WILLIAM S. VINCENT

Django for Beginners

5th Edition

Build Modern Web Applications with Python

Django for Beginners (5th Edition)

Build Modern Web Applications with Python

William S. Vincent

Table of Contents

Foreword

Django bills itself as “the web framework for perfectionists with deadlines,” an ode to its origins as a newspaper content management system (CMS) and its overriding approach of balancing features with practicality. From the beginning, Django was created for real-world developers who wanted a fast, elegant, and powerful way to build websites. It is hard to appreciate, almost twenty years after its first release, just how different the World Wide Web landscape is from back then.

I first encountered Django in the early 2000s, shifting from a planned academic career to a self-taught web developer. Back then, Django wasn’t the Django we know today, with a vibrant community, regular community-run conferences, fantastic documentation, thousands of third-party packages, and all the rest. Instead, it was a plucky upstart released by early twenty-something kids from a Kansas newspaper. But it was immediately apparent that this was what I was looking for! For one thing, Django was written in Python rather than the PHP most web developers (myself included) used at the time. Python is a wonderfully readable, rich, and powerful programming language that has now won over much of the programming world, but at the time, it, too, was something of an upstart. Django also emphasized a “batteries-included” approach similar to Python, providing built-in solutions for most tasks while still allowing customization.

As my work with Django deepened, I became a co-maintainer of Django REST Framework and several other packages in the ecosystem. In 2018, I eventually became a Django Fellow. The Fellowship program is a paid position sponsored by the Django Software Foundation, the non-profit that maintains Django. A Fellow works on Django, managing releases, reviewing pull requests, and doing all the unglamorous but necessary things for an open-source project to succeed.

It was at a DjangoCon US event in 2018 that I first met the author of this book, Will Vincent. He gave a talk on Django API Authentication, and we started talking. We both agreed that a Django-focused podcast was needed and launched the first episode of Django Chat in early 2019; today, five years and 160+ episodes later, it is still going strong. During that time, Will was elected to the Board of the Django Software Foundation in 2020 and served as Treasurer for three years (2020-2023). Looking back now, we’re old hands, we’re veterans.

The first edition of this book came out in 2018 and immediately filled a void. As a Django Fellow, I was routinely asked for recommendations on how to learn Django, and *Django for Beginners* has long been my answer. It adopts Will's typically patient approach in explaining how things work while including practical examples of how to quickly build Django websites. Although you start building a "Hello, World" app, by the end of the book, you've written multiple real-world Django projects from scratch, layering on new concepts and techniques each time. And don't be fooled by the "Beginners" in the title; this book covers the fundamentals and essential areas such as testing, deployment, environment variables, security, and more. It is full of best-practice wisdom that jump-starts anyone wanting to learn more about Django, whatever their background.

I am thrilled for you to embark on your journey with Django. As your guide, this book will open your eyes to the myriad ways Django can revolutionize your web development experience. I hope your journey is as rewarding for you as it has been for countless others. *Django for Beginners* is the perfect gateway to the framework, the ecosystem around it, and our wonderful community. Have fun, and welcome aboard!

– Dr. Carlton Gibson

Django Fellow (2018-2023) and core contributor / Co-maintainer Django REST Framework

noumenal.es

Chapter 0: Introduction

Welcome to *Django for Beginners*, a project-based approach to learning web development with Django, a free and open-source web framework written in Python. Django is used by everyone, from students and startup founders to the largest websites in the world, including Instagram, YouTube, Reddit, Netflix, Dropbox, and Spotify. Its "batteries-included" approach provides all the built-in functionality you need to create powerful, real-world web applications quickly, hence its tagline, "The web framework for perfectionists on a deadline."

Django's abundance of features can feel overwhelming to newcomers. It doesn't help that the official polls tutorial and the official documentation are targeted at intermediate-to-advanced level web developers, not beginners.

The good news is that, as a "loosely coupled" framework, Django's components work independently or together, allowing for a high degree of modularity. In other words, you only have to use (and learn) what you need. Even professional developers with years of experience will only utilize some of what the framework offers; it's simply too big and too expansive for all its features to fit into a single project.

You will find, though, that the same patterns and tasks arise in almost every Django website: create and structure a new project, connect to and query a database, add logic, perform CRUD (Create-Read-Update-Delete) operations, handle user accounts and forms, and so on. Django does not have to feel overwhelming; indeed, it shouldn't! There is a built-in solution for almost every conceivable use case: that's what the documentation is for! But no one, even the original creators and core developers who wrote much of the documentation, can keep it all in their heads. You shouldn't attempt to either!

This book started as my personal notes on building Django projects. It took a long time before I had internalized and felt comfortable with Django's structure. The best way to solidify my understanding was to create progressively more complex projects focused on a new concept or skill. Eventually, I published my notes as a series of blog posts and, based on their popularity, created this book, now in its fifth edition.

In this book, you will learn how to build, test, and deploy six progressively more complex web applications. We will start with a "Hello, World" application and conclude with a real-world Newspaper website that ties together all the fundamental concepts and techniques covered in the book, including models, views, URLs, templates, forms, user accounts, permissions, and more. By the end of this book, you should feel confident creating Django projects from scratch and have the background to fill in any knowledge gaps with more advanced educational resources.

Why Learn Django?

Django was initially created in the fall of 2003 at the *Lawrence Journal-World* newspaper and named after the famous jazz guitarist Django Reinhardt; it was released as a free, open-source project in July 2005. That makes it almost twenty years old now, quite mature in software terms, but it has continued to thrive and is arguably more vibrant today than ever before. Each week, double-digit new code submissions are accepted into the framework, monthly security and bugfix releases, and a major new release every eight months. A vast ecosystem of third-party packages provides additional functionality beyond the core framework.

Django is written in the wonderfully readable yet powerful Python programming language, arguably the most popular language in the world today. Python is the default choice in most undergraduate computer science curriculums, the dominant language for data science and artificial intelligence, and widely used in scientific research. Its ease of use and broad applicability make Python suitable for almost any task.

Django inherited Python's "batteries-included" approach and includes a wide range of built-in features for routine tasks in web development, including:

- **ORM (Object-Relational Mapper)**: write Python rather than raw SQL for creating and querying database tables
- **Authentication**: a full-featured and secure system for user accounts, groups, permissions, and cookie-based user sessions
- **Templating Engine**: a simple syntax for adding variables and logic to create dynamic HTML
- **Forms**: a powerful form library that handles rendering and validation
- **URL Routing**: a clean, elegant URL schema that is easy to maintain and reason about

- **Admin Interface**: a visual way to interact with all website data, including users and database tables
- **Internationalization**: multilingual support plus locale-specific formatting of dates, time, numbers, and time zones
- **Security**: protection against SQL injection, cross-site scripting, cross-site request forgery, clickjacking, and remote code execution

This approach allows web developers to focus on what makes a web application unique rather than reinventing the wheel every time. Millions of users have already used and tested the necessary code, so you know it will be secure and performant.

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 Flask 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. Flask is a good choice for a web framework; it just has different strengths and weaknesses compared to a full-featured option like Django.

There is a saying among long-time Django developers, "Come for the framework, stay for the community." And it is true! Django has an unusually warm and welcoming community for all levels of programmer, represented in annual volunteer-run DjangoCon conferences across multiple continents, an active forum for discussion, and regular meetups in major cities. Unlike other open-source projects run by companies or individuals, Django is organized as a non-profit organization via the Django Software Foundation, whose goal is to promote, support, and advance the web framework. Its Board of Directors is voted on annually by the community.

