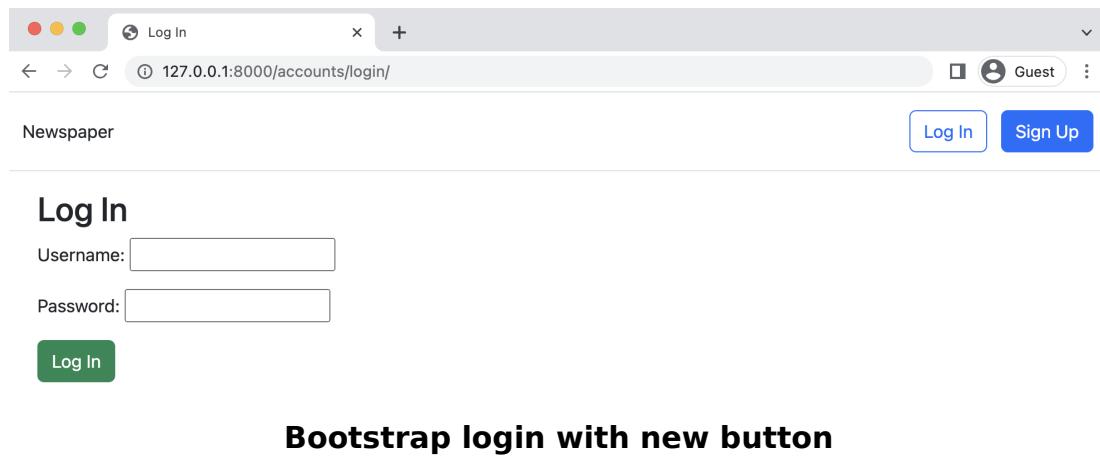
#### Code

```
<!-- 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 }}  
    <!-- new code here -->  
    <button class="btn btn-success ml-2" type="submit">Log In</button>  
    <!-- end new code -->  
</form>  
{% endblock content %}
```

Now refresh the page to see our new button in action.



## Signup Form

If you click on the link for “Sign Up” you’ll see that the page has Bootstrap stylings and distracting helper text. For example, after “Username” it says “Required. 150 characters or fewer. Letters, digits, and @./+/-/\_ only.”

The screenshot shows a web browser window with a 'Sign Up' page. The URL in the address bar is 127.0.0.1:8000/accounts/signup/. The page has a header with 'Sign Up' and 'Guest'. Below the header, there's a 'Newspaper' link and 'Log In' and 'Sign Up' buttons. The main content is a 'Sign Up' form with fields for Username, Email address, Age, and Password. It includes validation messages and a password confirmation field.

Sign Up

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

Email address:

Age:

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

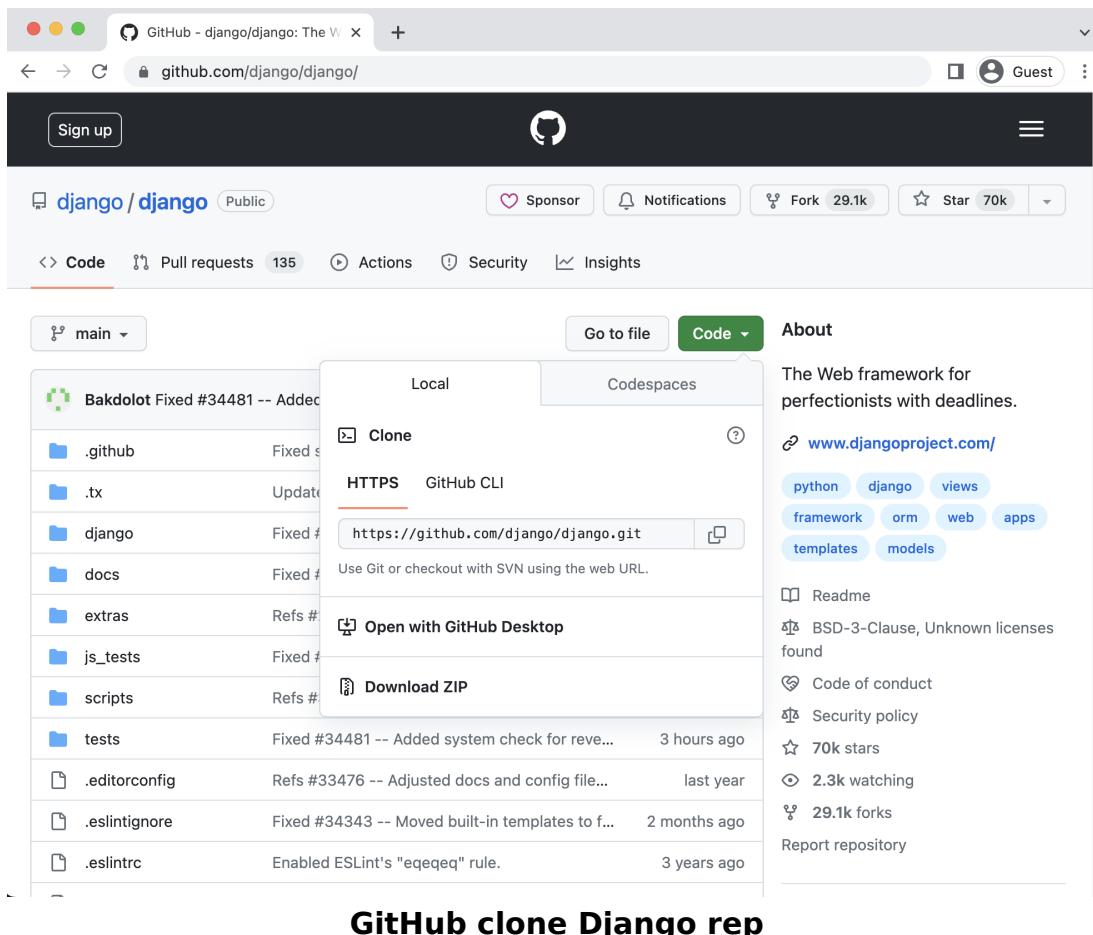
So where did that text come from? Whenever something feels like “magic” in Django, rest assured that it is decidedly not. If you did not write the code yourself, it exists somewhere within Django.

The best method to figure out what’s happening under the hood in Django is to download the source code and take a look yourself. All the code development occurs on GitHub, and you can find the Django repo at <https://github.com/django/django/>. To copy it onto your local computer, open a new tab (Command + t) on your command line and navigate to the desired location. Since we’ve been using a code folder on the desktop, let’s navigate there now.

### Shell

```
# Windows  
$ cd onedrive\desktop\code  
  
# macOS  
$ cd ~/desktop/code
```

If you click the green “Code” button on the GitHub repository page, you’ll see a dropdown list with multiple options. Select “Clone” (Git’s word for “copy”) and then the “SSH” method.



On the command line, type `git clone git@github.com:django/django.git` to copy the Django source code onto your computer in a new directory called `django`.

### Shell

```
$ git clone git@github.com:django/django.git
```

Having the Django source code on your computer will require manual updates now and then to stay current. After

all, Django undergoes regular updates to fix bugs, improve security, and add new features. You might ask, “Why not just use the built-in GitHub search?” Searching on GitHub *sometimes* works, but lately, it has been inconsistent, something GitHub is well aware of and working to improve. Downloading the source code and searching yourself is a valuable tool, so it is worth taking the time to learn how to do so, as we are here.

In your text editor, open the Django source code to perform searches. For example, in the VS Code text editor, press the keys command + shift + f to do a “find” search in all files. Type in a search for “150 characters or fewer” and you’ll find the top link is to the page for `django/contrib/auth/models.py`. The specific code is on line 349 and the text is part of the `auth` app, on the `username` field for `AbstractUser`.

**Note:** We now have two command line tabs open: one for our project code and one for the Django source code. Make sure you understand which is which. Going forward, we *will* do additional searches on the source code, but all code in this book requires switching *back* to the terminal tab with the project source code. Switch back as we are about to add more code to our project.

We have three options now:

- override the existing `help_text`
- hide the `help_text`
- restyle the `help_text`

We’ll choose the third option since it’s a good way to introduce the excellent 3rd party package [django-crispy-forms](#).

Working with forms is challenging, and `django-crispy-forms` makes writing DRY (Don’t-Repeat-Yourself) code easier. First,

stop the local server with `Control+c`. Then use `pip` to install the package in our project. We'll also install the [Bootstrap5 template pack](#).

#### Shell

---

```
(.venv) $ python -m pip install django-crispy-forms==2.0
(.venv) $ python -m pip install crispy-bootstrap5==0.7
```

---

Add the new apps to our `INSTALLED_APPS` list in the `django_project/settings.py` file. As the number of apps starts to grow, I find it helpful to distinguish between “3rd party” apps and “local” apps I've added myself. Here's what the code looks like now.

#### Code

---

```
# django_project/settings.py
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    # 3rd Party
    "crispy_forms",  # new
    "crispy_bootstrap5",  # new
    # Local
    "accounts",
    "pages",
]
```

---

And then, at the bottom of the `settings.py` file, add two new lines as well.

#### Code

---

```
# django_project/settings.py
CRISPY_ALLOWED_TEMPLATE_PACKS = "bootstrap5"  # new
CRISPY_TEMPLATE_PACK = "bootstrap5"  # new
```

---

Now in our `signup.html` template, we can quickly use crispy forms. First, we load `crispy_forms_tags` at the top and then swap out `{{ form.as_p }}` for `{{ form|crispy }}`. We'll also

update the “Sign Up” button to be green with the `btn-success` styling.

---

Code

---

```
<!-- templates/registration/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 %}
```

---

We can see the new changes if you start the server again with `python manage.py runserver` and refresh the signup page.

Sign Up

Username\*

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

Email address

Age

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**

### Crispy signup page

We can also add crispy forms to our login page. The process is the same. Here is that updated code:

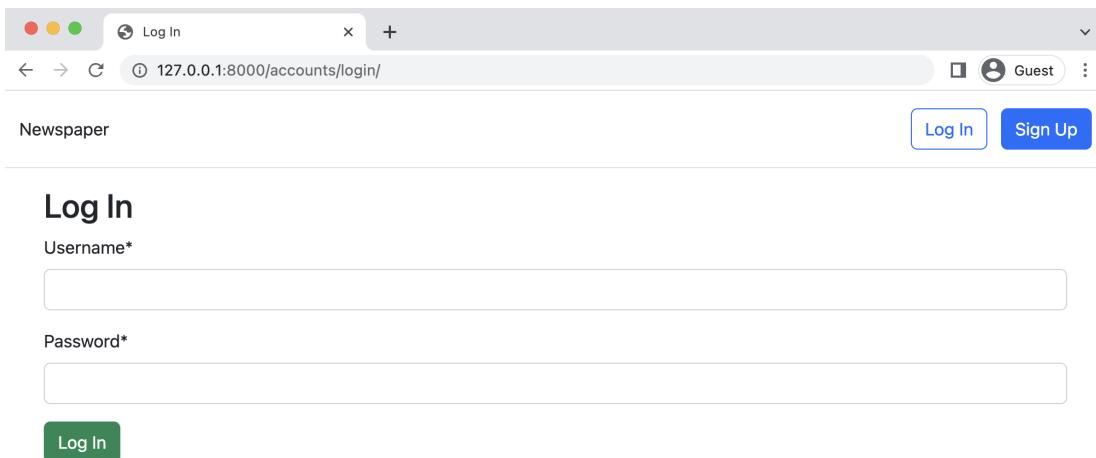
#### Code

```
<!-- 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 ml-2" type="submit">Log In</button>
</form>
{% endblock content %}
```

Refresh the login page, and the update will be visible.



A screenshot of a web browser window showing a login form. The title bar says "Log In". The address bar shows "127.0.0.1:8000/accounts/login/". The page header includes "Newspaper", "Guest", and "Log In / Sign Up" buttons. The main content is a "Log In" form with fields for "Username\*" and "Password\*", both with red asterisks indicating required fields. A green "Log In" button is at the bottom. The entire form is styled with Bootstrap classes from the Crispy Forms library.

**Crispy login page**

## Git

A quick Git commit is in order to save our work in this chapter and store it on GitHub.

### Shell

```
(.venv) $ git status  
(.venv) $ git add -A  
(.venv) $ git commit -m "add Bootstrap styling"  
(.venv) $ git push origin main
```

## Conclusion

Our *Newspaper* app looks pretty good. We added Bootstrap to our site and Django Crispy Forms to improve the look of our forms. The last step of our user auth flow is configuring password change and reset. Here again, Django has taken care of the heavy lifting, which requires a minimal amount of code on our part.

# **Chapter 13: Password Change and Reset**

In this chapter, we will complete the authorization flow of our *Newspaper* app by adding password change and password reset functionality. Initially, we'll implement Django's built-in views and URLs for password change and password reset before customizing them with our own Bootstrap-powered templates and email service.

## **Password Change**

Letting users change their passwords is a standard feature on many websites, and Django provides a default implementation that already works at this stage. To try it out, click the “Log In” button to ensure you’re logged in. Then navigate to the “Password change” page at [http://127.0.0.1:8000/accounts/password\\_change/](http://127.0.0.1:8000/accounts/password_change/).

A screenshot of a web browser showing the 'Password change' page from a 'Django site'. The URL in the address bar is 127.0.0.1:8000/accounts/password\_change/. The page has a dark blue header with 'Django administration' and a light blue footer with 'Home > Password change'. The main content area is titled 'Password change' and contains instructions: 'Please enter your old password, for security's sake, and then enter your new password twice so we can verify you typed it in correctly.' Below this are four input fields: 'Old password', 'New password', 'New password confirmation', and a 'CHANGE MY PASSWORD' button. A note below the 'New password' field states: '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.'

Enter both your old password and a new one. Then click the “Change My Password” button, and you will be redirected to the “Password change successful” page.

A screenshot of a web browser showing the 'Password change successful' page from a 'Django site'. The URL in the address bar is 127.0.0.1:8000/accounts/password\_change/done/. The page has a dark blue header with 'Django administration' and a light blue footer with 'Home > Password change'. The main content area is titled 'Password change successful' and displays the message: 'Your password was changed.'

## Password change done

## Customizing Password Change

Let's customize these two password change pages to match the look and feel of our *Newspaper* site. Because Django

already has created the views and URLs for us, we only need to change the templates; however, we must use the names `password_change_form.html` and `password_change_done.html`. In your text editor, create two new template files in the `registration` directory:

- `templates/registration/password_change_form.html`
- `templates/registration/password_change_done.html`

Update `password_change_form.html` with the following code. At the top, we extend `base.html`, load crispy forms, and set our page meta title, which appears in the tab of a web browser but not on the visible webpage itself. The form uses POST since we send data, a `csrf_token` for security reasons, and `{{ form|crispy }}` to use crispy forms styling. As a final tweak, we include a submit button that uses Bootstrap's `btn btn-success` styling to make it green.

---

#### Code

---

```
<!-- templates/registration/password_change_form.html -->
{% extends "base.html" %}
{% load crispy_forms_tags %}

{% block title %}Password Change{% endblock title %}

{% block content %}
<h1>Password change</h1>
<p>Please enter your old password, for security's sake, and then enter
your new password twice so we can verify you typed it in correctly.</p>

<form method="POST">{{ csrf_token }}
{{ form|crispy }}
<input class="btn btn-success" type="submit"
      value="Change my password">
</form>
{% endblock content %}
```

---

Load the page at  
`http://127.0.0.1:8000/accounts/password_change/` to see our changes.

Newspaper + New wsv ▾

## Password change

Please enter your old password, for security's sake, and then enter your new password twice so we can verify you typed it in correctly.

Old password\*

New 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.