Millions of programmers have already used Django to build their websites, and millions more turn to it each year because it doesn't make sense to reinvent the wheel when you can rely on a large community of brilliant developers who have already done the hard work for us.

Prerequisites

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. However, familiarity with basic Python, HTML, and CSS will go a long way toward solidifying your understanding of core concepts. 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.

What’s New in Django 5

Django 5.0 was released in December 2023 and has official support for Python 3.10, 3.11, and 3.12. It’s important to note that Django’s versioning policy is time-based rather than feature-based. Roughly every eight months, a new feature release occurs, along with monthly bug fixes and security patches as needed. Django also follows the pattern of .0, .1, .2, and then back to .0 for feature releases, meaning you can expect Django 5.1 in August 2024, Django 5.2 in April 2025, Django 6.0 in December 2025, and so on. Django has such a large and active community of contributors that the decision was made years ago to focus on regular rollouts rather than wait for specific features to be completed.

Specific releases (those that end in .2, like Django 5.2 and 6.2) are designated as long-term support (LTS) releases and receive security and data loss fixes applied for a guaranteed period, typically three years. This policy is designed for larger companies struggling to keep up with Django’s rapid release schedule. Still, the best security policy is to be on the latest possible release rather than an LTS version if you can.

So, what’s new in Django 5.0? The most significant change is form field rendering, which is now greatly simplified. Facet filters were added to the admin to allow for easier UI filtering, database-computed default values are now possible, and there is official support for Python 3.10, 3.11, and 3.12. Django has gradually added asynchronous support over the years, and this release adds a new async function to the `auth` module that controls user authentication. But perhaps the most noticeable change for developers upgrading to the latest edition is that logout links must now be `POST` rather than `GET` requests.

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 then learn about the powerful command line, Git, configuring text editors, and how to install the latest versions of Python and Django.

In **Chapter 2**, we review how websites and web frameworks work before diving into an overview of Django architecture. From there we build our first project, a minimal *Hello, World* website, while learning about views, URL, and apps. We even 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 *Personal Website* that introduces function-based views, templates, and the Django Templating Language. We explore the template context and write our first tests using Django's built-in testing framework.

Class-based views, template inheritance, and more advanced testing patterns are covered in **Chapter 4**, where we build a *Company Website*. This is the final project before we turn to Django models and database-backed websites.

We build our first database-backed project in **Chapter 5**, a *Message Board* website. 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 and store a remote copy on GitHub.

In **Chapters 6-8**, we're ready for a *Blog* website that implements CRUD (Create-Read-Update-Delete) functionality. Using first function-based views and then switching over to 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.

The remainder of the book is dedicated to building and deploying a production-ready *Newspaper* website. **Chapter 9** demonstrates setting up a new project using a custom user model and appropriate tests. **Chapter 10** covers a complete user authentication flow of login, logout, and signup, while **Chapter 11** adds Bootstrap for enhanced CSS styling. **Chapter 12** implements password reset and change via email and in **Chapters 13-15**, we add articles, comments, proper permissions, and authorizations. Finally, in **Chapter 16** 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 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, you are likely 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 for an answer is the only thing that separates them from junior programmers. There is some truth to this.

Of course, you can only trust some of what you read online. 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 mailing list, 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 correctly now will save you a lot of pain and heartache later.

You are probably eager to dive right in and start using Django, but configuring your computer now for Django development is a one-time task that will pay many dividends in the future. It is important to be comfortable with the command line and shell commands, understand how to use virtual environments, install the latest version of Python, use a text editor, and work with Git for version control. 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. Refrain from mindlessly copying and posting commands you find online; rely only on trusted resources.

In everyday usage, multiple terms 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:\Windows\System32>
```

Before the prompt is `PS`, which refers to PowerShell, the initial `C` directory of the Windows operating system, followed by the `Windows` directory and, within it, the `System32` directory. 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.

At this point, we need to navigate to the `users` directory, so enter the command `cd \users` followed by the `Enter` key to change directories (`cd`) into `users`.

Shell

```
PS C:\Windows\System32> cd \users
PS C:\Users>
```

On macOS, the built-in terminal is called *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%
```

Since 2019, the default shell for macOS has been zsh, which uses the `%` prompt. If you see `$` as your prompt, you are using the previous default macOS shell, Bash. While most of the commands in this book will work interchangeably, if your computer still uses `Bash`, it is recommended to look online at how to change to `zsh` via `System Preferences`.

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 not executed on the command line.

Shell

```
# Windows
$ whoami
wsv2024/wsv

# macOS
$ whoami
wsv
```

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

Shell

```
# Windows
$ pwd

Path
----
C:\Users

# 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. On Windows, you first need to change directories into your user directory, which is represented by the `whoami` command. On my computer, it is `wsv`, but yours will be different.

Shell

```
# Windows
$ cd wsv
$ cd onedrive\desktop
$ pwd

Path
----
C:\Users\wsv\onedrive\desktop

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

Tip: The `>` key on Windows and `tab` on macOS will autocomplete a command, so if you type `cd d` and then hit `> or tab`, the rest of the name will be filled in automatically.

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

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. As we've discussed, the `tab` command autocompletes the line. 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`.

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 select the result for Python 3.12 on the Microsoft Store. Click on the blue "Get" button to download it.

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

Shell

```
$ python --version
Python 3.12.3
```

The result should be at least Python 3.12. Then, type `python` to open the Python interpreter from the command-line shell.

Shell

```
$ python
Python 3.12.3 (v3.12.3:f6650f9ad7, Apr  9 2024, 08:18:47)
 [MSC v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You can exit the Python interpreter by typing either `exit()` or `Ctrl-Z` plus `Return`.

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 3.12. 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 3.12.3
```

Then, type `python3` to open the Python interpreter.

Shell

```
$ python3
Python 3.12.3 (v3.12.3:f6650f9ad7, Apr  9 2024, 08:18:47)
 [Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You can exit the Python interpreter by typing either `exit()` or `Ctrl-D` plus `Return`.

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!")`. Press the `Enter` or `Return` keys after each command 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 code. However, 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 versions of Python and Django is the correct approach for any new project. However, in the real world, it is common for existing projects to rely on older versions of each. Consider the following situation: *Project A* uses Django 4.0, but *Project B* uses Django 5.0. 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. 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, but it will appear empty on macOS. 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. The directory also contains a copy of the Python interpreter and a few management scripts, but you will not need to use them directly in this book.

Once created, a virtual environment must be *activated*. On Windows, there is a one-time additional step of setting an *Execution Policy* to enable running scripts. The Execution Policy tells Windows, 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 changes to Python packages (such as updating existing ones or installing new ones) will occur *only* within our 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 3.7 and 3.12 installed on your computer, it is possible for `pip install` to use Python 3.7 at one point but Python 3.12 later: not desired behavior. Brett Cannon has a much fuller explanation if you are curious about the underlying reasons why this is the case.

Install Django

Now that we have learned how to install Python properly, use virtual environments, and update `pip` to the latest version, it is time to install Django for the first time.

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~=5.0.0

# macOS
$ source .venv/bin/activate
(.venv) $ python3 -m pip install django~=5.0.0
```

This command uses the comparison operator, `~=`, to install the latest version of Django 5.0.x. As I type these words, the newest version is `5.0.4`, but soon it will be `5.0.5` and then a month later `5.0.6`. By using `~=5.0.0`, we ensure that the latest version of `5.0.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. 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: If they differ, I will provide separate Windows and macOS commands. However, when 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` without a period at the end, 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
```

Do you see the multiple `django_project` directories? First, there is a top-level `django_project` directory, and within it *another* one containing the files we need for our Django project. Opinions differ on the "best" approach within the Django community, but having these two directories with the same name feels redundant. Deployment is also somewhat simpler with only one `django_project` directory, 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
```

As you progress in your journey learning Django, you will encounter more situations like this, where the Django community has different opinions on the correct best practices. 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, decide, and stick with them!

The Development Server

Django includes a built-in, lightweight local development web server accessible via the runserver command. The development server automatically reloads Python code for each request and serves static files. However, some actions, such as adding files, will not automatically trigger a restart, so if your code is not working as expected, a manual restart is always a good first step.

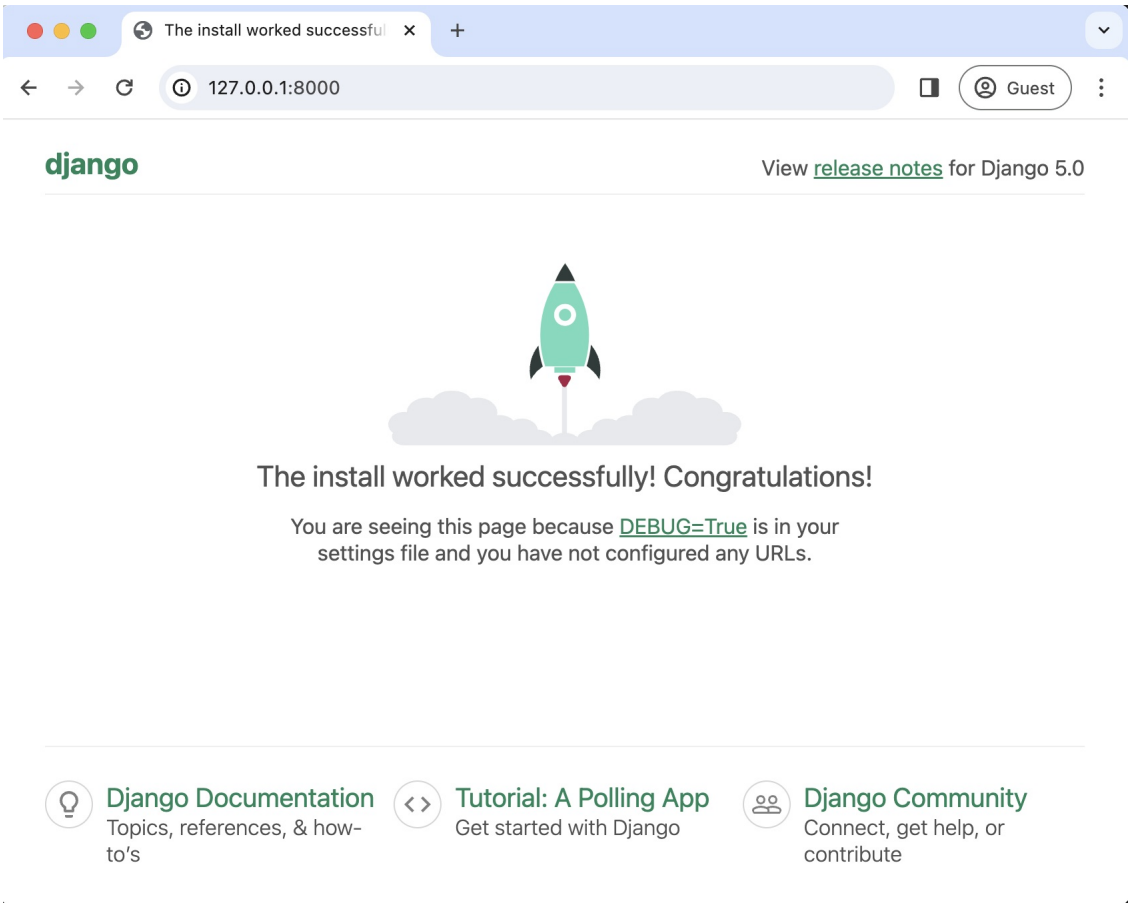
By default, the server runs on port 8000 on the IP address 127.0.0.1, which is known as the "loopback address" because no data is sent from our computer (host) to the local network or internet; instead, it is "looped back" on itself so the computer sending the data becomes the recipient.

Let's confirm everything is working correctly by starting the local development server now. We'll use `manage.py` to execute the `runserver` management 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.
June 28, 2024 - 16:43:31
Django version 5.0.6, using settings 'django_project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

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



Django Welcome Page

Note: On Windows, 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.

For readers new to web development, it is worth mentioning that `localhost` is a common shorthand for `127.0.0.1`, so the URL addresses `http://127.0.0.1:8000/` and `http://localhost:8000/` are functionally equivalent. 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 acceptable.

If you look at your files and folders, you will notice a new `db.sqlite3` file has been created. SQLite automatically creates a new file the first time you try to connect if one doesn't already exist.

Layout

```
├── django_project
│   ├── __init__.py
```

```
|   ├── asgi.py
|   ├── settings.py
|   ├── urls.py
|   └── wsgi.py
├── db.sqlite3  # new
├── manage.py
```

Go ahead and stop the local server with `Control + c`. Then, exit the virtual environment by typing `deactivate` and then `Return`.

Shell

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

This book provides lots of practice with virtual environments, so don't worry if they are still confusing. The basic pattern for any new Django project is to create 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 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 code is 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, and they come with helpful extensions to make Python and Django development more accessible. Two of the more popular options are PyCharm and Visual Studio Code. PyCharm has a paid Professional and free Community version, while VSCode is free. Ultimately, it does not matter what text editor you choose: 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" and the official Microsoft extension will be the first result. Install it.

The second is adding Black, a Python code formatter quickly becoming 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.

Shell

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

Next, within VSCode, open the settings by navigating to File -> Preferences -> Settings on Windows or Code -> Preferences -> Settings on macOS. Search for "default formatter". Under "Editor: Default Formatter" select "Black Formatter" from the drop down list.

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 and final configuration allows you to open VSCode directly from your terminal. This technique is useful since a standard workflow opens the terminal, navigates to the code directory you want to work on, and opens it with your text editor.

To enable this functionality, press `Command + Shift + P` simultaneously within VSCode to open the command palette, which allows us to customize our VS Code settings. Then, 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. A success message will appear: “Shell command ‘code’ successfully installed in PATH.” The PATH variable, by the way, is often used to customize terminal prompts.

Return 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 code version, 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 sufficient and can 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.” 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 that Git is installed on Windows, close all current shell windows and open a new one to load the changes to our PATH variable. (PATH is an environment variable that specifies where executable programs are located. In other words, when you type `git`, where does your command line look?) Type in `git --version` to display the installed version of Git.

Shell

```
# Windows
$ git --version
git version 2.45.2.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
git version 2.45.2
```

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. If you do not see the message, type `xcode-select --install` instead to install Xcode directly.

Be aware that Xcode is a large package, so the initial download may take some time. Xcode is primarily designed for building iOS apps but 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 installation worked.