New password confirmation\*

**Change my password**

### New password change form

Next up is the `password_change_done` template. It also extends `base.html` and includes a new meta title; however, there's no form on the page, just new text.

#### Code

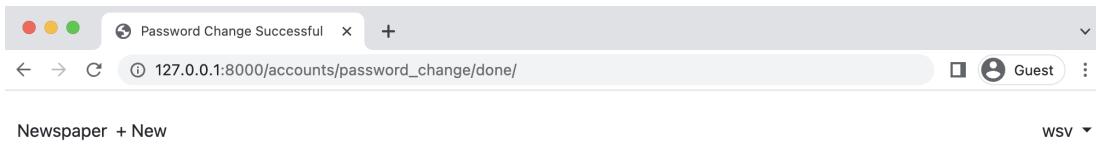
```
<!-- templates/registration/password_change_done.html -->
{% extends "base.html" %}

{% block title %}Password Change Successful{% endblock title %}

{% block content %}
<h1>Password change successful</h1>
<p>Your password was changed.</p>
{% endblock content %}
```

This updated page is at:

[http://127.0.0.1:8000/accounts/password\\_change/done/](http://127.0.0.1:8000/accounts/password_change/done/)



That wasn't too bad, right? Certainly, it was much less work than creating everything from scratch, especially all the code around securely updating a user's password. Next up is the password reset functionality.

## Password Reset

Password reset handles the common case of users forgetting their passwords. The steps are very similar to configuring password change, as we just did. Django already provides a default implementation that we will use and then customize the templates to match the look and feel of the rest of our site.

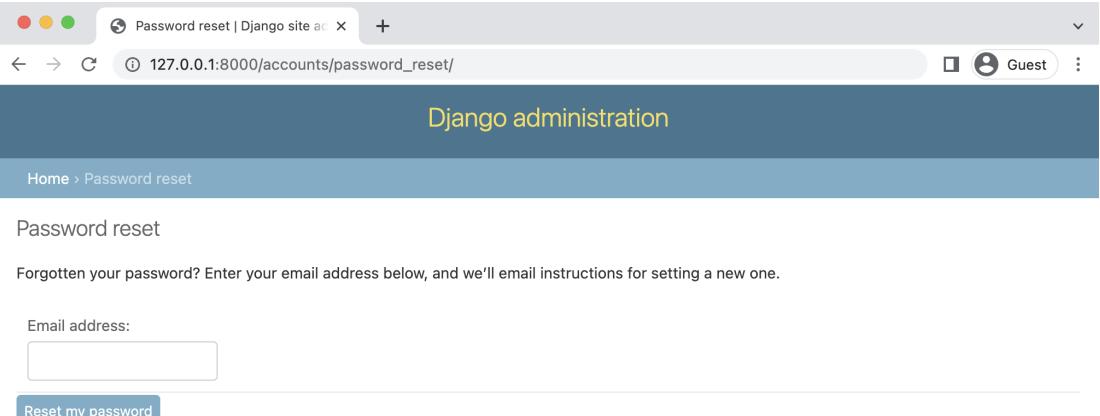
The only configuration required is telling Django *how* to send emails. After all, a user can only reset a password if they can access the email linked to the account. For testing purposes, we can rely on Django's [console backend](#) setting, which outputs the email text to our command line console instead.

Add the following one-line change at the bottom of the `django_project/settings.py` file.

Code

```
# django_project/settings.py
EMAIL_BACKEND = "django.core.mail.backends.console.EmailBackend" # new
```

And we're all set! Django will take care of all the rest for us. Let's try it out. Navigate to [http://127.0.0.1:8000/accounts/password\\_reset/](http://127.0.0.1:8000/accounts/password_reset/) to view the default password reset page.

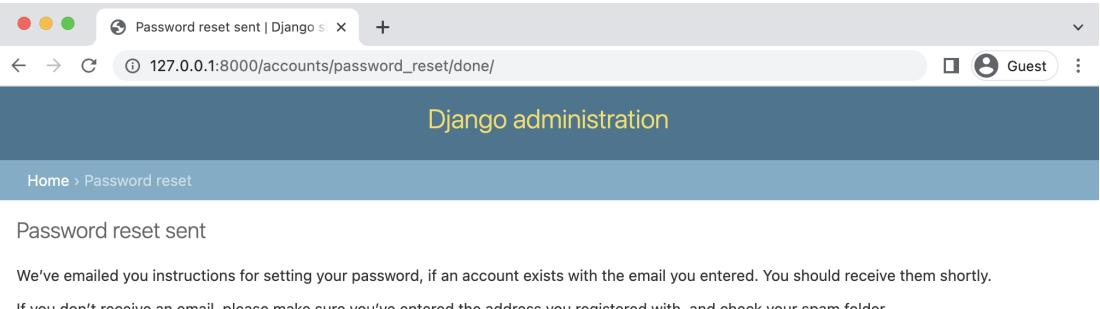


A screenshot of a web browser showing the Django password reset page. The title bar says "Password reset | Django site admin". The address bar shows "127.0.0.1:8000/accounts/password\_reset/". The page header is "Django administration". Below it is a breadcrumb trail "Home > Password reset". The main content area has a heading "Password reset" and a sub-instruction "Forgotten your password? Enter your email address below, and we'll email instructions for setting a new one.". There is a form field labeled "Email address:" with a placeholder "Email address" and a "Reset my password" button.

### Default password reset page

Make sure the email address you enter matches one of your existing user accounts. Recall that `testuser` does not have a linked email account, so you should use `testuser2`, which, if you follow my example, has an email address of `testuser2@email.com`. Upon submission, you'll then be redirected to the password reset done page at:

[http://127.0.0.1:8000/accounts/password\\_reset/done/](http://127.0.0.1:8000/accounts/password_reset/done/)



A screenshot of a web browser showing the Django password reset done page. The title bar says "Password reset sent | Django site admin". The address bar shows "127.0.0.1:8000/accounts/password\_reset/done/". The page header is "Django administration". Below it is a breadcrumb trail "Home > Password reset". The main content area has a heading "Password reset sent" and two paragraphs of text: "We've emailed you instructions for setting your password, if an account exists with the email you entered. You should receive them shortly." and "If you don't receive an email, please make sure you've entered the address you registered with, and check your spam folder."

### Default password reset done page

This page says to check our email. Since we've told Django to send emails to the command line console, the email text will now be there. This is what I see in my console.

### Shell

---

```
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 8bit
Subject: Password reset on 127.0.0.1:8000
From: webmaster@localhost
To: testuser2@email.com
Date: Mon, 24 Apr 2023 15:36:30 -0000
Message-ID:
<168235059067.94136.7639058137157368290@1.0.0.0.0.0.ip6.arpa>
```

You're receiving this email because you requested a password reset for your user account at 127.0.0.1:8000.

Please go to the following page and choose a new password:

<http://127.0.0.1:8000/accounts/reset/Mw/bn63cu-b77b6590a2e38a75c4b2af244c40297e/>

Your username, in case you've forgotten: testuser2

Thanks for using our site!

The 127.0.0.1:8000 team

---

```
[24/Apr/2023 15:36:30] "POST /accounts/password_reset/ HTTP/1.1" 302 0
[24/Apr/2023 15:36:30] "GET /accounts/password_reset/done/ HTTP/1.1" 200 3014
```

---

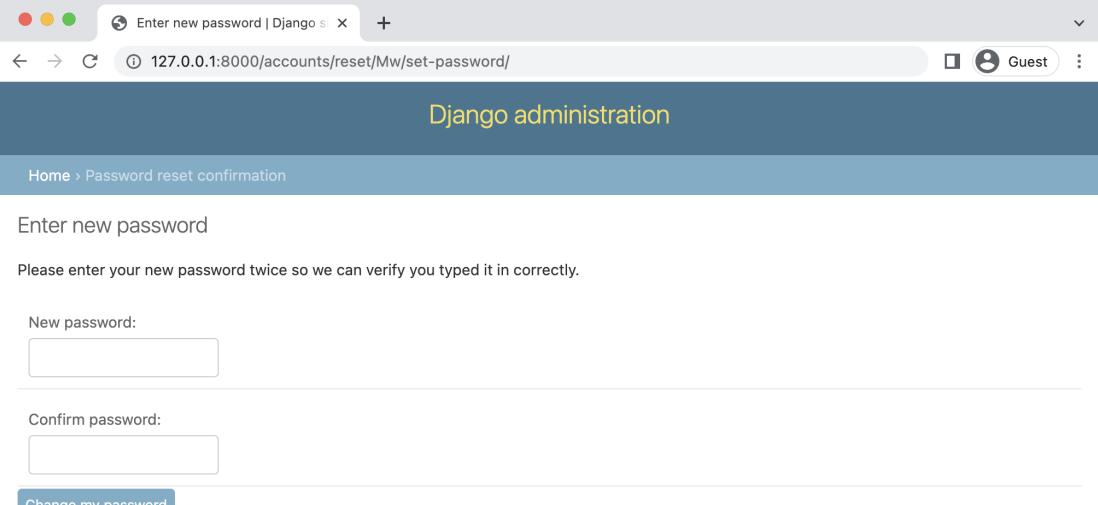
Your email text should be identical except for three lines:

- the “To” on the sixth line contains the email address of the user
- the URL link contains a secure token that Django randomly generates for us and can be used only once
- the username which we’re helpfully reminded of by Django

We will customize all of the default email text shortly, but for now, focus on finding the link provided and entering it into your web browser. In this example, mine is:

<http://127.0.0.1:8000/accounts/reset/Mw/bn63cu-b77b6590a2e38a75c4b2af244c40297e/>

You'll be redirected to the "Password reset confirmation" page.

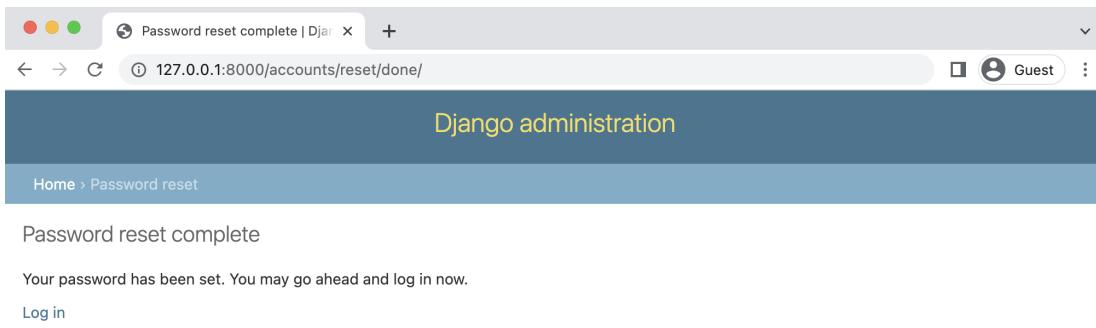


The screenshot shows a web browser window with the following details:

- Title Bar:** Enter new password | Django
- Address Bar:** 127.0.0.1:8000/accounts/reset/Mw/set-password/
- User Information:** Guest
- Page Header:** Django administration
- Breadcrumbs:** Home > Password reset confirmation
- Form Fields:**
  - New password:
  - Confirm password:
- Buttons:** Change my password

### Default password reset confirmation

Enter a new password and click the "Change my password" button. This final step will redirect you to the "Password reset complete" page.



To confirm everything worked, click the “Log in” link and use your new password. It should work.

## Custom Templates

As with the password change pages, we can create new templates to customize the look and feel of the entire password reset flow. If you noticed, there are four separate templates used. Create these new files now in your `templates/registration/` directory.

- `templates/registration/password_reset_form.html`
- `templates/registration/password_reset_done.html`
- `templates/registration/password_reset_confirm.html`
- `templates/registration/password_reset_complete.html`

Start with the password reset form, which is `password_reset_form.html`. At the top, we extend `base.html`, load `crispy_forms_tags`, and set the meta page title. Because we used “block” titles in our `base.html` file, we can override them here. The form uses `POST` since we send data, a `csrf_token` for security reasons, and `{% form|crispy %}` for the forms. And we again update the submit button to be green. At this point, updating these template pages should start to feel somewhat familiar.

## Code

```
<!-- templates/registration/password_reset_form.html -->
{% extends "base.html" %}
{% load crispy_forms_tags %}

{% block title %}Forgot Your Password?{% endblock title %}

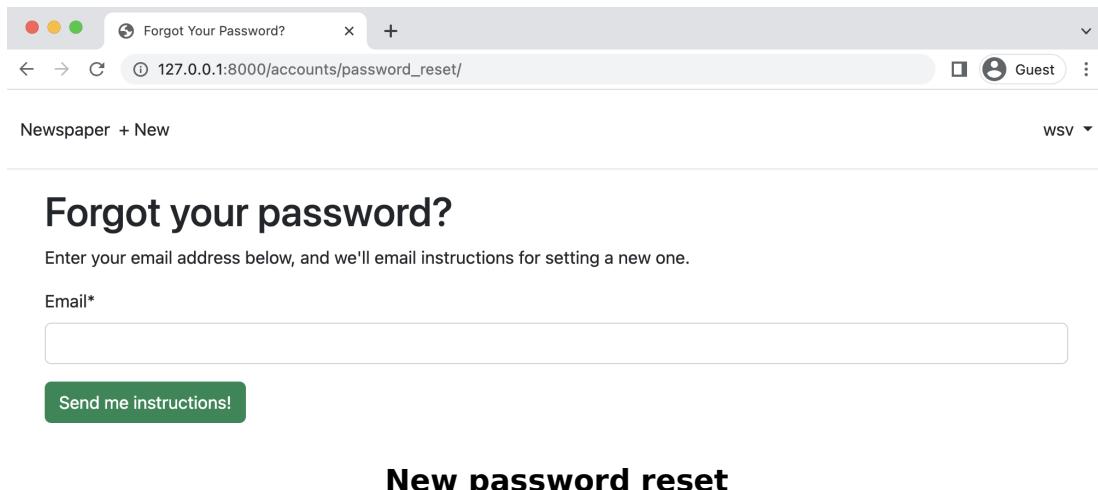
{% block content %}
<h1>Forgot your password?</h1>
<p>Enter your email address below, and we'll email instructions  
for setting a new one.</p>

<form method="POST">{% csrf_token %}
{{ form|crispy }}
<input class="btn btn-success" type="submit"  
value="Send me instructions!">
</form>
{% endblock content %}
```

Start up the server again with `python manage.py runserver` and navigate to:

`http://127.0.0.1:8000/accounts/password_reset/`

Refresh the page and you will see our new page.



Now we can update the other three pages. Each takes the same form of extending `base.html`, setting a new meta title, and adding new content text. When a form is involved, we switch to loading and using crispy forms.

Let's begin with the password\_reset\_done.html template.

Code

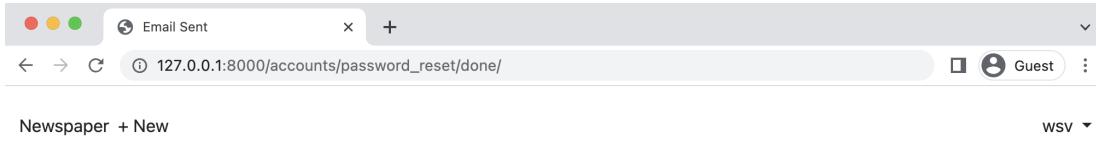
```
<!-- templates/registration/password_reset_done.html -->
{% extends "base.html" %}

{% block title %}Email Sent{% endblock title %}

{% block content %}
<h1>Check your inbox.</h1>
<p>We've emailed you instructions for setting your password.  
You should receive the email shortly!</p>
{% endblock content %}
```

Confirm the changes by going to

[http://127.0.0.1:8000/accounts/password\\_reset/done/](http://127.0.0.1:8000/accounts/password_reset/done/).