Shell

```
# macOS
$ git --version
git version 2.45.2
```

Once Git installs on your local machine, we must 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
```

You can always change these configs later by retyping the same commands with a new name or email address.

Conclusion

Configuring a software development environment from scratch is challenging. Even experienced programmers have difficulty with the task, but it is a one-time pain that is more than worth it. We can now start 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 Website

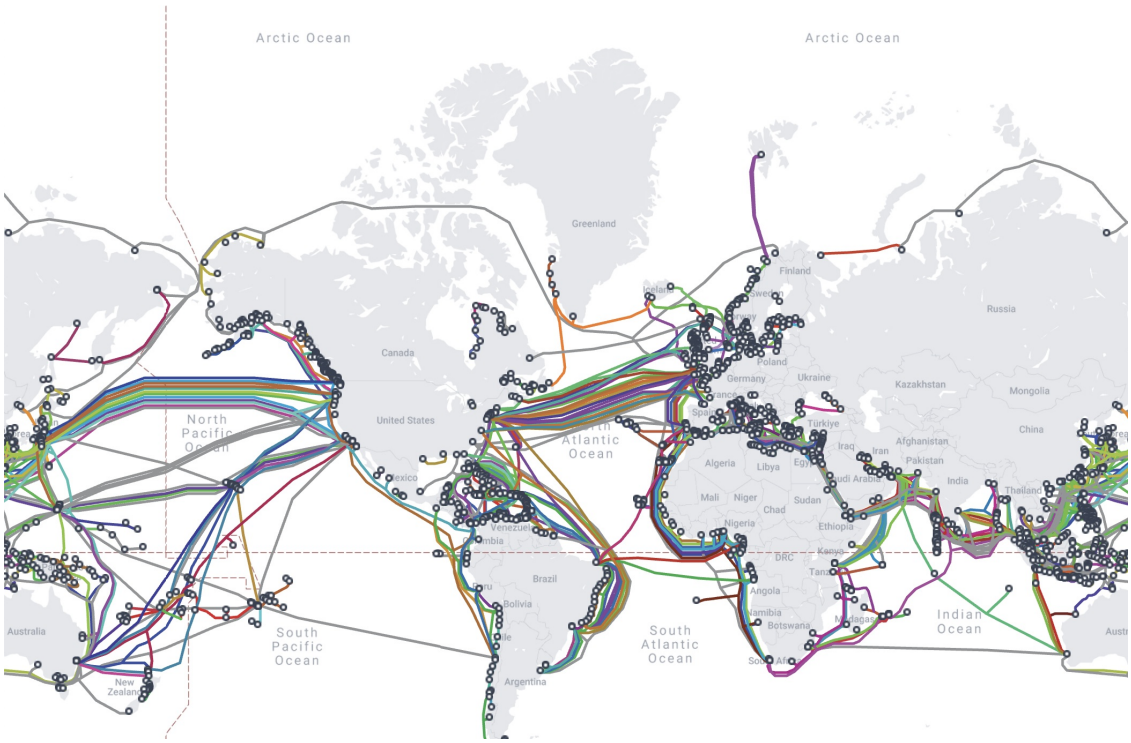
In this chapter, we will review how websites and web frameworks work, examine Django's architecture, and build a simple Django website that displays "Hello, World." We will also examine two fundamental parts of Django, URL mappers, and views, and use Git for the first time. The complete source code for this and all future chapters is available online at the official GitHub repo for the book.

How the Internet Works

You probably use the Internet every day, but unless you are a web developer, you are unlikely to know the full details of what happens when you type an address like `https://learndjango.com` into your browser and press the `return` key. Underpinning this user experience is a complex network of communication protocols, web servers, and logic.

Computer science students typically take an entire course devoted to Network Communication, and there are network engineers who focus their careers on this one area. For our purposes as web developers, a general understanding is sufficient to get started, but if you eventually work on high-traffic websites, the details and nuance of network communication increasingly matter.

Underpinning the Internet is a network of connected machines called *servers*. These are special computers that don't have a screen, a mouse, or a keyboard and typically live in data centers amongst racks and racks of other servers. In the old days, web developers had to run their own physical servers, but these days, it is far more common to rent space from a large hosting company instead, known colloquially as "the cloud." One way to visualize this web of connected machines is via the Submarine Cable Map, an interactive display of the underwater cables connecting continents and countries.



Submarine Cable Map

Let's trace what happens when you try to visit the website, `https://learndjango.com`. We can simplify it into six steps:

1. You enter a domain name into a browser
2. The browser looks up the IP address for the domain name via DNS (Domain Name System)
3. The browser establishes a network connection to the web server
4. The browser sends an HTTP *request* for the desired resource (e.g., the homepage)
5. The website processes the request (more on this below) and returns an HTTP *response*
6. The browser begins rendering the webpage

HTTP (Hypertext Transfer Protocol) is the set of rules computers use to communicate with one another over the Internet to power websites. It was created by Tim Berners-Lee, the inventor of the World Wide Web, who also created the HTML markup language and the URL system. Once an HTTP request has been received, the web browser sets about rendering the web page using HTML. Any additional resources required for a webpage or additional

webpages undergo the same HTTP request and HTTP response cycle until a user leaves the website and the network connection is closed.

When you request a website like `learndjango.com`, your web browser first asks the DNS (Domain Name System) to translate the domain name into an IP (Internet Protocol) address, a unique numerical identifier computers use to find one another across the Internet. Once the web browser knows the IP address, it establishes a network connection with the server at that IP address containing the contents of the desired website. The browser sends an HTTP *request* for the desired resource (e.g., the homepage) and the server returns an HTTP *response* with its contents.

How Web Frameworks Work

Websites come in two basic categories: static and dynamic. A static website consists of individual HTML documents where if your website has ten pages, you need ten individual HTML documents that can be served. This approach only works for very small sites. Most websites are dynamic, consisting of a database, HTML templates, and an application server that can generate files before sending them to your browser. With a dynamic website, a relatively small amount of code can generate hundreds or even thousands of web pages. Web frameworks like Django are designed for dynamic websites.

In the early days of the World Wide Web, developers had to hand code all the pieces of a dynamic website themselves, which was error-prone, often insecure, and not performant. Web frameworks like Django soon emerged to standardize this process. Web developers realized many tasks were routine and would be better performed and monitored by a community rather than individuals.

At its core, a web framework like Django has three main tasks:

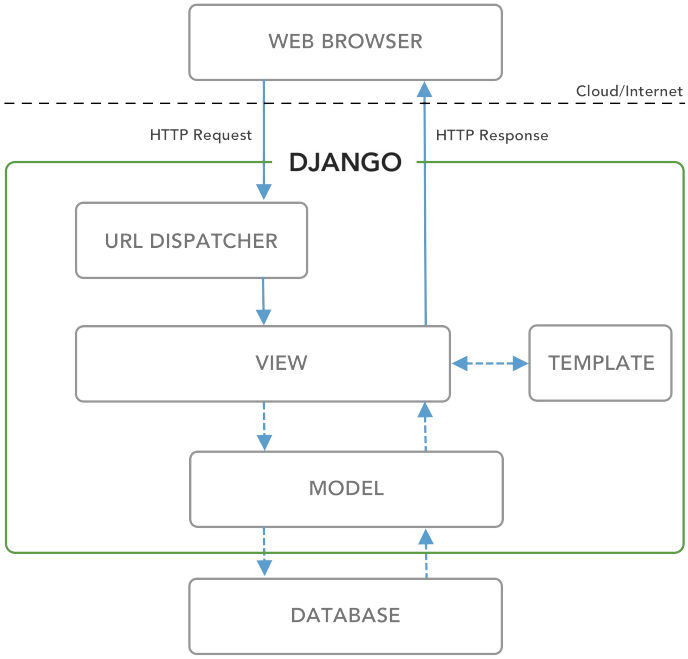
1. Map URLs to view logic for rendering pages
2. Provide an abstraction layer for interacting with a database
3. Display HTML-like code via a templating system

That’s it! While web frameworks are written in different programming languages and have slightly different design philosophies, their overall goal is essentially the same.

Django Architecture

Now that we have reviewed how websites and web frameworks work let's examine Django's architecture. There are four main components to consider: URLs, views, models, and templates.

Visually, the Django request and response cycle looks as follows where the solid lines represent required interactions and the dotted lines optional ones.



Django Architecture

When an HTTP Request comes in from a Web Browser the first part of Django it engages with is the URL Dispatcher (`urls.py` file), which searches through configured URL patterns and stops at the first matching View (`views.py` file). The View assembles the requested data and styling before generating an HTTP Response back to the web browser. Technically, this is all we need. It is possible to have a Django website with just a URL Dispatcher and a View, as we will see later in the chapter.

It is more common, however, to have two more components involved: Model and Template. For a database-backed website, the View will interact with the Model (`models.py` file), which defines database tables, behaviors, and supports queries from the Database. This data is then sent back to the View, which in most cases then sends it to a Template for rendering. The Template is primarily an HTML file but can be in any text-based format, including XML and JSON. Once the View has all the necessary information, it returns an HTTP Response to the Web Browser.

This Django request/response cycle repeats for each new HTTP Request made by the Web Browser.

Model-View-Controller vs Model-View-Template

If you have built websites before, you might be familiar with the **Model-View-Controller (MVC)** pattern used by many web frameworks, including Ruby on Rails, Spring (Java), Laravel (PHP), and ASP.NET (C#). This is a popular way to internally separate an application's data and logic and display it into separate components that are easier for developers 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 is more accurately a 4-part pattern incorporating URL configuration, **Model-View-Template-URL (MVTU)**:

- 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 dispatcher."

If you are new to web development, the distinction between MVC and MVT will not matter much: this book demonstrates Django's way of doing things. 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," which is more loosely coupled and allows for easier modifications than the MVC approach.

Initial Set Up

For our first Django website, we will build a "Hello, World" website as simply as possible. Although most Django websites have a URL dispatcher, views, model, and template, technically, we only need a URL dispatcher and views. That's what we'll use here, but subsequent chapters will introduce both templates and models.

To begin, open up a new command line shell or use the built-in terminal on VS Code. For the latter, click "Terminal" at the top and then "New Terminal" to bring it up at the bottom of the VS Code interface.

Make sure you are not in an existing virtual environment by checking that nothing is in parentheses before your command line prompt. You can even type `deactivate` to be entirely sure. Then, navigate to your desktop's `code` directory 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~=5.0.0
(.venv) $ python -m pip install black

# macOS
```

```
$ python3 -m venv .venv
$ source .venv/bin/activate
(.venv) $ python3 -m pip install django~=5.0.0
(.venv) $ python3 -m pip install black
```

Now, we'll use the Django `startproject` command to create a new project called `django_project`. Remember to include the period (`.`) at the end of the command so that the project is installed in our current directory.

Shell

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

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

Code

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

Django has created a `django_project` directory and a `manage.py` file. Within `django_project`, there 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 do often in Django!
- `asgi.py` configures an optional ASGI (Asynchronous Server Gateway Interface) application
- `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` configures a WSGI (Web Server Gateway Interface) application, the default setting for Django

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

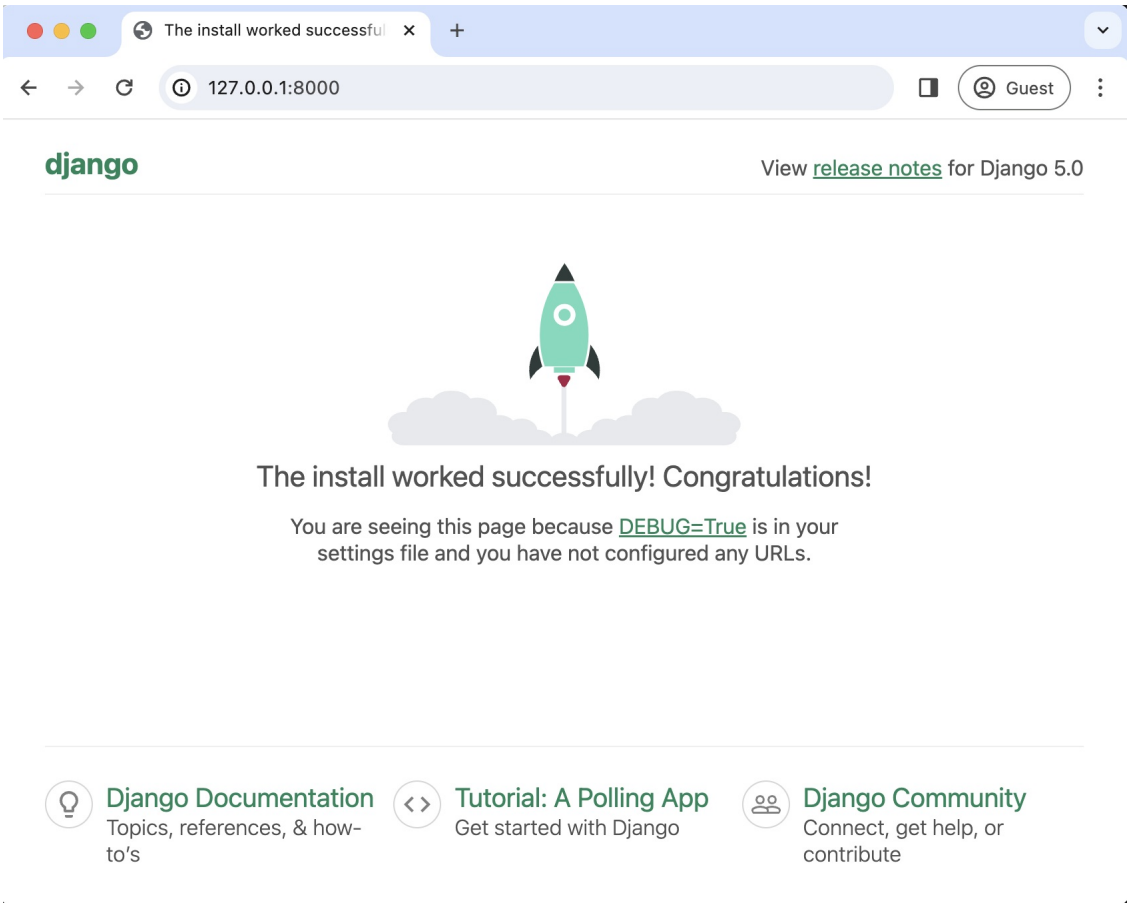
a new app.

Let’s try out our new project by executing `python manage.py runserver` to launch Django’s built-in web server. This server is suitable for development but not production. We will look deeper at production setups when we deploy websites later in the book.

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 th\
e migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
June 28, 2024 - 16:43:31
Django version 5.0.6, using settings 'django_project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

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



Django Welcome Page

Notice that a `db.sqlite3` file has been created in your project directory since we attempted to connect to SQLite for the first time. At the moment, it is empty.

Code

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

Migrations

On the command line, we still see that warning about `18 unapplied migrations`; let's explore now what is occurring. Migrations are special scripts Django creates automatically to track changes to the database. As a project

grows over time, there are often many changes to the Django database models that define the structure of a database and all its tables. The Django migrations framework allows developers to track changes over time and change the database to match the configurations within a specific migrations file.