## Check your inbox.

We've emailed you instructions for setting your password. You should receive the email shortly!

## New reset done

Next up is password\_reset\_confirm.html. Note that it has a form so we'll use crispy forms here.

Code

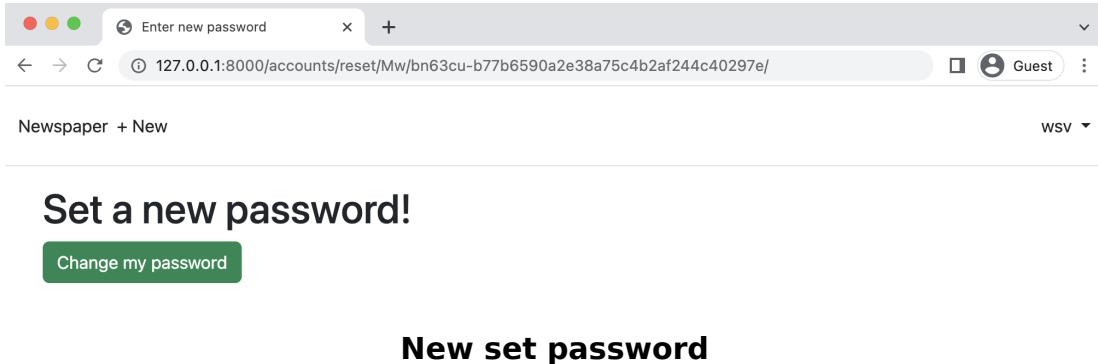
```
<!-- templates/registration/password_reset_confirm.html -->
{% extends "base.html" %}

{% load crispy_forms_tags %}

{% block title %}Enter new password{% endblock title %}

{% block content %}
<h1>Set a new password!</h1>
<form method="POST">{% csrf_token %}
{{ form|crispy }}
<input class="btn btn-success" type="submit" value="Change my password">
</form>
{% endblock content %}
```

In the command line, grab the URL link from the custom email previously outputted to the console, and you'll see the following:



A screenshot of a web browser window titled "Enter new password". The address bar shows the URL "127.0.0.1:8000/accounts/reset/Mw/bn63cu-b77b6590a2e38a75c4b2af244c40297e/". The page content is titled "Set a new password!" with a green "Change my password" button. Below it, a section titled "New set password" is visible.

Finally, here is the password reset complete code:

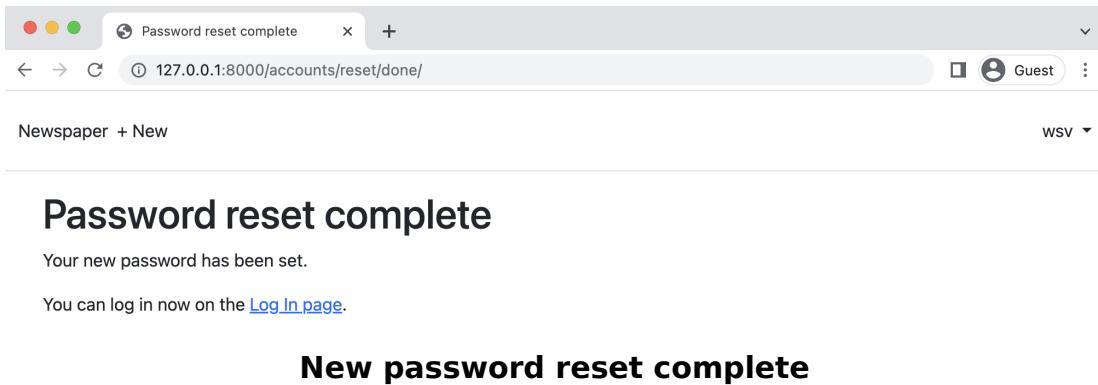
Code

```
<!-- templates/registration/password_reset_complete.html -->
{% extends "base.html" %}

{% block title %}Password reset complete{% endblock title %}

{% block content %}
<h1>Password reset complete</h1>
<p>Your new password has been set.</p>
<p>You can log in now on the
<a href="{% url 'login' %}">Log In page</a>.</p>
{% endblock content %}
```

You can view it at <http://127.0.0.1:8000/accounts/reset/done/>.



A screenshot of a web browser window titled "Password reset complete". The address bar shows the URL "127.0.0.1:8000/accounts/reset/done/". The page content is titled "Password reset complete" and contains the message "Your new password has been set." followed by "You can log in now on the [Log In page](#)". Below it, a section titled "New password reset complete" is visible.

## Try It Out

Let's confirm everything is working by resetting the password for the testuser2 account. Log out of your current account and head to the login page—the logical location for a “Forgot your password?” link that sends a user into the password reset section. Let's add that link now.

First, we'll need to add the password reset link to the existing login page since we can't assume the user will know the correct URL! That goes on the bottom of 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 ml-2" type="submit">Log In</button>
</form>
<!-- new code here -->
<p><a href="{% url 'password_reset' %}">Forgot your password?</a></p>
<!-- end new code -->
{% endblock content %}
```

---

Refresh the login webpage to confirm the new “Forgot your password?” link is there.

Newspaper + New

Log In

Username\*

Password\*

Log In

[Forgot your password?](#)

### Forgot Your Password link

Click on the link to bring up the password reset template and complete the flow using the email for testuser2@email.com. Remember that the unique link will be outputted to your console. Set a new password and use it to log in to the testuser2 account. Everything should work as expected.

## Git

Another chunk of work has been completed. Use Git to save our work before continuing.

### Shell

```
(.venv) $ git status
(.venv) $ git add -A
(.venv) $ git commit -m "password change and reset"
(.venv) $ git push origin main
```

## Conclusion

In the next chapter, we will build out our actual *Newspaper* app that displays articles.

# Chapter 14: Newspaper App

It's time to build out our *Newspaper* app. We will have an articles page where journalists can post articles, set up permissions so only the author of an article can edit or delete it, and finally add the ability for other users to write comments on each article.

## Articles App

To start, create an articles app and define the database models. There are no hard and fast rules around what to name your apps except that you can't use the name of a built-in app. If you look at the `INSTALLED_APPS` section of `django_project/settings.py`, you can see which app names are off-limits:

- admin
- auth
- contenttypes
- sessions
- messages
- staticfiles

A general rule of thumb is to use the plural of an app name: `posts`, `payments`, `users`, etc. One exception would be when doing so is obviously wrong, such as `blogs`. In this case, using the singular `blog` makes more sense.

Start by creating our new articles app.

---

Shell

---

```
(.venv) $ python manage.py startapp articles
```

---

Then add it to our `INSTALLED_APPS` and update the time zone since we'll be timestamping our articles. You can find your time zone in [this Wikipedia list](#). For example, I live in Boston, MA, in the Eastern time zone of the United States; therefore, my entry is `America/New_York`.

Code

---

```
# django_project/settings.py
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    # 3rd Party
    "crispy_forms",
    "crispy_bootstrap5",
    # Local
    "accounts",
    "pages",
    "articles", # new
]
TIME_ZONE = "America/New_York" # new
```

---

Next, we define our database model, which contains four fields: `title`, `body`, `date`, and `author`. We're letting Django automatically set the time and date based on our `TIME_ZONE` setting. For the `author` field, we want to [reference our custom user model](#) `"accounts.CustomUser"` which we set in the `django_project/settings.py` file as `AUTH_USER_MODEL`. We will also implement the best practice of defining a `get_absolute_url` and a `__str__` method for viewing the model in our admin interface.

Code

---

```
# articles/models.py
from django.conf import settings
from django.db import models
from django.urls import reverse

class Article(models.Model):
    title = models.CharField(max_length=255)
    body = models.TextField()
```

```
date = models.DateTimeField(auto_now_add=True)
author = models.ForeignKey(
    settings.AUTH_USER_MODEL,
    on_delete=models.CASCADE,
)

def __str__(self):
    return self.title

def get_absolute_url(self):
    return reverse("article_detail", kwargs={"pk": self.pk})
```

---

There are two ways to refer to a custom user model:  
AUTH\_USER\_MODEL and [get\\_user\\_model](#). As general advice:

- AUTH\_USER\_MODEL makes sense for references within a models.py file
- get\_user\_model() is recommended everywhere else, such as views, tests, etc.

Since we have a new app and model, it's time to make a new migration file and apply it to the database.

#### Shell

---

```
(.venv) $ python manage.py makemigrations articles
(.venv) $ python manage.py migrate
```

---

At this point, I like to jump into the admin to play around with the model before building out the URLs/views/templates needed to display the data on the website. But first, we need to update articles/admin.py so our new app is displayed.

#### Code

---

```
# articles/admin.py
from django.contrib import admin
from .models import Article

admin.site.register(Article)
```

---

Now we start the server.

#### Shell

---

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

Navigate to the admin at <http://127.0.0.1:8000/admin/> and log in.

The screenshot shows the Django administration site's main dashboard. At the top, there's a header bar with the title "Site administration | Django site", the URL "127.0.0.1:8000/admin/", and a "Guest" user indicator. Below the header, the title "Django administration" is displayed. On the left, there's a sidebar with three main categories: "ACCOUNTS" (with "Users" listed), "ARTICLES" (with "Articles" listed), and "AUTHENTICATION AND AUTHORIZATION" (with "Groups" listed). Each category has a "+ Add" button and a "Change" link. To the right of the sidebar, there are two sections: "Recent actions" (empty) and "My actions" (listing a single action for "wsv User"). The overall interface is clean and follows the standard Django admin design.

### Admin page

If you click “+ Add” next to “Articles” at the top of the page, we can enter some sample data. You’ll likely have three users available: your superuser, testuser, and testuser2 accounts. Create new articles using your superuser account as the author. I’ve added three new articles, as you can see on the updated Articles page.

The screenshot shows the Django admin interface. On the left is a sidebar with 'ACCOUNTS' (Users, Groups), 'ARTICLES' (Articles), and 'AUTHENTICATION AND AUTHORIZATION'. The 'ARTICLES' section is selected, and its 'Articles' sub-section is highlighted. In the main area, there is a heading 'Select article to change' and a list of articles. The list includes 'ARTICLE', 'World news', 'Local News', and 'Hello, World!'. At the bottom of the list, it says '3 articles'. At the top right of the main area, there is a button labeled 'ADD ARTICLE'.

### Admin three articles

But wouldn't it be nice to see a little more information in the admin about each article? We can quickly do that by updating `articles/admin.py` with [list\\_display](#).

#### Code

```
# articles/admin.py
from django.contrib import admin
from .models import Article

class ArticleAdmin(admin.ModelAdmin):
    list_display = [
        "title",
        "body",
        "author",
    ]

admin.site.register(Article, ArticleAdmin)
```

What we've done is extend `ModelAdmin`, a class that represents a model in the admin interface. And we made sure to then “register” it at the bottom of the file along with the `Article` model that we imported at the top. There are many customizations available in the Django admin so the official docs are worth a close read.

### Admin three articles with description

If you click on an individual article, you will see that the title, body, and author are displayed but not the date, even though we defined a date field in our model. That's because the date was automatically added by Django for us and, therefore, can't be changed in the admin. We *could* make the date editable-in more complex apps, it's common to have both a `created_at` and `updated_at` attribute-but to keep things simple, we'll have the date be set upon creation by Django for us for now. Even though date is not displayed here, we can still access it in our templates for display on webpages.

## URLs and Views

The next step is to configure our URLs and views. Let's have our articles appear at `articles/`. Add a URL pattern for articles in our `django_project/urls.py` file.

### Code

---

```
# django_project/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
```

```
        path("admin/", admin.site.urls),
        path("accounts/", include("accounts.urls")),
        path("accounts/", include("django.contrib.auth.urls")),
        path("articles/", include("articles.urls")), # new
        path("", include("pages.urls")),
]
```

---

Next, we create a new `articles/urls.py` file in the text editor and populate it with our routes. Let's start with the page to list all articles at `articles/`, which will use the view `ArticleListView`.

Code

---

```
# articles/urls.py
from django.urls import path

from .views import ArticleListView

urlpatterns = [
    path("", ArticleListView.as_view(), name="article_list"),
]
```

---

Now create our view using the built-in generic `ListView` from Django. The only two attributes we need to specify are the model `Article` and our template name, which will be `article_list.html`.

Code

---

```
# articles/views.py
from django.views.generic import ListView

from .models import Article

class ArticleListView(ListView):
    model = Article
    template_name = "article_list.html"
```

---

The last step is to create a new template file in the text editor called `templates/article_list.html`. Bootstrap has a built-in component called [Cards](#) that we can customize for our individual articles. Recall that `ListView` returns an object with `<model_name>_list` that we can iterate using a `for` loop.

We display each article's title, body, author, and date. We can even provide links to the "detail", "edit", and "delete" pages that we haven't built yet.

---

#### Code

---

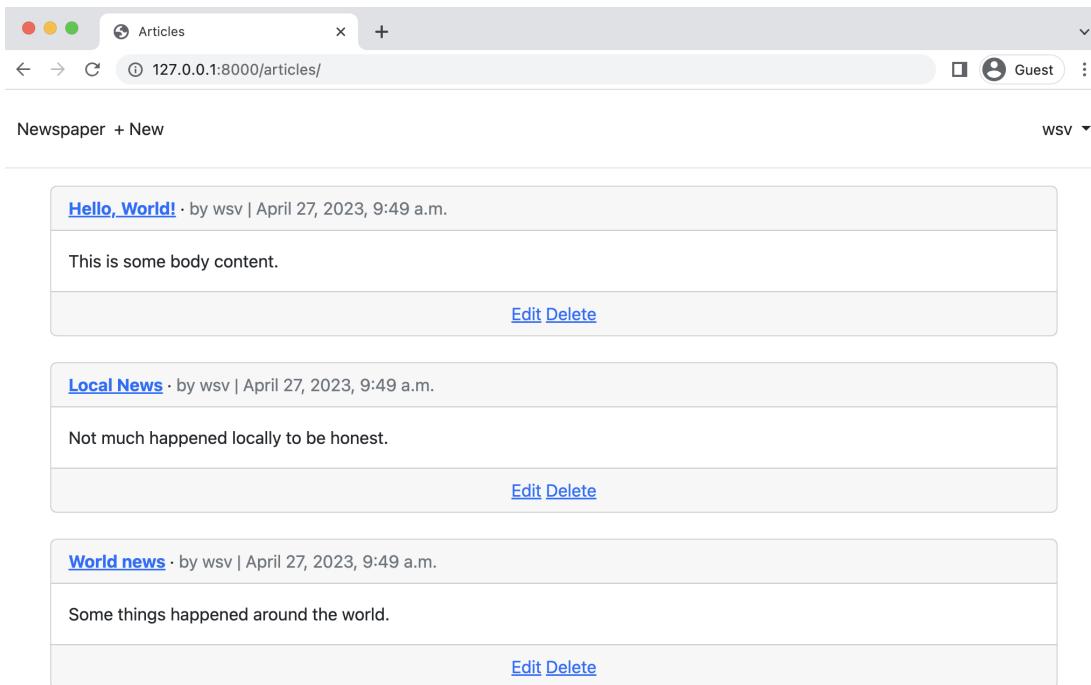
```
<!-- templates/article_list.html -->
{% extends "base.html" %}

{% block title %}Articles{% endblock title %}

{% block content %}
{% for article in article_list %}
<div class="card">
  <div class="card-header">
    <span class="fw-bold">
      <a href="#">{{ article.title }}</a>
    </span> &middot;
    <span class="text-muted">by {{ article.author }} | {{ article.date }}</span>
  </div>
  <div class="card-body">
    {{ article.body }}
  </div>
  <div class="card-footer text-center text-muted">
    <a href="#">Edit</a> <a href="#">Delete</a>
  </div>
</div>
<br />
{% endfor %}
{% endblock content %}
```

---

Start the server again and check out our page at  
<http://127.0.0.1:8000/articles/>.



## Articles page

Not bad, eh? If we wanted to get fancy, we could create a [custom template filter](#) so that the date outputted is shown in seconds, minutes, or days. This can be done with some if/else logic and Django's [date options](#), but we won't implement it here.

## Detail/Edit/Delete

The next step is to add detail, edit, and delete options for the articles. That means new URLs, views, and templates. Let's start with the URLs. The Django ORM automatically adds a primary key to each database entry, meaning that the first article has a `pk` value of 1, the second of 2, and so on. We can use this to craft our URL paths.

For our detail page, we want the route to be at `articles/<int:pk>`. The `int` here is a [path converter](#) and essentially tells Django that we want this value to be treated

as an integer and not another data type like a string. Therefore the URL route for the first article will be at articles/1/. Since we are in the articles app, all URL routes will be prefixed with articles/ because we set that in django\_project/urls.py. We only need to add the <int:pk> part here.

Next up are the edit and delete routes that will also use the primary key. They will be at the URL routes articles/1/edit/ and articles/1/delete/ with the primary key of 1. Here is how the updated articles/urls.py file should look.

---

Code

---

```
# articles/urls.py
from django.urls import path

from .views import (
    ArticleListView,
    ArticleDetailView, # new
    ArticleUpdateView, # new
    ArticleDeleteView, # new
)

urlpatterns = [
    path("<int:pk>", ArticleDetailView.as_view(),
         name="article_detail"), # new
    path("<int:pk>/edit/", ArticleUpdateView.as_view(),
         name="article_edit"), # new
    path("<int:pk>/delete/", ArticleDeleteView.as_view(),
         name="article_delete"), # new
    path("", ArticleListView.as_view(),
         name="article_list"),
]
```

---

We will use Django's generic class-based views for DetailView, UpdateView, and DeleteView. The detail view only requires listing the model and template name. For the update/edit view, we also add the specific attributes title and body—that can be changed. And for the delete view, we must add a redirect for where to send the user after deleting the entry. That requires importing reverse\_lazy and specifying the success\_url along with a corresponding named URL.

## Code

---

```
# articles/views.py
from django.views.generic import ListView, DetailView # new
from django.views.generic.edit import UpdateView, DeleteView # new
from django.urls import reverse_lazy # new
from .models import Article


class ArticleListView(ListView):
    model = Article
    template_name = "article_list.html"


class ArticleDetailView(DetailView): # new
    model = Article
    template_name = "article_detail.html"


class ArticleUpdateView(UpdateView): # new
    model = Article
    fields = (
        "title",
        "body",
    )
    template_name = "article_edit.html"


class ArticleDeleteView(DeleteView): # new
    model = Article
    template_name = "article_delete.html"
    success_url = reverse_lazy("article_list")
```

---

If you recall the acronym CRUD (Create-Read-Update-Delete) from Chapter 6, you'll see that we are implementing three of the four functionalities here. We'll add the fourth, for create, later in this chapter. Almost every website uses CRUD, and this pattern will quickly feel natural when using Django or any other web framework.

The URL paths and views are done, so the final step is to add templates. Create three new template files in your text editor:

- templates/article\_detail.html
- templates/article\_edit.html
- templates/article\_delete.html

We'll start with the details page, displaying the title, date, body, and author with links to edit and delete. It will also link back to all articles. Recall that the Django templating language's `url` tag wants the URL name and then any arguments passed in. The name of our edit route is `article_edit`, and we need to use its primary key, `article.pk`. The delete route name is `article_delete`, and requires a primary key, `article.pk`. Our articles page is a `ListView`, so it does not require any additional arguments passed in.

Code

---

```
<!-- templates/article_detail.html -->
{% extends "base.html" %}

{% block content %}
<div class="article-entry">
    <h2>{{ object.title }}</h2>
    <p>by {{ object.author }} | {{ object.date }}</p>
    <p>{{ object.body }}</p>
</div>
<div>
    <p><a href="{% url 'article_edit' article.pk %}">Edit</a>
        <a href="{% url 'article_delete' article.pk %}">Delete</a>
    </p>
    <p>Back to <a href="{% url 'article_list' %}">All Articles</a>.</p>
</div>
{% endblock content %}
```

---

For the edit and delete pages, we can use Bootstrap's [button styling](#) to make the edit button light blue and the delete button red.

Code

---

```
<!-- templates/article_edit.html -->
{% extends "base.html" %}
{% load crispy_forms_tags %}

{% block content %}
<h1>Edit</h1>
<form action="" method="post">{{ csrf_token }}
    {{ form|crispy }}
    <button class="btn btn-info ml-2" type="submit">Update</button>
</form>
{% endblock content %}
```

---

Code

---

```
<!-- templates/article_delete.html -->
{% extends "base.html" %}

{% block content %}
<h1>Delete</h1>
<form action="" method="post">{% csrf_token %}
    <p>Are you sure you want to delete "{{ article.title }}"?</p>
    <button class="btn btn-danger ml-2" type="submit">Confirm</button>
</form>
{% endblock content %}
```

---

As a final step, in `article_list.html`, we can now add URL routes for detail, edit, and delete pages to replace the existing `<a href="#">` placeholders. For the detail page, we can use the `get_absolute_url` method defined in our model. And we can use the `url` template tag, the URL name, and the `pk` of each article for the edit and delete links.

#### Code

---

```
<!-- templates/article_list.html -->
{% extends "base.html" %}

{% block title %}Articles{% endblock title %}

{% block content %}
{% for article in article_list %}
<div class="card">
    <div class="card-header">
        <span class="fw-bold">
            <!-- add link here! -->
            <a href="{{ article.get_absolute_url }}>{{ article.title }}</a>
        </span> &middot;
        <span class="text-muted">by {{ article.author }} | {{ article.date }}</span>
    </div>
    <div class="card-body">
        {{ article.body }}
    </div>
    <div class="card-footer text-center text-muted">
        <!-- new links here! -->
        <a href="{% url 'article_edit' article.pk %}">Edit</a>
        <a href="{% url 'article_delete' article.pk %}">Delete</a>
    </div>
</div>
<br />
{% endfor %}
{% endblock content %}
```

---

Ok, we're ready to view our work. Start the server with `python manage.py runserver` and navigate to the articles list page at `http://127.0.0.1:8000/articles/`. Click the “Edit” link next to the first article, and you'll be redirected to `http://127.0.0.1:8000/articles/1/edit/`.

The screenshot shows a web browser window titled "Newspaper App". The address bar displays the URL `127.0.0.1:8000/articles/1/edit/`. The page content is titled "Edit". It contains two form fields: "Title\*" with the value "Hello, World!" and "Body\*" with the value "This is some body content.". A blue "Update" button is located at the bottom left of the form.

### Edit page

If you update the “Title” attribute by adding “(edited)” at the end and click “Update”, you'll be redirected to the detail page, which shows the new change.



Newspaper + New

wsv ▾

## Hello, World! (edited)

by wsv | April 27, 2023, 9:49 a.m.

This is some body content.

[Edit](#) [Delete](#)

Back to [All Articles](#).

### Detail page

You'll be redirected to the delete page if you click the "Delete" link.



Newspaper + New

wsv ▾

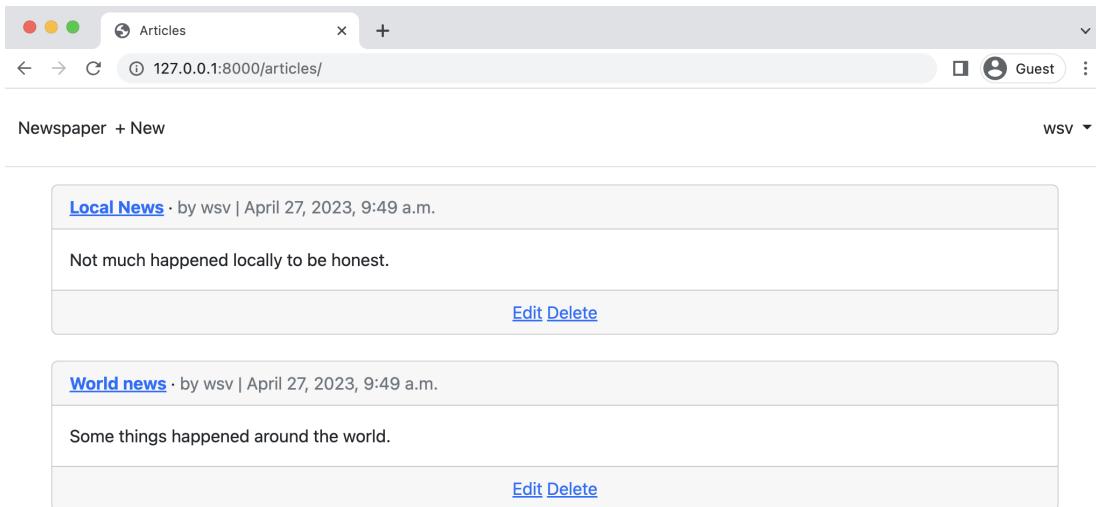
## Delete

Are you sure you want to delete "Hello, World! (edited)"?

[Confirm](#)

### Delete page

Press the scary red button for "Confirm." You'll be redirected to the articles page, which now only has two entries.



## Articles page two entries

## Create Page

The final step is a create page for new articles, which we can implement with Django's built-in `CreateView`. Our three steps are to create a view, URL, and template. This flow should feel familiar by now.

In the `articles/views.py` file, add `CreateView` to the imports at the top and make a new class at the bottom of the file called `ArticleCreateView` that specifies our model, template, and the fields available.

### Code

```
# articles/views.py
...
from django.views.generic.edit import (
    UpdateView, DeleteView, CreateView # new
)
...
class ArticleCreateView(CreateView): # new
    model = Article
    template_name = "article_new.html"
    fields = (
        "title",
        "body",
```

```
        "author",
    )
```

---

Note that our fields attribute has author since we want to associate a new article with an author; however, once an article has been created, we do not want a user to be able to change the author, which is why ArticleUpdateView only has the attributes ['title', 'body',].

Now update the articles/urls.py file with the new route for the view.

Code

---

```
# articles/urls.py
from django.urls import path

from .views import (
    ArticleListView,
    ArticleDetailView,
    ArticleUpdateView,
    ArticleDeleteView,
    ArticleCreateView, # new
)

urlpatterns = [
    path("<int:pk>/",
        ArticleDetailView.as_view(), name="article_detail"),
    path("<int:pk>/edit/",
        ArticleUpdateView.as_view(), name="article_edit"),
    path("<int:pk>/delete/",
        ArticleDeleteView.as_view(), name="article_delete"),
    path("new/", ArticleCreateView.as_view(), name="article_new"), # new
    path("", ArticleListView.as_view(), name="article_list"),
]
```

---

To complete the new create functionality, add a template named templates/article\_new.html and update it with the following HTML code.

Code

---

```
<!-- templates/article_new.html -->
{% extends "base.html" %}
{% load crispy_forms_tags %}

{% block content %}
<h1>New article</h1>
<form action="" method="post">{% csrf_token %}
```

```
    {{ form|crispy }}
```

```
<button class="btn btn-success ml-2" type="submit">Save</button>
```

```
</form>
```

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

---

## Additional Links

We should add the URL link for creating new articles to our navbar, so it is accessible everywhere on the site to logged-in users.

Code

---

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

```
...
```

```
{% if user.is_authenticated %}
```

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

```
    class="nav-link px-2 link-dark">+ New</a></li>
```

```
...
```

---

Refreshing the webpage and clicking “+ New” will redirect to the create new article page.

Newspaper + New

## New article

Title\*

Body\*

Author\*

**Save**

### New article page

One final link to add is to make the articles list page accessible from the home page. Currently, a user would need to know or guess that it is located at <http://127.0.0.1:8000/articles/>, but we can fix that by adding a button link to the `templates/home.html` file.

#### Code

```
<!-- templates/home.html -->
{% extends "base.html" %}

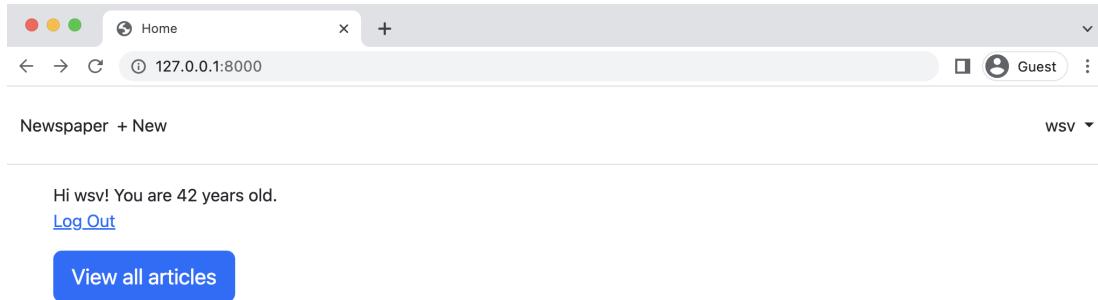
{% block title %}Home{% endblock title %}

{% block content %}
{% if user.is_authenticated %}
Hi {{ user.username }}! You are {{ user.age }} years old.
<p><a href="{% url 'logout' %}">Log Out</a></p>
<p><a class="btn btn-primary btn-lg" href="{% url 'article_list' %}" role="button">
    View all articles</a></p> <!-- new -->
{% 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 %}
```

---

Refresh the homepage and the button will appear and work as intended.



### Home page with all articles link

If you need help to make sure your HTML file is accurate now, please refer to the [official source code](#).

## Git

We added quite a lot of new code in this chapter, so let's save it with Git before proceeding to the next chapter.

### Shell

---

```
(.venv) $ git status  
(.venv) $ git add -A  
(.venv) $ git commit -m "newspaper app"  
(.venv) $ git push origin main
```

---

## Conclusion

We have now created a dedicated articles app with CRUD functionality. Articles can be created, read, updated, deleted, and even viewed as an entire list. But there are no permissions or authorizations yet, which means anyone can

do anything! If logged-out users know the correct URLs, they can make edits to an existing article or delete it, even one that's not their own! In the next chapter, we will add permissions and authorizations to our project to fix this.

# Chapter 15: Permissions and Authorization

There are several issues with our current *newspaper* website. For one thing, we want our newspaper to be financially sustainable, and, with more time, we could add a dedicated payments app to charge for access. But at a minimum, we want to add rules around permissions and authorization: we will require a user to log in to view articles. This is known as *authorization*. Note that this is different than *authentication*, which is the process of registering and logging-in users. Authorization restricts access; authentication enables users to sign up, log in, log out, and so on.

As a mature web framework, Django has built-in authorization functionality that we can use quickly. In this chapter, we'll limit access to the articles list page to only logged-in users and add additional restrictions so that only the author of an article can edit or delete it.

## Improved CreateView

Currently, the `author` on a new article can be set to any existing user. Instead, it should be automatically set to the currently logged-in user. We can modify Django's `CreateView` to achieve this by removing `author` from the `fields` and setting it automatically via the `form_valid` method instead.

Code

---