When you start a new project using the `startproject` command, Django includes several built-in apps (more on what an app is shortly) that make changes to the database, including `admin`, `auth`, `contenttypes`, and `sessions`. We can apply these changes to the local database using the management command, `migrate`. Type `Control + c` first to stop the local server.

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
```

The `migrate` command applies all available migrations and lists them: `Apply all migrations: admin, auth, contenttypes, sessions`. The output includes the full name of each app and its migration script. For example, `Applying contenttypes.0001_initial... OK` indicates that migration script `0001_initial` in the `contenttypes` app was run successfully.

Restart the development server, and there will no longer be any warnings.

Shell

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

```
System check identified no issues (0 silenced).
June 28, 2024 - 16:43:31
Django version 5.0.6, using settings 'django_project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

The `db.sqlite3` file is now populated with built-in tables and data from Django. If you install the free SQLite viewer extension, SQLite Viewer, you can visually inspect them.

We will cover databases in more depth later in the book. At this stage, the important point is that Django uses migration files to control changes to the database, and the `migrate` management command applies them.

Create An App

A Django project can contain many "apps," an organizational technique for keeping our code clean and readable. Each app should control an isolated piece of functionality. If you take a look at the `django_project/settings.py` file, there are already six built-in apps Django has provided for us. They are located in the `django.contrib` directory and control functionality for the admin, auth, contenttypes, sessions, messages, and staticfiles. You don't need to know what each one does at this point.

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",
]
```

Nothing is forcing the app convention–you could write all your code in a single file, if desired–but this convention of separating logic makes it much easier to structure and reason about a Django project. We use apps when we want to add functionality to the Django project. For example, an e-commerce site might have one app for user authentication, another for payments, and a third to power item listing details. 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 separate apps, each with a single function.

To create a new app, go to the command line and quit the running server using `Control+c`. Then, use the `startapp` command followed by our app's name. In this example, we will use the name "pages." A best practice in Django is for app names to be plural–pages, payments, etc.–unless doing so does not make sense, such as for a "blog" app.

Shell

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

If you look visually at the `django_project` directory, Django has created within it a new `pages` directory containing the following app 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

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 and scroll down to `INSTALLED_APPS`, where you'll see six built-in Django apps. 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

We'll create a static page for our first website that outputs the text "Hello, World!" This page does not involve a database or even a template 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 almost anything. When a web page is requested, Django automatically creates a `HttpRequest` object that contains metadata about the request. Then Django loads the appropriate view, passing `HttpRequest` as the first parameter 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.
```

We will take a proper look at `render` in the next chapter, but for now, 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 HttpResponse("Hello, World!")
```

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

- Import the class `HttpResponse` from the `django.http` module.

- Define a function called `home_page_view`. In Python, it is customary to use `snake_case`–all lowercase words separated by underscores–for function and variable names.
- The first parameter passed into a view is the `HttpRequest` object. It is a convention to name it `request` for readability, but the order is what matters, not the name, so you could technically call it `req` or any other name and it would still work.
- The view returns an `Http Response` object with the string of text "Hello, World!"

All views work this way: first, you define a name for the view, name the `HttpRequest` object (`request` in this case), and then return something. Adding more logic and parameters to views is possible, but the general pattern is the same.

URL Dispatcher

With our view in place, it is time to configure a related URL. In your text editor, create a new file called `urls.py` within the `pages` app and 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),
]
```

On the top line, we import `path` from Django to power our URL pattern. 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 named `home_page_view`. Note the lack of a leading slash `/` in the `path` pattern. Django automatically prefixes the leading slash for us.

Our URL pattern here has two parts:

- the route itself, here defined by the empty string, `""`
- a reference to the view `home_page_view`

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, the gateway to other URL patterns distinct from each app. This architectural pattern will make more sense as we build increasingly complex web applications later in the book.

Django automatically imports and sets a path for the built-in admin. To include additional URL paths, we import the function `include` from the `django.urls` module, and then set its path. In this case, we again use the empty string, `""`, and include *all* URLs contained in the `pages` app. The Django docs have a fuller description of how this process works under the hood if you are hungry for more details at this stage.

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
]
```

Now, whenever a user visits the homepage, represented by the empty string here, `""`, Django will look within the `pages` app for matching URL routes.

We have all the code we need. 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 will now display the text "Hello, World!"

Hello World Homepage

And that's it! We've created a Django project from scratch and had our first interactions with Django's architecture and typical pattern:

- Creating a project and then an app within it
- Writing a view
- Connecting the view to a URL dispatcher

It is possible to be clever and write a Django "Hello, World" app in fewer lines of code and even in a single file: see the repository [django-microframework] if you're curious to learn more. But it's more important to start internalizing typical Django project structure and patterns at this stage.

Git

In the previous chapter, we installed the version control system Git. Let's use it here. The first step is initializing (or adding) 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 type `git status` you'll see a list of changes since the last Git commit. Since this is our first commit, this list includes the contents of the entire directory.

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 the virtual environment, `.venv`, is included. It is a best practice not to include your virtual environment in Git source control because it can significantly increase the size of the repo and is less portable than having a `requirements.txt` file instead (which we will implement shortly).

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 its 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
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── models.py
│   ├── tests.py
│   ├── urls.py
│   └── views.py
├── .gitignore  # new
├── db.sqlite3
├── manage.py
```

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 no longer there. It has been "ignored" by Git. 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. However, optimization is not relevant to learning projects such as this.

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 nine packages even though we have installed only two: Django and Black. That's because Django and Black also depend on *other* packages. It is often the case that when you install one Python package, you're also installing multiple dependent packages. Since keeping track of all the packages is difficult, a `requirements.txt` file is critical.

requirements.txt

```
asgiref==3.8.1
black==24.4.2
click==8.1.7
Django==5.0.6
mypy-extensions==1.0.0
packaging==24.1
pathspec==0.12.1
platformdirs==4.2.2
sqlparse==0.5.0
```

Next, we want to perform our first Git commit to store all the recent changes. Git has a lengthy list of options/flags. For example, to add *all* recent changes, we can use `git add -A`. Then, to `commit` the changes, we will use a `-m flag` ("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 debug efforts later since you can search through your commit history.

Shell

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

Conclusion

Congratulations! This chapter covered a lot of material, starting with how the Internet and web frameworks work and progressing on to Django's architecture. We then built our first Django website while learning about apps, views, URLs, and the internal Django web server. We also worked with Git to track our changes, create a `.gitignore` file, and generate a `requirements.txt` file.

If you become stuck, compare your code against the official repo.

Please continue to the next chapter, where we'll build a more complex Django application using templates and more advanced function-based views while also incorporating testing.

Chapter 3: Personal Website

In this chapter, we will build a *Personal Website* containing a Homepage and an About page while learning more about Django's templates, function-based views, and testing. Templates are the presentation layer that controls how data is displayed, but they also allow for inheritance and basic logic consistent with Django's design philosophy of Don't Repeat Yourself (DRY). Views combine URLs and templates while adding quite a bit of logic to the data; they are often considered the logic layer in a Django app. The third fundamental concept is testing, which is vital to any web project and well-supported in Django. By the end of this chapter, you will have learned how to work with simple function-based views and templates and write your first tests.

Initial Set Up

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

- make a new directory for our code called `personal_website` 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`
- make a new app called `pages`

Within a new command line shell, navigate to the `code` folder on the desktop and create a new folder called `personal_website`. You should not see an active virtual environment, represented by `(.venv)`, before the command line prompt of `>` on Windows or `%` on macOS. If you do, you're still in an existing virtual environment, so type `deactivate` to leave it. Change directories into `personal_website`, create a new virtual environment, and activate it.