```
# articles/views.py
class ArticleCreateView(CreateView):
    model = Article
    template_name = "article_new.html"
    fields = ("title", "body") # new

    def form_valid(self, form): # new
```

```
form.instance.author = self.request.user  
return super().form_valid(form)
```

---

How did I know I could update `CreateView` like this? The answer is, I looked at the source code and used [Classy Class-Based Views](#), an amazing resource that breaks down how each generic class-based view works in Django. Generic class-based views are great, but when you want to customize them, you must roll up your sleeves and start to understand what's happening under the hood. This is the downside of class-based views vs. function-based views: more is hidden, and the inheritance chain must be understood. The more you use and customize built-in views, the more comfortable you will become with making customizations like this. Generally, a specific method, like `form_valid` can be overridden to achieve your desired result instead of having to rewrite everything from scratch yourself.

Now reload the browser and try clicking on the “+ New” link in the top nav. It will redirect to the updated create page where `author` is no longer a field.

Newspaper App

127.0.0.1:8000/articles/new/

Newspaper + New

WSV ▾

## New article

Title\*

Body\*

Save

### New article link

If you create a new article and go into the admin, you will see it is automatically set to the currently logged-in user.

## Authorizations

There are multiple issues around the lack of authorizations in our current project. Obviously, we would like to restrict access to only users, so we can one-day charge readers to access our newspaper. But beyond that, any random logged-out user who knows the correct URL can access any part of the site.

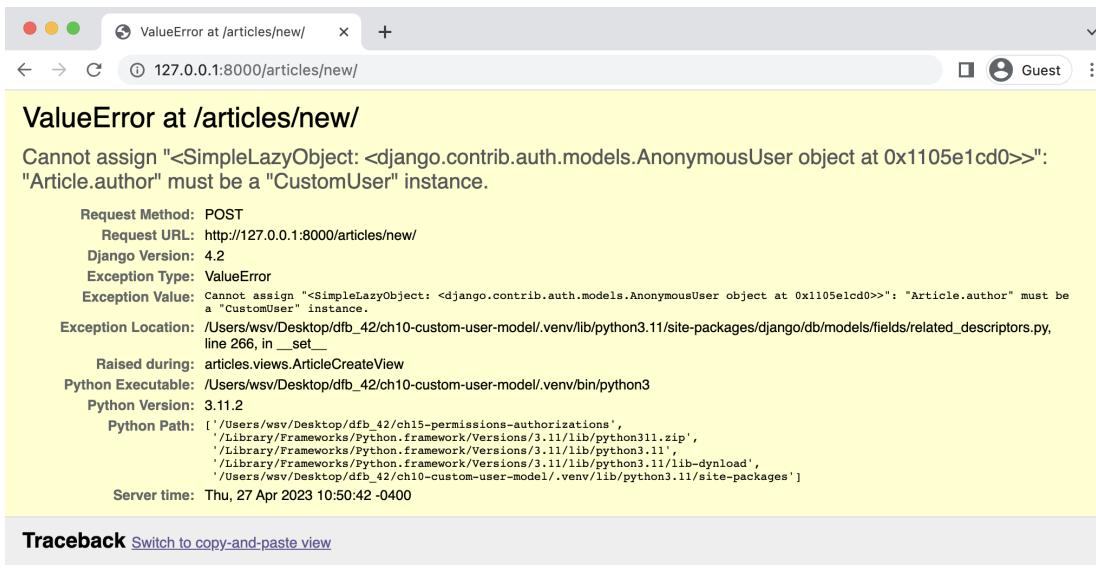
Consider what would happen if a logged-out user tried to create a new article? To try it out, click on your username in the upper right corner of the navbar, then select “Log Out” from the dropdown options. The “+ New” link disappears from the navbar, but what happens if you go to it directly:

<http://127.0.0.1:8000/articles/new/>

The page is still there.

A screenshot of a web browser window titled "Newspaper App". The address bar shows the URL "127.0.0.1:8000/articles/new/". The page content is titled "New article". It contains two input fields: "Title\*" and "Body\*". Below the fields is a green "Save" button. At the top right of the page, there are "Log In" and "Sign Up" buttons. The status bar at the bottom right says "Logged out new".

Now try to create a new article with a title and body. Click on the “Save” button.



## Create page error

An error! This is because our model **expects** an author field which is linked to the currently logged-in user. But since we are not logged in, there's no author, so the submission fails. What to do?

## Mixins

We want to set some authorizations so only logged-in users can access specific URLs. To do this, we can use a *mixin*, a special kind of multiple inheritance that Django uses to avoid duplicate code and still allow customization. For example, the built-in generic [ListView](#) needs a way to return a template. But so does [DetailView](#) and almost every other view. Rather than repeat the same code in each big generic view, Django breaks out this functionality into a mixin known as [TemplateResponseMixin](#). Both `ListView` and `DetailView` use this mixin to render the proper template.

If you read the Django source code, which is freely available [on Github](#), you'll see mixins used everywhere. To restrict view access to only logged-in users, Django has a

[LoginRequired mixin](#) that we can use. It's powerful and extremely concise.

In the articles/views.py file, import LoginRequiredMixin and add it to ArticleCreateView. Make sure that the mixin is to the left of CreateView so it will be read first. We want the CreateView to know we intend to restrict access.

And that's it! We're done.

---

Code

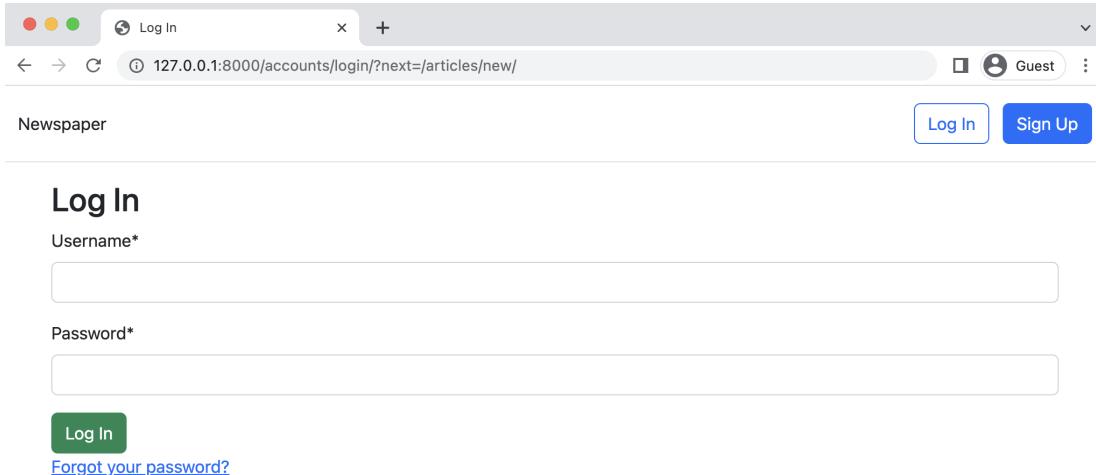
---

```
# articles/views.py
from django.contrib.auth.mixins import LoginRequiredMixin # new
from django.views.generic import ListView, DetailView
...
class ArticleCreateView(LoginRequiredMixin, CreateView): # new
    ...

```

---

Return to the homepage at <http://127.0.0.1:8000/> to avoid resubmitting the form. Navigate to <http://127.0.0.1:8000/articles/new/> again to access the URL route for a new article.



**Log In Redirect Page**

What's happening? Django automatically redirected users to the login page! If you look closely, the URL is `http://127.0.0.1:8000/accounts/login/?next=/articles/new/`, which shows we *tried* to go to `articles/new/` but were instead redirected to log in.

## LoginRequiredMixin

Restricting view access requires adding `LoginRequiredMixin` at the beginning of all existing views. Let's update the rest of our `articles` views since we don't want a user to be able to create, read, update, or delete an article if they aren't logged in.

The complete `views.py` file should now look like this:

Code

---

```
# articles/views.py
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import ListView, DetailView
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from django.urls import reverse_lazy

from .models import Article

class ArticleListView(LoginRequiredMixin, ListView): # new
    model = Article
    template_name = "article_list.html"

class ArticleDetailView(LoginRequiredMixin, DetailView): # new
    model = Article
    template_name = "article_detail.html"

class ArticleUpdateView(LoginRequiredMixin, UpdateView): # new
    model = Article
    fields = (
        "title",
        "body",
    )
    template_name = "article_edit.html"

class ArticleDeleteView(LoginRequiredMixin, DeleteView): # new
    model = Article
```

```
template_name = "article_delete.html"
success_url = reverse_lazy("article_list")

class ArticleCreateView(LoginRequiredMixin, CreateView):
    model = Article
    template_name = "article_new.html"
    fields = ("title", "body",)

    def form_valid(self, form):
        form.instance.author = self.request.user
        return super().form_valid(form)
```

---

Go ahead and play around with the site to confirm that redirecting to the login works as expected. If you need help recalling the proper URLs, log in first and write down the URLs for each route to create, edit, delete, and list all articles.

## UpdateView and DeleteView

We're making progress, but there's still the issue of our edit and delete views. Any *logged-in* user can make changes to any article, and we want to restrict this access so that only the author of an article has this permission.

We could add permissions logic to each view for this, but a more elegant solution is to create a dedicated mixin, a class with a particular feature we want to reuse in our Django code. And better yet, Django ships with a built-in mixin, [UserPassesTestMixin](#), just for this purpose!

To use `UserPassesTestMixin`, first, import it at the top of the `articles/views.py` file and then add it to both the update and delete views where we want this restriction. The `test_func` method is used by `UserPassesTestMixin` for our logic; we need to override it. In this case, we set the variable `obj` to the current object returned by the view using `get_object()`. Then we say, if the `author` on the current object matches the current user on the webpage (whoever is logged in and

trying to make the change), then allow it. If false, an error will automatically be thrown.

The code looks like this:

---

Code

---

```
# articles/views.py
from django.contrib.auth.mixins import (
    LoginRequiredMixin,
    UserPassesTestMixin # new
)
from django.views.generic import ListView, DetailView
from django.views.generic.edit import UpdateView, DeleteView, CreateView
from django.urls import reverse_lazy

from .models import Article
...
class ArticleUpdateView(
    LoginRequiredMixin, UserPassesTestMixin, UpdateView): # new
    model = Article
    fields = (
        "title",
        "body",
    )
    template_name = "article_edit.html"

    def test_func(self): # new
        obj = self.get_object()
        return obj.author == self.request.user

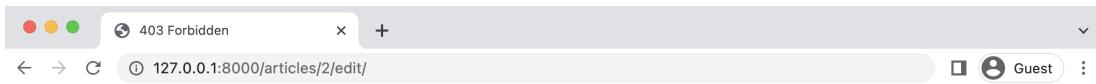

class ArticleDeleteView(
    LoginRequiredMixin, UserPassesTestMixin, DeleteView): # new
    model = Article
    template_name = "article_delete.html"
    success_url = reverse_lazy("article_list")

    def test_func(self): # new
        obj = self.get_object()
        return obj.author == self.request.user
```

---

When using mixins with class-based views, the order is critical. `LoginRequiredMixin` comes first so that we force login, then we add `UserPassesTestMixin` for an additional layer of functionality, and finally, either `UpdateView` or `DeleteView`. The code will not work properly if you do not have this order.

Log in with your testuser account and go to the articles list page. If the code works, you should not be able to edit or delete any posts written by your superuser account; instead, you will see a Permission Denied 403 error page.



## 403 Forbidden

### 403 error page

However, if you create a new article with testuser, you *will* be able to edit and delete it. And if you log in with your superuser account instead, you can edit and delete posts written by that author.

## Template Logic

Although we have successfully restricted access to the “edit” and “delete” pages for each article, they are still present on the list all articles page and the individual article page. It would be better not to display them to users who cannot access them. In other words, we want to restrict their display to only the owner of an article.

We can do this by adding simple logic to our two template files: `templates/article_list.html` and `templates/article_detail.html`. Use Django templates built-in `if` filter so that only the article author can see the edit and delete links.

### Code

---

```
<!-- templates/article_list.html -->
...
<div class="card-footer text-center text-muted">
<!-- new code here -->
{% if article.author.pk == request.user.pk %}
<a href="{% url 'article_edit' article.pk %}">Edit</a>
```

```
<a href="{% url 'article_delete' article.pk %}">Delete</a>
{%
  endif %}
<!-- new code here --&gt;
&lt;/div&gt;
...
</pre>

---


```

## Code

```
<!-- templates/article_detail.html -->
{% extends "base.html" %}

{% block content %}
<div class="article-entry">
  <h2>{{ object.title }}</h2>
  <p>by {{ object.author }} | {{ object.date }}</p>
  <p>{{ object.body }}</p>
</div>
<div>
  <!-- new code here -->
  {% if article.author.pk == request.user.pk %}
    <p><a href="{% url 'article_edit' article.pk %}">Edit</a>
      <a href="{% url 'article_delete' article.pk %}">Delete</a>
    </p>
  {% endif %}
  <!-- new code here -->
  <p>Back to <a href="{% url 'article_list' %}">All Articles</a>.</p>
</div>
{% endblock content %}
```

---

To make sure that our new edit/delete logic works as intended, log out of your superuser account and log in as testuser. Then create a new article using the “+ New” button in the top navbar. Now if you refresh the all articles webpage, only the article authored by testuser should have the edit and delete links visible.

[Local News](#) · by wsv | April 27, 2023, 9:49 a.m.  
Not much happened locally to be honest.

[World news](#) · by wsv | April 27, 2023, 9:49 a.m.  
Some things happened around the world.

[A new article](#) · by testuser | April 27, 2023, 10:58 a.m.  
Seeing if this works  
[Edit](#) [Delete](#)

### Edit/delete links not shown

Click on the article name to navigate to its detail page. The edit and delete links are visible.

[A new article](#) · by testuser | April 27, 2023, 10:58 a.m.  
Seeing if this works  
[Edit](#) [Delete](#)  
Back to [All Articles](#).

### Edit/delete links shown for testuser

However, if you navigate to the detail page of an article created by superuser, the links are gone.

The screenshot shows a web browser window titled "Newspaper App". The URL in the address bar is "127.0.0.1:8000/articles/2/". The page content includes a header "Newspaper + New", a user dropdown "testuser", and a title "Local News" by "wsv | April 27, 2023, 9:49 a.m.". Below the title is a short text snippet: "Not much happened locally to be honest." There is a link "Back to [All Articles](#)".

### Edit/delete links not shown

## Git

A quick save with Git is in order as we finish this chapter.

### Shell

```
(.venv) $ git status  
(.venv) $ git add -A  
(.venv) $ git commit -m "permissions and authorizations"  
(.venv) $ git push origin main
```

## Conclusion

Our newspaper app is almost done. We could take further steps at this point, such as only displaying edit and delete links to the appropriate users, which would involve [custom template tags](#), but overall the app is in good shape. Our articles are correctly configured, set permissions and authorizations, and have a working user authentication flow. The last item needed is the ability for fellow logged-in users to leave comments which we'll cover in the next chapter.

# Chapter 16: Comments

We could add comments to our *newspaper* site in two ways. The first is to create a dedicated *comments* app and link it to *articles*; however, that seems like over-engineering. Instead, we can add a model called *Comment* to our *articles* app and link it to the *Article* model through a foreign key. We will take the more straightforward approach since adding more complexity later is always possible.

The whole Django structure of one project containing multiple smaller apps is designed to help the developer reason about the website. The computer doesn't care how the code is structured. Breaking functionality into smaller pieces helps us—and future teammates—understand the logic in a web application. But you don't need to optimize prematurely. If your eventual comments logic becomes lengthy, then yes, by all means, spin it off into its own comments app. But the first thing to do is make the code work, make sure it is performant, and structure it, so it's understandable to you or someone else months later.

What do we need to add comments functionality to our website? We already know it will involve models, URLs, views, templates, and in this case, also, forms. We need all four for the final solution, but the order in which we tackle them is largely up to us. That said, many Django developers find that going in this order—models -> URLs -> views -> templates/forms—works best, so that is what we will use here. By the end of this chapter, users will be able to add comments to any existing article on our website.

## Model

Let's begin by adding another table to our existing database called Comment. This model will have a many-to-one foreign key relationship to Article: one article can have many comments, but not the other way around. Traditionally the name of the foreign key field is simply the model it links to, so this field will be called article. The other two fields will be comment and author.

Open up the file articles/models.py and underneath the existing code, add the following. Note that we include \_\_str\_\_ and get\_absolute\_url methods as best practices.

---

Code

---

```
# articles/models.py
...
class Comment(models.Model): # new
    article = models.ForeignKey(Article, on_delete=models.CASCADE)
    comment = models.CharField(max_length=140)
    author = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
    )

    def __str__(self):
        return self.comment

    def get_absolute_url(self):
        return reverse("article_list")
```

---

Since we've updated our models, it's time to make a new migration file and apply it. Note that by adding articles at the end of the makemigrations command—which is optional—we are specifying we want to use just the articles app here. This is a good habit to use. For example, what if we changed models in two different apps? If we **did not** specify an app, then both apps' changes would be incorporated in the same migrations file, making it harder to debug errors in the future. Keep each migration as small and contained as possible.

Shell

---

```
(.venv) $ python manage.py makemigrations articles
(.venv) $ python manage.py migrate
```

---

## Admin

After making a new model, playing around with it in the admin app before displaying it on our website is a good idea. Add Comment to our admin.py file so it will be visible.

Code

---

```
# articles/admin.py
from django.contrib import admin
from .models import Article, Comment # new

class ArticleAdmin(admin.ModelAdmin):
    list_display = [
        "title",
        "body",
        "author",
    ]

admin.site.register(Article, ArticleAdmin)
admin.site.register(Comment) # new
```

---

Then start the server with `python manage.py runserver` and navigate to our main page `http://127.0.0.1:8000/admin/`.

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', '127.0.0.1:8000/admin/', 'Guest', and a user icon. The main title is 'Django administration'. On the left, there are three main sections: 'ACCOUNTS' (with 'Users' listed), 'ARTICLES' (with 'Articles' and 'Comments' listed), and 'AUTHENTICATION AND AUTHORIZATION' (with 'Groups' listed). Each section has a '+ Add' and a 'Change' link. To the right, a sidebar titled 'Recent actions' lists the following entries:

- Hello, World! Article
- Local News Article
- World news Article
- World news Article
- Local News Article
- Hello, World! Article
- WSV User

## Admin homepage with comments

Under our “Articles” app, you’ll see our two tables: Comments and Articles. Click on the “+ Add” next to Comments. There are dropdowns for Article, Author, and a text field next to Comment.

The screenshot shows the 'Add comment' form in the Django administration interface at the URL `127.0.0.1:8000/admin/articles/comment/add/`. The top navigation bar and sidebar are identical to the previous screenshot. The main content area is titled 'Add comment' and contains three fields: 'Article:' with a dropdown menu, 'Comment:' with a text input field, and 'Author:' with a dropdown menu. At the bottom, there are three buttons: 'SAVE', 'Save and add another', and 'Save and continue editing'.

## Admin comments

Select an article, write a comment, and then choose an author that is not your superuser, perhaps testuser as I've done in the picture. Then click on the "Save" button.

Django administration

Home > Articles > Comments > Add comment

Start typing to filter...

ACCOUNTS

Users + Add

ARTICLES

Articles + Add

Comments + Add

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Add comment

Article: A new article

Comment: My first comment

Author: testuser

SAVE Save and add another Save and continue editing

### Admin testuser comment

You should next see your comment on the admin "Comments" page.

Select comment to change

WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT

The comment "My first comment" was added successfully.

ADD COMMENT +

Action:	Go	0 of 1 selected
<input type="checkbox"/> COMMENT		
<input type="checkbox"/> My first comment		

1 comment

### Admin comment one

At this point, we could add an additional admin field to see the comment and the article on this page. But wouldn't it be better to see all Comment models related to a single Article model? It turns out we can with a Django admin feature called *inlines* which displays foreign key relationships in a visual way.

There are two main inline views used: [TabularInline](#) and [StackedInline](#). The only difference between the two is the template for displaying information. In a TabularInline, all model fields appear on one line; in a StackedInline, each field has its own line. We'll implement both so you can decide which one you prefer.

Update `articles/admin.py` in your text editor to add the `StackedInline` view.

---

Code

```
# articles/admin.py
from django.contrib import admin
from .models import Article, Comment

class CommentInline(admin.StackedInline): # new
    model = Comment

class ArticleAdmin(admin.ModelAdmin): # new
    inlines = [
        CommentInline,
    ]
    list_display = [
        "title",
        "body",
        "author",
    ]

admin.site.register(Article, ArticleAdmin) # new
admin.site.register(Comment)
```

---

Now go to the main admin page at `http://127.0.0.1:8000/admin/` and click “Articles.” Select the

article you just commented on, which was “A new article” in my case.

The screenshot shows the Django administration interface for a 'Change article' page. The top navigation bar includes links for 'A new article | Change article', '127.0.0.1:8000/admin/articles/article/4/change/', 'Guest', and 'WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT'. The main content area is titled 'Change article' and shows a single article entry titled 'A new article'. The 'Body' field contains the text 'Seeing if this works'. Below this, the 'Author' field is set to 'testuser'. A 'COMMENTS' section follows, displaying four comments: 'Comment: My first comment' (author testuser), 'Comment: #2' (author -----), 'Comment: #3' (author -----), and 'Comment: #4' (author -----). At the bottom of the page, the text 'Admin change page' is displayed.

Better, right? We can see and modify all our related articles and comments in one place. Note that, by default, the Django admin will display three empty rows here. You can change the default number that appears with the extra field. So if you wanted no extra fields by default, the code would look like this:

## Code

```
# articles/admin.py
...
class CommentInline(admin.StackedInline):
    model = Comment
    extra = 0 # new
```

The screenshot shows the Django administration interface for creating a new article. The left sidebar lists 'ACCOUNTS' (Users), 'ARTICLES' (Articles, Comments), and 'AUTHENTICATION AND AUTHORIZATION' (Groups). The main area is titled 'Change article' and 'A new article'. The 'Title:' field contains 'A new article'. The 'Body:' field contains 'Seeing if this works'. The 'Author:' dropdown is set to 'testuser'. Below this, a 'COMMENTS' section shows a single comment: 'Comment: My first comment' with a link 'View on site' and a 'Delete' checkbox. The comment text is 'My first comment' and the author is 'testuser'. There is also a '+ Add another Comment' link. At the bottom are buttons for 'SAVE', 'Save and add another', 'Save and continue editing', and a red 'Delete' button.

## Admin no extra comments

Personally, though, I prefer using `TabularInline` as it shows more information in less space: the comment, author, and more on one single line. To switch to it, we only need to change our `CommentInline` from `admin.StackedInline` to `admin.TabularInline`.

## Code

---

```
# articles/admin.py
from django.contrib import admin
from .models import Article, Comment

class CommentInline(admin.TabularInline): # new
    model = Comment
    extra = 0 #

class ArticleAdmin(admin.ModelAdmin):
    inlines = [
        CommentInline,
    ]
    list_display = [
        "title",
        "body",
        "author",
    ]

admin.site.register(Article, ArticleAdmin)
admin.site.register(Comment)
```

---

Refresh the current admin page for Articles and you'll see the new change: all fields for each model are displayed on the same line.

A new article | Change article | +

127.0.0.1:8000/admin/articles/article/4/change/

Django administration

WELCOME, **WSV**. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home > Articles > Articles > A new article

Start typing to filter...

ACCOUNTS

Users [+ Add](#)

ARTICLES

Articles [+ Add](#)

Comments [+ Add](#)

AUTHENTICATION AND AUTHORIZATION

Groups [+ Add](#)

Change article

**A new article**

Title:

Body:

Seeing if this works

Author:  [edit](#) [add](#) [view](#)

COMMENTS

COMMENT	AUTHOR	DELETE?
My first comment <a href="#">View on site</a>	testuser	<input type="checkbox"/>

[+ Add another Comment](#)

**SAVE** **Save and add another** **Save and continue editing** **Delete**

### TabularInline page

Much better. Now we need to display the comments on our website by updating our template.

## Template

We want comments to appear on the articles list page and allow logged-in users to add a comment on the detail page for an article. That means updating the template files `article_list.html` and `article_detail.html`.

Let's start with `article_list.html`. If you look at the `articles/models.py` file again, it is clear that `Comment` has a foreign key relationship to the `article`. To display **all**

comments related to a specific article, we will [follow the relationship backward](#) via a “query,” which is a way to ask the database for a specific bit of information. Django has a built-in syntax known as `Foo_set` where `Foo` is the lowercase source model name. So for our `Article` model, we use the syntax `{% for comment in article.comment_set.all %}` to view all related comments. And then, within this `for` loop, we can specify what to display, such as the `comment` itself and author.

Here is the updated `article_list.html` file –the changes start after the “`card-body`” `div` class.

---

#### Code

---

```
<!-- templates/article_list.html -->
...
<div class="card-body">
    {{ article.body }}
    {% if article.author.pk == request.user.pk %}
        <a href="{% url 'article_edit' article.pk %}">Edit</a>
        <a href="{% url 'article_delete' article.pk %}">Delete</a>
    {% endif %}
</div>
<div class="card-footer">
    {% for comment in article.comment_set.all %}
        <p>
            <span class="fw-bold">
                {{ comment.author }} ···
            </span>
            {{ comment }}
        </p>
    {% endfor %}
</div>
</div>
...
```

---

If you refresh the articles page at `http://127.0.0.1:8000/articles/`, we can see our new comment on the page.

[Local News](#) · by wsv | April 27, 2023, 9:49 a.m.  
Not much happened locally to be honest. [Edit](#) [Delete](#)

[World news](#) · by wsv | April 27, 2023, 9:49 a.m.  
Some things happened around the world. [Edit](#) [Delete](#)

[A new article](#) · by testuser | April 27, 2023, 10:58 a.m.  
Seeing if this works  
testuser · My first comment

## Articles page with comments

Let's also add comments to the detail page for each article. We'll use the exact same technique of following the relationship backward to access comments as a foreign key of the article model.

### Code

```
<!-- templates/article_detail.html -->
{% extends "base.html" %}

{% block content %}
<div class="article-entry">
    <h2>{{ object.title }}</h2>
    <p>by {{ object.author }} | {{ object.date }}</p>
    <p>{{ object.body }}</p>
</div>

<!-- Changes start here! -->
<hr>
<h4>Comments</h4>
{% for comment in article.comment_set.all %}
<p>{{ comment.author }} &middot; {{ comment }}</p>
{% endfor %}
<hr>
<!-- Changes end here! -->
```

```
{% if article.author.pk == request.user.pk %}  
  <p><a href="{% url 'article_edit' article.pk %}">Edit</a>  
    <a href="{% url 'article_delete' article.pk %}">Delete</a>  
  </p>  
{% endif %}  
<p>Back to <a href="{% url 'article_list' %}">All Articles</a>.</p>  
{% endblock content %}
```

---

Navigate to the detail page of your article with a comment, and any comments will be visible.

The screenshot shows a web browser window titled "Newspaper App". The address bar displays "127.0.0.1:8000/articles/4/". The page content is as follows:

Newspaper + New

WSV ▾

## A new article

by testuser | April 27, 2023, 10:58 a.m.

Seeing if this works

---

### Comments

testuser · My first comment

---

Back to [All Articles](#).

### Article details page with comments

We won't win any design awards for this layout, but this is a book on Django, so outputting the correct content is our goal.

## Comment Form

The comments are now visible, but we need to add a form so users can add them to the website. Web forms are a very complicated topic since security is essential: any time you accept data from a user that will be stored in a database, you must be highly cautious. The good news is Django forms handle most of this work for us.

[ModelForm](#) is a helper class designed to translate database models into forms. We can use it to create a form called, appropriately enough, `CommentForm`. We *could* put this form in our existing `articles/models.py` file, but generally, the best practice is to put all forms in a dedicated `forms.py` file within your app. That's the approach we'll use here.

With your text editor, create a new file called `articles/forms.py`. At the top, import `forms`, which has `ModelForm` as a module. Then import our model, `Comment`, since we'll need to add that, too. And finally, create the class `CommentForm` specifying both the underlying model and specific field to expose, `comment`. When we create the corresponding view, we will automatically set the `author` to the currently logged-in user.

---

#### Code

---

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

from .models import Comment

class CommentForm(forms.ModelForm):
    class Meta:
        model = Comment
        fields = ("comment",)
```

---

Web forms can be incredibly complex, but Django has thankfully abstracted away much of the complexity for us.

## Comment View

Currently, we rely on the generic class-based `DetailView` to power our `ArticleDetailView`. It displays individual entries, but it needs to be configured to add additional information like a form. Class-based views are powerful because their inheritance structure means that if we know where to look,

there is often a specific module we can override to attain our desired outcome.

The one we want in this case is called `get_context_data()`. It is used to add information to a template by updating the `context`, a dictionary object containing all the variable names and values available in our template. For performance reasons, Django templates compile only once; if we want something available in the template, it must load into the context at the beginning.

What do we want to add in this case? Well, just our `CommentForm`. And since context is a dictionary, we must also assign a variable name. How about `form`? Here is what the new code looks like in `articles/views.py`.

---

#### Code

---

```
# articles/views.py
...
from .models import Article
from .forms import CommentForm # new

class ArticleDetailView(LoginRequiredMixin, DetailView):
    model = Article
    template_name = "article_detail.html"

    def get_context_data(self, **kwargs): # new
        context = super().get_context_data(**kwargs)
        context["form"] = CommentForm()
        return context
```

---

Near the top of the file, just above `from .models import Article`, we added an import line for `CommentForm` and then updated the module for `get_context_data()`. First, we pulled all existing information into the context using `super()`, added the variable name `form` with the value of `CommentForm()`, and returned the updated context.

## Comment Template

To display the form in our `article_detail.html` template file, we'll rely on the `form` variable and crispy forms. This pattern is the same as what we've done before in our other forms. At the top, load `crispy_form_tags`, create a standard-looking post form that uses a `csrf_token` for security, and display our form fields via `{{ form|crispy }}`.

---

### Code

---

```
<!-- templates/article_detail.html -->
{% extends "base.html" %}
{% load crispy_forms_tags %} <!-- new! -->

{% block content %}
<div class="article-entry">
    <h2>{{ object.title }}</h2>
    <p>by {{ object.author }} | {{ object.date }}</p>
    <p>{{ object.body }}</p>
</div>

<hr>
<h4>Comments</h4>
{% for comment in article.comment_set.all %}
    <p>{{ comment.author }} &middot; {{ comment }}</p>
{% endfor %}
<hr>

<!-- Changes start here! -->
<h4>Add a comment</h4>
<form action="" method="post">{% csrf_token %}
    {{ form|crispy }}
    <button class="btn btn-success ml-2" type="submit">Save</button>
</form>
<!-- Changes end here! -->

<div>
    {% if article.author.pk == request.user.pk %}
        <p><a href="{% url 'article_edit' article.pk %}">Edit</a>
            <a href="{% url 'article_delete' article.pk %}">Delete</a>
        </p>
    {% endif %}
    <p>Back to <a href="{% url 'article_list' %}">All Articles</a>.</p>
</div>
{% endblock content %}
```

---

If you refresh the detail page, the form is now displayed with familiar Bootstrap and crispy forms styling.

Newspaper + New

WSV ▾

## A new article

by testuser | April 27, 2023, 10:58 a.m.

Seeing if this works

---

### Comments

testuser · My first comment

---

#### Add a comment

Comment\*

Save

Back to [All Articles.](#)

### Form displayed

Success! However, we are only half done. If you attempt to submit the form, you'll receive an error because our view doesn't yet support any POST methods!