Shell

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

```
# macOS
$ cd ~/desktop/code
$ mkdir personal_website
$ cd personal_website
$ 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`. These commands will become very familiar by the end of this book, as the only thing that changes is the name of your project or app.

Shell

```
(.venv) $ python -m pip install django~=5.0.0 black
(.venv) $ django-admin startproject django_project .
(.venv) $ python manage.py startapp pages
```

Even though we added a new app, Django will not recognize it until we update 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
```

Then, navigate to `http://127.0.0.1:8000/` to see the Django welcome page. With only a few commands, we have a fresh Django project running within a virtual environment, Black installed, and a `pages` app.

Homepage

A Django “view” is a Python function that accepts a Web request and returns a Web response. When a Web page is requested, Django automatically creates an `HttpRequest` object with metadata about the request. The view returns an `HttpResponse` object.

We will begin by repeating the steps from the previous chapter to create a homepage using just a view and a URL dispatcher. Then, we will create an About page using a template and more complex view logic.

Let’s start with the view. Django prepopulates the `views.py` file within an app with the following code:

Code

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

# Create your views here.
```

We will use `render` for the About page, so leaving that import as is is fine. For our home page, we will repeat the steps from the previous chapter: import the class `HttpResponse`, create a view named `home_page_view`, name the first parameter (which is always the `HttpRequest` object!) as `request`, and then return the text “Homepage.”

Code

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


def home_page_view(request):
    return HttpResponse("Homepage")
```

The next step is creating a `urls.py` file within the `pages` app that imports `path` from `django.urls`, `home_page_view` from the `views.py` file in the local directory, and sets a route at the empty string, `""`, that calls `home_page_view`.

Code

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

from .views import home_page_view

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

The final step is updating the project-level `urls.py` file, the initial entry point for all URL requests. We import the `include` function to include other URL configurations and set a URL route of `""` for the `pages` application.

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
]
```

That's it! If the local server is still running on the command line, you should be able to visit the home page in your browser:

Homepage

Function-Based View About Page

render() is a Django shortcut function useful for working with a template. The first parameter in `render()` is the request object, and the second parameter is a template name. That means if we want to create a view for an about page using a template called `about.html` we can write it like this:

Code

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


def home_page_view(request):
    return HttpResponse("Homepage")


def about_page_view(request):  # new
    return render(request, "pages/about.html")
```

The new view is named `about_page_view` and has as its first parameter the `HttpRequest` object, which we've named `request`. It uses the `render()` function to return the request object and a related template named "pages/about.html."

Templates

Templates are the presentation layer in Django, providing a convenient way to generate HTML files that can *also* include CSS, JavaScript, and media such as images. By default, Django's template loading engine looks for a "templates" subdirectory in each app. That means we *could* add a `templates` directory within the `pages` app and then include the `about.html` in the following way:

Layout

```
└── pages
 ├── templates
  ├── about.html
```

This approach works, but it is not a best practice. Django chooses the first template it finds whose name matches. What happens if there are `about.html` files within two separate apps? Django cannot easily tell which one should be used in our function. Fortunately, a simple fix is to *namespace* template files by placing them within another directory containing the application name. In other words: app -> templates -> app -> template file.

In our `pages` app, that would look like this:

Layout

```
└── pages
 ├── templates
  ├── pages
   ├── about.html
```

As a Django best practice, you should always adopt this approach when storing template files within an app. You can create these new directories within your text editor or directly on the command line:

Shell

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

Then, add a new file called `about.html` within the `pages/templates/pages` directory. You can do this in Visual Studio Code by navigating with your mouse

to the top left of your screen, clicking “File,” and then “New File.” Make sure to name and save the file in the correct location.

The `about.html` file will have a <h1> tag for a headline and a `<p>` tag for paragraph text.

Code

```
<!-- pages/templates/pages/about.html -->
<h1>About page</h1>
<p>This is the new template-powered About page.</p>
```

Our template is complete! The view, `about_page_view`, can access the template when called, but the remaining step is configuring a URL dispatcher.

URL Dispatcher

At this point, the pattern for adding a new URL route should start to feel familiar. We import our view, `about_page_view`, and then set its path, `about/`, and the view name.

Code

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

from .views import home_page_view, about_page_view  # new

urlpatterns = [
    path("about/", about_page_view),  # new
    path("", home_page_view),
]
```

And that’s it! Start up the development web server using the `runserver` command.

Shell

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

If you navigate to `http://127.0.0.1:8000/about/`, the About page will be visible.

FBV About Page

The Django Template Language

All web frameworks (including Django) need a way to generate HTML dynamically, and the most common approach is to use templates that contain static content and a special syntax for inserting dynamic content. Django has its own templating language that is not attempting to replace dedicated JavaScript front-ends like Vue, React, or Angular. Rather, it is a deliberately limited–but still powerful–way to use comments, variables, filters, tags, and template inheritance.

Let's revisit our `about.html` template to try out some of these features. We'll begin with comment tags that allow us to add code that is readable to the developer but not displayed on the page. Anything between `{% comment %}` and `{% endcomment %}` tags will be ignored.

Code

```
<!-- pages/templates/pages/about.html -->
<h1>About page</h1>
<p>This is the new template-powered About page.</p>
{% comment %}This is just a comment. It won't appear on the web page!{% endcomment %}
```

If you save the file and refresh the web browser, the comment will not be visible. It is not rendered on the template HTML sent to the `about_page_view` and then the web browser. If you want to add multi-line comments, the comment tag can be used, but broadly speaking, Django templates should not be complicated. While it is OK to add basic logic to them, it is better to move the logistical heavy thinking to views or model files (which we will do later in the book).

Template Context

Whenever a Django template is rendered, a context is also created. This context is a dictionary-like object with variable names for keys and variable values as values. We can and will update this context with information, often from a database. When you render a template with a given context, every key in the context dictionary becomes a variable in the template that you can access and use.

The render() shortcut function expects parameters in the following order:

1. The `HttpRequest` object conventionally named `request`
2. The `template_name`
3. The `context` dictionary

In other words: `render(request, template_name, context)`. By default, `context` is set to `None`, but we can manually add information to see it in action.

Code

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


def home_page_view(request):
    return HttpResponse("Homepage")


def about_page_view(request):
    context = {"name": "Alice"}  # new
    return render(request, "pages/about.html", context)  # new
```

Here, we've added a variable, `context`, containing a dictionary with the key variable of "name" and a value variable of "Alice." To display the context in our template, the syntax for variables is to surround the variable name with brackets `{}`. Since the key value is "name," we type `{{ name }}` to make it visible.

Code

```
<!-- pages/templates/pages/about.html -->
<h1>About page</h1>
<p>My name is {{ name }}.</p>
```

If you refresh your browser, it will now show the name "Alice."

About Page with Name

As an exercise, swap out the name “Alice” with your name in the `pages/views.py` file. Then, refresh the web browser to see if it is now visible on the web page.

We can also add other key/value pairs to the context. For example, let’s add an “age.”