## Comment Post View

We ultimately need a view that handles both GET and POST requests depending upon whether the form should be merely displayed or capable of being submitted. We could reach for [FormMixin](#) to combine both into our ArticleDetailView, but as the [Django docs illustrate quite well](#), there are risks with this approach.

To avoid subtle interactions between DetailView and FormMixin, we will separate the GET and POST variations into their dedicated views. We can then transform ArticleDetailView into a *wrapper view* that combines them. This is a very common pattern in more advanced Django development because a single URL must often behave

differently based on the user request (GET, POST, etc.) or even the format (returning HTML vs. JSON).

Let's start by renaming ArticleDetailView into CommentGet since it handles GET requests but not POST requests. We'll then create a new CommentPost view that is empty for now. And we can combine both CommentPost and CommentGet into a new ArticleDetailView that subclasses [View](#), the foundational class upon which all other class-based views are built.

---

#### Code

---

```
# articles/views.py
from django.contrib.auth.mixins import LoginRequiredMixin, UserPassesTestMixin
from django.views import View # new
from django.views.generic import ListView, DetailView
...
class CommentGet(DetailView): # new
    model = Article
    template_name = "article_detail.html"

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context["form"] = CommentForm()
        return context

class CommentPost(): # new
    pass

class ArticleDetailView(LoginRequiredMixin, View): # new
    def get(self, request, *args, **kwargs):
        view = CommentGet.as_view()
        return view(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        view = CommentPost.as_view()
        return view(request, *args, **kwargs)
...

```

---

Navigate back to the homepage in your web browser and then reload the article page with a comment. Everything should work as before.

We're ready to write `CommentPost` and complete our task of adding comments to our website. We are almost done!

[FormView](#) is a built-in view that displays a form, any validation errors, and redirects to a new URL. We will use it with [SingleObjectMixin](#) to associate the current article with our form; in other words, if you have a comment at `articles/4/`, as I do in the screenshots, then `SingleObjectMixin` will grab the 4 so that our comment is saved to the article with a `pk` of 4.

Here is the complete code, which we'll run through below line-by-line.

---

#### Code

---

```
# articles/views.py
from django.contrib.auth.mixins import LoginRequiredMixin, UserPassesTestMixin
from django.views import View
from django.views.generic import ListView, DetailView, FormView # new
from django.views.generic.detail import SingleObjectMixin # new
from django.views.generic.edit import UpdateView, DeleteView, CreateView
from django.urls import reverse_lazy, reverse # new

from .forms import CommentForm
from .models import Article

...
class CommentPost(SingleObjectMixin, FormView): # new
    model = Article
    form_class = CommentForm
    template_name = "article_detail.html"

    def post(self, request, *args, **kwargs):
        self.object = self.get_object()
        return super().post(request, *args, **kwargs)

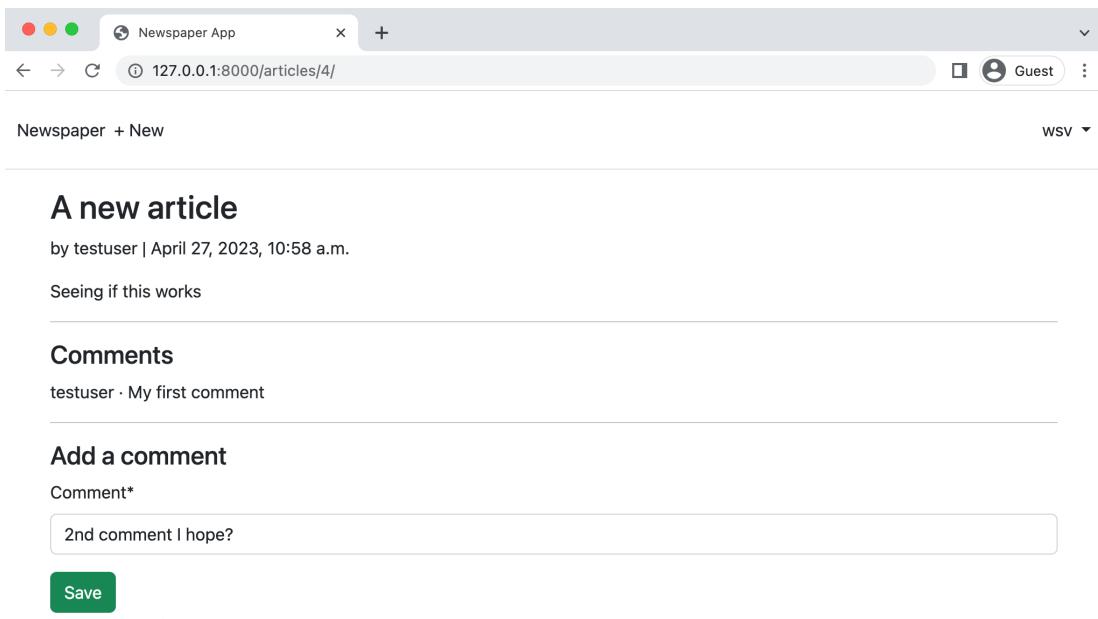
    def form_valid(self, form):
        comment = form.save(commit=False)
        comment.article = self.object
        comment.author = self.request.user
        comment.save()
        return super().form_valid(form)

    def get_success_url(self):
        article = self.object
        return reverse("article_detail", kwargs={"pk": article.pk})
...
```

---

At the top, import `FormView`, `SingleObjectMixin`, and `reverse`. `FormView` relies on `form_class` to set the form name we're using, `CommentForm`. First up is `post()`: we use `get_object()` from `SingleObjectMixin` to grab the article `pk` from the URL. Next is `form_valid()`, which is called when form validation has succeeded. Before we save our comment to the database, we must specify the article it belongs to. Initially, we save the form but set `commit` to `False` because we associate the correct article with the form object in the next line. We also set the `author` field in our `Comment` model to the current user. In the following line, we save the form. Finally, we return it as part of `form_valid()`. The final module, `get_success_url()`, is called after the form data is saved; we redirect the user to the current page in this case.

And we're done! Go ahead and load your articles page now, refresh the page, then try to submit a second comment.



The screenshot shows a web browser window titled "Newspaper App". The address bar displays "127.0.0.1:8000/articles/4/". The main content area shows an article titled "A new article" by "testuser" on April 27, 2023, at 10:58 a.m. The article content is "Seeing if this works". Below the article is a "Comments" section. It shows one comment from "testuser" with the text "My first comment". There is a "Comments" section with a "Save" button and a "Back to All Articles" link. At the bottom, there is a "Submit comment in form" button.

Newspaper + New

WSV ▾

## A new article

by testuser | April 27, 2023, 10:58 a.m.

Seeing if this works

---

### Comments

testuser · My first comment

---

#### Add a comment

Comment\*

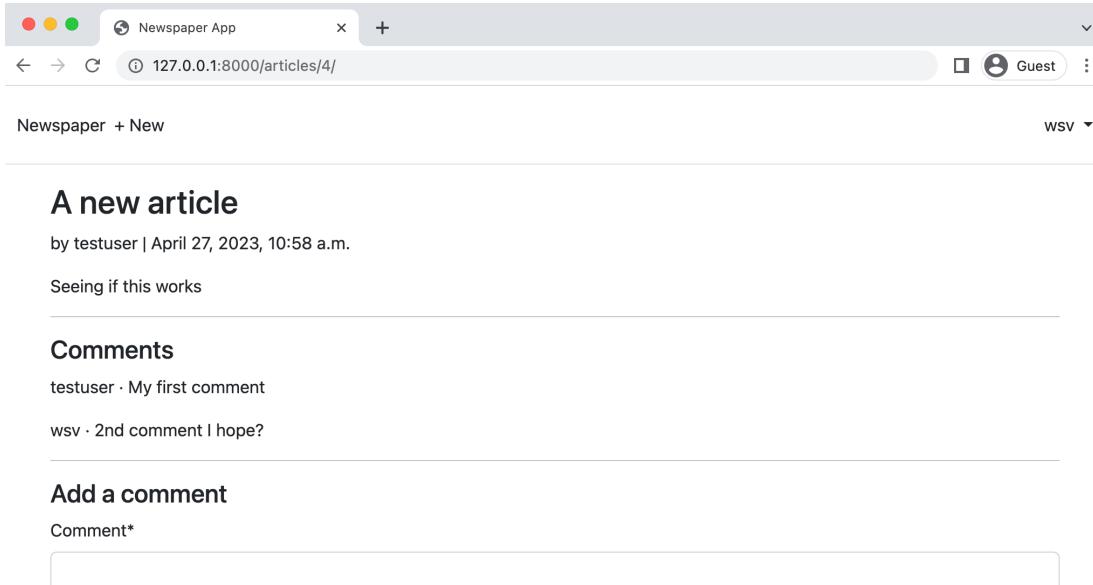
2nd comment I hope?

Save

Back to [All Articles](#).

**Submit comment in form**

It should automatically reload the page with the new comment displayed like this:



The screenshot shows a web browser window titled "Newspaper App" at the URL "127.0.0.1:8000/articles/4/". The page displays an article titled "A new article" by "testuser" on April 27, 2023, at 10:58 a.m. A comment from "testuser" reads "Seeing if this works". Below it, a comment from "wsv" reads "2nd comment I hope?". A "Comments" section shows one comment from "testuser" and one from "wsv". A "Add a comment" form is present, with a "Comment\*" input field, a "Save" button, and a link to "All Articles".

**Comment displayed**

## New Comment Link

Although you can add a comment to an article from its detail page, it is far more likely that a reader will want to comment on something from the all articles page. They can do that if they know to click on the detail link but that is not very good user design. Let's add a "New Comment" link to each article listed; it will navigate to the detail page but allow for that functionality. Do so by adding one line to the card-body section outside the if/endif loop.

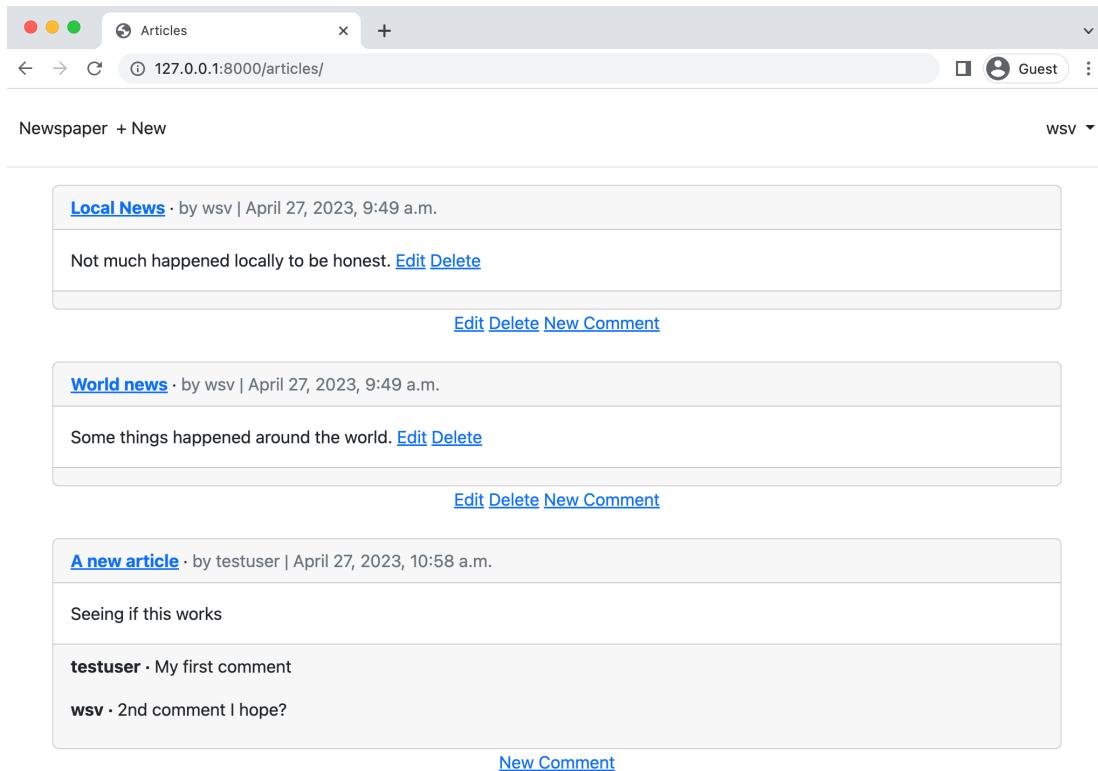
### Code

```
<!-- templates/article_list.html -->
...
<div class="card-body">
    <p>{{ article.body }}</p>
    {% if article.author.pk == request.user.pk %}
```

```
<a href="{% url 'article_edit' article.pk %}">Edit</a>
<a href="{% url 'article_delete' article.pk %}">Delete</a>
{% endif %}
<a href="{{ article.get_absolute_url }}">New Comment</a> <!-- new -->
</div>
```

---

Refresh the all articles webpage to see the change and then click the “New Comment” link to confirm it works as expected.



### New comment on all articles page

## Git

We added quite a lot of code in this chapter. Let's make sure to save our work before the upcoming final chapter.

### Shell

```
(.venv) $ git status
(.venv) $ git add -A
```

```
(.venv) $ git commit -m "comments app"  
(.venv) $ git push origin main
```

---

## Conclusion

Our *Newspaper* app is now complete. It has a robust user authentication flow that uses a custom user model and email. Improved styling thanks to Bootstrap, articles, and comments. We even dipped our toes into permissions and authorizations.

Our remaining task is to deploy it online. Our deployment process has grown in complexity with each successive application, but we're still taking shortcuts around security and performance. In the next chapter, we'll see how to properly deploy a Django site using environment variables, PostgreSQL, and additional settings.

# Chapter 17: Deployment

There is a fundamental tension between the ease of use desired in a local Django development environment and the security and performance necessary in a production environment. Django is designed to make web developers' lives easier and defaults to a local configuration when the `startproject` command is first run: SQLite as the file-based local database and various configurations in `settings.py` that all optimize local web development. In this final chapter, we will configure static files, review our deployment checklist, and deploy the *newspaper* website in a manner suitable for a professional website.

## Static Files

The *newspaper* project has no static files like CSS, JavaScript, or images. We've relied entirely on hosted Bootstrap rather than our own CSS and JavaScript as we did in the *Blog* app. That is likely to change as the site grows, and we've seen that the Django admin has its own static files that must be collected properly for production, so let's set up static files now.

Since we already walked through static files in some depth previously, we can move quicker now. We will need to create a dedicated static folder and within it folders for `css`, `js`, and `img` for images. This can all be done on the command line.

---

### Shell

---

```
(.venv) $ mkdir static
(.venv) $ mkdir static/css
(.venv) $ mkdir static/js
(.venv) $ mkdir static/img
```

---

A quirk of Git is that it will not track empty folders containing no files. Since the `css`, `js`, and `img` folders are empty right now, they won't be pushed up to GitHub. One solution is to add a `.keep` file to an otherwise empty directory. Add an empty `.keep` file to the three subfolders in your text editor: `static/css/.keep`, `static/js/.keep`, and `static/img/.keep`.

In production, it is desirable to have one single location for all static files across the entire project. Four configurations in `django_project/settings.py` need to be set so the `collectstatic` management command works correctly and compiles them all as expected.

- `STATIC_URL` is the URL location of all static files in production
- `STATICFILES_DIRS` refers to any additional locations for static files beyond an app's `static` folder
- `STATIC_ROOT` is the directory location of all static files compiled when `collectstatic` is run
- `STORAGES` is a dictionary containing the settings for all storages to be used by Django

In the `django_project/settings.py` file, scroll down to the bottom where there is an existing section on static files and `STATIC_URL = "static/"`. Update that section with the two new configurations for `STATICFILES_DIRS` and `STATIC_ROOT`. And make the default `STORAGES` settings for `default` and `staticfiles` explicit before we update them in the next section for `WhiteNoise`.

---

Code

```
# django_project/settings.py
STATIC_URL = "static/"
STATICFILES_DIRS = [BASE_DIR / "static"] # new
STATIC_ROOT = BASE_DIR / "staticfiles" # new
STORAGES = {
    "default": {
        "BACKEND": "django.core.files.storage.FileSystemStorage",
```

```
        },
        "staticfiles": {
            "BACKEND": "django.contrib.staticfiles.storage.StaticFilesStorage",
        },
    }
```

---

Now run `python manage.py collectstatic`.

#### Shell

```
(.venv) $ python manage.py collectstatic
```

---

Since the only static files in the project at the moment are contained within the built-in admin app, a new `staticfiles` directory should appear with sections for the admin. When we add static files ourselves, they will also be compiled in this directory.

#### Code

```
# staticfiles/
└── admin
    ├── css
    ├── img
    └── js
```

---

As a final step, for our templates to display any static files in the future, they must be loaded in, so add `{% load static %}` to the top of the `base.html` file.

#### Code

```
<!-- templates/base.html -->
{% load static %}
<!DOCTYPE html>
...

---


```

## Deployment Checklist

This will now be our fourth deployment in the book. Let's review the complete list used for the *Blog* app.

- install Gunicorn as a WSGI server
- install Psycopg to connect with a PostgreSQL database

- install environs for environment variables
- update DATABASES in django\_project/settings.py
- install WhiteNoise for static files
- create a requirements.txt file
- add a .dockerignore file
- create a .env file
- update a .gitignore file
- update ALLOWED\_HOSTS, CSRF\_TRUSTED\_ORIGINS, DEBUG, and SECRET\_KEY

## Gunicorn, Psycopg, and environs

To begin, we can install Gunicorn, Psycopg, and environs using pip.

---

### Shell

```
(.venv) $ python -m pip install gunicorn==20.1.0
(.venv) $ python -m pip install "psycopg[binary]"==3.1.8
(.venv) $ python -m pip install "environs[django]"==9.5.0
```

---

Then update django\_project/settings.py with three new lines so environment variables can be loaded in.

---

### Code

```
# django_project/settings.py
from pathlib import Path
from environs import Env # new

env = Env() # new
env.read_env() # new
```

---

## DATABASES

Next, update the DATABASES setting in the django\_project/settings.py file so that SQLite is used locally but PostgreSQL in production. This elegant solution works because we installed the environs[django] package containing several Django-specific packages, including dj-

database-url, which looks for a DATABASE\_URL environment variable and automatically configures things for us.

---

#### Code

---

```
# django_project/settings.py
DATABASES = {
    "default": env.dj_db_url("DATABASE_URL", default="sqlite:///db.sqlite3"),
}
```

---

## WhiteNoise

We will need to install WhiteNoise to serve our static files in production, so let's install that package.

---

#### Shell

---

```
(.venv) $ python -m pip install whitenoise==6.4.0
```

---

WhiteNoise must be added to django\_project/settings.py in the following locations:

- whitenoise above django.contrib.staticfiles in INSTALLED\_APPS
- WhiteNoiseMiddleware above CommonMiddleware
- staticfiles alias within STORAGES pointing to WhiteNoise

---

#### Code

---

```
# django_project/settings.py
INSTALLED_APPS = [
    ...
    "whitenoise.runserver_nostatic",  # new
    "django.contrib.staticfiles",
]

MIDDLEWARE = [
    "django.middleware.security.SecurityMiddleware",
    "django.contrib.sessions.middleware.SessionMiddleware",
    "whitenoise.middleware.WhiteNoiseMiddleware",  # new
    ...
]

STORAGES = {
    "default": {
        "BACKEND": "django.core.files.storage.FileSystemStorage",
    },
    "staticfiles": {
```

```
"BACKEND": "whitenoise.storage.CompressedManifestStaticFilesStorage",
# new
},
}
```

---

Run the `collectstatic` command again. The prompt will warn about overwriting existing files, but that is intentional: we want to compile them using WhiteNoise now. Type yes and press Return to continue:

Shell

```
(.venv) $ python manage.py collectstatic
```

---

## requirements.txt

All additional third-party packages are installed so we can generate a `requirements.txt` file containing the contents of our virtual environment.

Shell

```
(.venv) $ python -m pip freeze > requirements.txt
```

---

My `requirements.txt` file looks like this:

Code

```
# requirements.txt
asgi==3.6.0
black==23.3.0
click==8.1.3
crispy-bootstrap5==0.7
dj-database-url==2.0.0
dj-email-url==1.0.6
Django==4.2
django-cache-url==3.4.4
django-crispy-forms==2.0
environs==9.5.0
gunicorn==20.1.0
marshmallow==3.19.0
mypy-extensions==1.0.0
packaging==23.1
pathspec==0.11.1
platformdirs==3.2.0
psycopg==3.1.8
psycopg-binary==3.1.8
python-dotenv==1.0.0
```

```
sqlparse==0.4.4
typing_extensions==4.5.0
whitenoise==6.4.0
```

---

## .dockerignore and .env

Then we can create two hidden files: `.dockerignore` to tell Docker what to ignore and a `.env` file containing environment variables meant to be secret and not tracked by Git. Both files should exist in the root directory next to the existing `.gitignore` file. Go ahead and create them now in your text editor.

---

.dockerignore

---

```
.venv/
__pycache__/
*.sqlite3
.git
.env
```

---

---

.gitignore

---

```
.venv/
__pycache__/
*.sqlite3
.env # new
```

---

## DEBUG and SECRET\_KEY

When `DEBUG` is set to `True`, as it is now, Django displays rich error pages for us, which is useful for local development but a major security risk in production. We want it to remain `True` for local usage yet be set to `False` once deployed. This can be done with environment variables, as we've seen previously.

Update the `.env` file and set `DEBUG` to `True`.

---

.env

---

```
DEBUG=True # new
```

---

Then update the DEBUG configuration in django\_project/settings.py to load in that environment variable.

Code

---

```
# django_project/settings.py
DEBUG = env.bool("DEBUG", default=False)
```

---

The SECRET\_KEY is a randomly generated string that provides cryptographic signing throughout our project. It needs to be secure. On the command line, generate a new string for us to use since the existing one that starts with django-insecure-... has already been committed to Git and is, therefore, insecure.

Shell

---

```
(.venv) $ python -c "import secrets; print(secrets.token_urlsafe())"
4Dx5f3TRmyGGQJ5U19eozyMy8s1hRoDh7d5DUSfc8ZU
```

---

Update the .env file with this new value.

.env

---

```
DEBUG=True
SECRET_KEY=4Dx5f3TRmyGGQJ5U19eozyMy8s1hRoDh7d5DUSfc8ZU # new
```

---

Then update the existing SECRET\_KEY configuration in django\_project/settings.py to point to this new environment variable.

Code

---

```
# django_project/settings.py
SECRET_KEY = env.str("SECRET_KEY")
```

---

## **ALLOWED\_HOSTS and CSRF\_TRUSTED\_ORIGINS**

ALLOWED\_HOSTS lists all of the host/domain names our Django website can serve, while CSRF\_TRUSTED\_ORIGINS is a list of trusted origins for unsafe requests like POSTs. Both need to

be explicitly set now and once we have our production URLs from Fly, we can lock them down properly.

Code

---

```
# django_project/settings.py
ALLOWED_HOSTS = [".fly.dev", "localhost", "127.0.0.1"] # new
CSRF_TRUSTED_ORIGINS = ["https://*.fly.dev"] # new
```

---

## Fly Deployment

Log into your Fly account using `flyctl auth login` from the command line.

Shell

---

```
(.venv) $ flyctl auth login
```

---

We'll run `fly launch` to see our production URL and then immediately add it to the `ALLOWED_HOSTS` and `CSRF_TRUSTED_ORIGINS` settings before deployment. In the wizard, we will respond to the questions as follows:

- **Choose an app name:** this will be your dedicated `fly.dev` subdomain
- **Choose the region for deployment:** select the one closest to you or [another region](#) if you prefer
- **Decline overwriting our `.dockerignore` file:** our choices are already optimized for the project
- **Setup a Postgres database cluster:** the “Development” option is appropriate for this project. [Fly Postgres is a regular app deployed to Fly.io, not a managed database.](#)
- **Select “Yes” to scale a single node pg to zero after one hour:** this will save money for toy projects
- **Decline to setup a Redis database:** we don't need one for this project

Shell

---

```
(.venv) $ fly launch
Creating app in ~/desktop/code/news
Scanning source code
Detected a Django app
? Choose an app name (leave blank to generate one): dfb-ch17-news
automatically selected personal organization: Will Vincent
Some regions require a paid plan (fra, maa).
See https://fly.io/plans to set up a plan.
? Choose a region for deployment: Boston, Massachusetts (US) (bos)
App will use 'bos' region as primary
Created app 'dfb-ch17-news' in organization 'personal'
Admin URL: https://fly.io/apps/dfb-ch17-news
Hostname: dfb-ch17-news.fly.dev
? Overwrite "/Users/wsv/Desktop/dfb_ch8/.dockerignore"? No
Set secrets on dfb-ch17-news: SECRET_KEY
? Would you like to set up a Postgresql database now? Yes
? Select configuration: Development - Single node, 1x shared CPU, 256MB RAM,
1GB
? Scale single node pg to zero after one hour? Yes
Creating postgres cluster in organization personal
Creating app...
...
? Would you like to set up an Upstash Redis database now? No
Wrote config file fly.toml
Your app is ready! Deploy with `flyctl deploy`
```

---

My hostname here is `dfb-ch17-news.fly.dev`, but your hostname will naturally differ. Add your hostname to the `ALLOWED_HOSTS` and `CSRF_TRUSTED_ORIGINS` settings.

---

#### Code

---

```
# django_project/settings.py
ALLOWED_HOSTS = ["dfb-ch17-news.fly.dev", "localhost", "127.0.0.1"] # new
CSRF_TRUSTED_ORIGINS = ["https://dfb-ch17-news.fly.dev"] # new
```

---

We should be all set. Before deploying our project to Fly servers, add another Git commit with all the changes that have occurred.

---

#### Shell

---

```
(.venv) $ git status
(.venv) $ git add -A
(.venv) $ git commit -m "Fly deployment"
(.venv) $ git push origin main
```

---

Run the command `flyctl deploy` to deploy the project on Fly servers and open up the website in a web browser via the

`fly open` command.

---

#### Shell

---

```
(.venv) $ flyctl deploy  
(.venv) $ fly open
```

---

If there are any issues with your deployment, you can use `fly logs` to see recent log file entries or go to `fly.io` to see the full dashboard.

## Production Database

Remember that we now have a production PostgreSQL database that is empty. It does not contain any newspaper articles or users we've added to our local version. SSH into the production database and create a superuser account.

---

#### Shell

---

```
(.venv) $ fly ssh console --pty -C "python /code/manage.py createsuperuser"
```

---

Then visit the `/admin` page, create some new articles, and refresh the main homepage; they should appear. You can also create user accounts and confirm that the user authentication flow works correctly by resetting your password.

For future updates to Fly.io, you should first save the changes to GitHub and then use the command `flyctl deploy` to update the code on your production website.

## Deployment Conclusion

Phew! We just covered a ton of material so you likely feel overwhelmed right now. That's normal. There are many steps involved in configuring a website for proper deployment, and the good news is that this same list of

production settings will hold true for almost every Django project. Don't worry about memorizing all the steps.

After you've built and deployed several Django websites, these steps will soon feel very familiar. And in fact, we've only scratched the surface of additional security measures that can be configured. Django has its own [deployment checklist](#) that can be run via the command line to highlight additional security issues.

The other big stumbling block for newcomers is becoming comfortable with the difference between local and production environments. You will likely forget to push code changes into production and spend minutes or hours wondering why the change isn't live on your site. Or even worse, you'll change your local SQLite database and expect them to magically appear in the production PostgreSQL database. It's part of the learning process. But Django really does make it much smoother than it otherwise would be. And now you know enough to deploy any Django project online confidently.

# Chapter 18: Conclusion

Congratulations on finishing *Django for Beginners!* After starting from absolute zero, we've built five different web applications from scratch and covered Django's major features: templates, views, URLs, users, models, security, testing, and deployment. You now have the knowledge to go off and build modern websites with Django.

As with any new skill, practicing and applying what you've just learned is important. The CRUD (Create-Read-Update-Delete) functionality in our *Blog* and *Newspaper* sites is commonplace in many other web applications. For example, can you make a Todo List web application? A Twitter or Facebook clone? You already have all the tools you need. When starting out, the best approach is to build as many small projects as possible, incrementally add complexity, and research new things.

## Next Steps

We explored a lot of Django material in this book, but there is still much more to learn, especially if you want to build large websites that simultaneously handle thousands or millions of visitors. Django is more than capable of this, and I've written a follow-up book called [Django for Professionals](#) that tackles many of the challenges around building truly *production-ready* websites such as using Docker, using a production database like PostgreSQL locally, incorporating advanced user registration, including additional security, improving performance, and much more.

Django is also frequently used to create back-end APIs consumable by mobile apps (iOS/Android) or websites that

use a dedicated JavaScript front-end framework such as Vue, React, or Angular. Thanks to the power of [Django REST Framework](#), a third-party package tightly coupled with Django itself, it is possible to transform any existing Django website into an API with a minimal amount of code. If you'd like to learn more, I've written a book called [Django for APIs](#).

## 3rd Party Packages

As we've seen in this book, third-party packages are a vital part of the Django ecosystem, especially regarding deployment or improvements around user registration. It's common for a professional Django website to rely on dozens of such packages.

However, a word of caution is in order: don't blindly install and use third-party packages because it saves a small amount of time. Every additional package introduces another dependency, another risk that its maintainer won't fix every bug or keep up to date with the latest version of Django. Take the time to understand what it is doing.

If you'd like to view more packages, the [Django Packages](#) website is a comprehensive resource of all available third-party packages. A more curated option, the [awesome-django](#) repo, which I run with the current maintainer of Django Packages, is also worth a look.

## Learning Resources

As you become more comfortable with Django and web development in general, you'll find the [official Django documentation](#) and [source code](#) increasingly valuable. I refer to both on an almost daily basis. There is also the [official Django forum](#), a great albeit underutilized resource for Django-specific questions.

I run a dedicated website called [LearnDjango.com](#) that contains free and (soon) premium tutorials on Django. It is worth a look for additional information on how to use Django in the real world.

If you need help starting a new project quickly, take a look at my two free starter projects: [DjangoX](#) for Django and [DRFX](#) for a Django API.

If you're interested in a podcast on Django, I co-host [Django Chat](#), which features interviews with leading developers and topic deep-dives. And I co-write a weekly newsletter, [Django News](#), filled with news, articles, tutorials, and more about Django.

## Python Books

Django is, ultimately, just Python, so if your Python skills could improve, I recommend Eric Matthes's [Python Crash Course](#). For intermediate to advanced developers, [Fluent Python](#) and [Effective Python](#) are worthy of additional study.

## Feedback

If you purchased this book on Amazon, please leave an honest review. It makes an enormous impact on book sales and helps me continue to teach Django full-time.

As a final note, I'd love to hear your thoughts about the book. It is a constant work in progress, and the detailed feedback I receive from readers helps me continue to improve it. I try to respond to every email at [will@wsvincent.com](mailto:will@wsvincent.com).

Thank you for reading the book. Good luck on your journey with Django!