Code

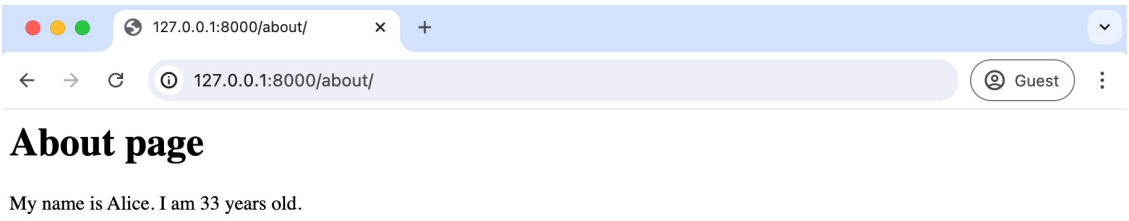
```
# pages/views.py
from django.http import HttpResponse
from django.shortcuts import render


def home_page_view(request):
    return HttpResponse("Homepage")


def about_page_view(request):
    context = {
        "name": "Alice",
        "age": 33,  # new
    }
    return render(request, "pages/about.html", context)
```

To display the age on our Web page, we can display it as a variable in the template using the syntax `{{ age }}`. Note that `name` is a string since it is in quotations, whereas `33` is an integer. Setting data types is important in Python code and Django applications, which we will explore shortly.

Code

```
<!-- pages/templates/pages/about.html -->
<h1>About page</h1>
<p>My name is {{ name }}. I am {{ age }} years old.</p>
```

Refresh the About page again to see it update.



About Page with Name and Age

In the `pages/views.py` file, update the `age` variable with your own age and refresh the web page to see the updated result. Anything we pass as a variable to our template context can be rendered on our web page.

The Django template language also comes with over eighty built-in tags and built-in filters that provide additional functionality. We will explore them in greater depth later in the book.

Tests

Writing tests for your code is as important as writing the code itself. Without accompanying tests, you can't submit new code changes to an open-source project or within any well-structured company. The reasoning is simple: while your code change may be small, there is no telling what else it might inadvertently break in the project. Writing good tests provides confidence in your codebase, doesn't take too long, and can be automated to run automatically on any new code changes. 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 create 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, there's no telling what can change.

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 test database transactions directly while `LiveServerTestCase` launches a live server thread for testing with browser-based tools like Selenium.

Note: You may have noticed that 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 uses 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, return HTTP 200 status codes, 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 not provided automatically, but that causes a 301 redirect, not a 200 success response!

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
```

We learned in the last chapter that by creating a `.gitignore` file, we can instruct Git to ignore certain files and directories. We will include the `.venv` directory containing our virtual environments. But we will also ignore the `__pycache__` directory, which contains bytecode compiled and ready to be executed, and the database itself, `db.sqlite3`. In addition to being quite large, a file-based database included in source control poses a security risk. It is common to have separate levels of access to a project amongst a large team. Often, every developer can view the source code since any mistakes can be quickly rolled back and changed with Git. Rolling back changes to a production database is considerably more difficult, and most developers don't need access to it so long as they have a local or testing database available. Thus, it is a best practice to keep your database access separate from your source code, even with file-based

databases like SQLite, so we will do that here. Create a new `.gitignore` file and add the following three lines:

.gitignore

```
.venv/
__pycache__/
db.sqlite3
```

Run `git status` again to confirm these three files are being ignored. We want a record of installed packages in our virtual environment, which we can do by creating a `requirements.txt` file.

Shell

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

Run `git status` one final time to confirm that the newly created `requirements.txt` file is visible. Then, add all intended files and 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 “personal-website” 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/personal-website.git
(.venv) $ git branch -M main
(.venv) $ git push -u origin main
```

Conclusion

Congratulations on building your second Django website. The complete source code for this chapter is available on GitHub if you need a reference. Although

the website is quite basic, we have explored several fundamental concepts, including function-based views, templates, URL dispatchers, and testing. We also used Git again, created `requirements.txt` and `.gitignore` files, and pushed our code to GitHub for the first time. With this foundation in place, we can move more quickly to the next chapter, where we will build a Company website using class-based views, template inheritance, and even more testing.

Chapter 4: Company Website

In this chapter, we will build our third project, a Company Website, while learning more about templates, introducing class-based views, and integrating more advanced testing. This is the final project before we turn to the database and Django models, so it is a chance to reinforce our past learnings and explore what the three other parts of Django–views, URLs, and templates–can do.

Initial Set Up

Our initial setup should start to feel familiar now and contains the following steps:

- make a new directory for our code called `company` 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 `company`, change directories into it, and activate a new Python virtual environment called `.venv`.

Shell

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

# macOS
$ cd ~/desktop/code
$ mkdir company
$ cd company
$ python3 -m venv .venv
```

```
$ source .venv/bin/activate
(.venv) $
```

Next, install Django and Black, create a new project called `django_project`, and create a new app called `pages`. We've called all our apps "pages" because they have been used for relatively static pages. In future projects, we will populate our pages from the database, and the app names will reflect that new dynamic.

Shell

```
(.venv) $ python -m pip install django~=5.0.0
(.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 explicitly 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
```

Then navigate to `http://127.0.0.1:8000/` to see the Django welcome page.

Project-Level Templates

We previously saw that Django expects template files to be located within an app in a directory called `templates` and that it was a best practice to namespace this

further by adding the directory name again. In other words, template files for a `pages` app would be located in a directory at `pages/templates/pages/`.

However, many Django developers favor another approach: creating a single project-level `templates` directory and placing *all* templates within it. This makes finding and updating all the templates in one location easier. By tweaking our `django_project/settings.py` file, we can also tell Django to look in this directory for templates.

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 where our new templates directory is. This requires a one-line change to the `"DIRS"` configuration under `TEMPLATES`.

Code

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

We'll use that approach to organizing templates for the rest of the book. 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.

For now, the `home.html` file will have a simple headline.

Code

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

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

Function-Based View and URL

It is entirely up to the developer whether to write the view first or the URLs. Ultimately, we need both to display our web page, so deciding the execution order becomes a personal preference over time. In this instance, we will start with the view in the `pages` app.

Code

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


def home_page_view(request):  # new
    return render(request, "home.html")
```

This code should look familiar from the previous chapter. We use the `render()` shortcut function, which is imported at the top. Then we create our view, `home_page_view`, and name its first parameter–the `HttpRequest` object–as `request`. We return the request object and specify the proper template file, `home.html`.

Next are the views, both the project-level `urls.py` file, which serves as our website's entry point, and the app-specific `urls.py` file, which has the specific route and view for the home page.

The `django_project/urls.py` file is the initial entry point for all URL requests to our project. We must import the `include` function at the top and then use it to include the URL routes from the `pages` application, which will be set to the empty string of `""`.

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 app-level `pages/urls.py` file imports the view, `home_page_view`, and sets it to the URL path of the empty string, `""`.

Code

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

from .views import home_page_view

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

Start up the development web server using the `runserver` command.

Shell

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

If you navigate `http://127.0.0.1:8000/`, the home page is now visible.

FBV Company Homepage

Easy enough, right? So far, the only new concept introduced is the project-level `templates` directory.

Template Context, Tags, and Filters

Let's add a template context to our homepage view and then play around with some of Django's built-in tags and filters. A tag performs more complex operations such as loops, conditionals, and template inheritance. At the same time, a filter is used to perform more simple transformations that modify the display of variables, such as formatting dates, truncating text, or converting strings to uppercase. There are far too many tags and filters to memorize; it is helpful to know that for almost any content display, there are a host of native solutions.

A template `context` has a dictionary structure of keys and values. For demonstration purposes, we can add two: an `inventory_list` of three widgets and a `greeting` text string that deliberately mixes up upper and lowercase letters.

Code

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


def home_page_view(request):
    context = {  # new
        "inventory_list": ["Widget 1", "Widget 2", "Widget 3"],
        "greeting": "THAnk you FOR visitING.",
    }
    return render(request, "home.html", context)
```

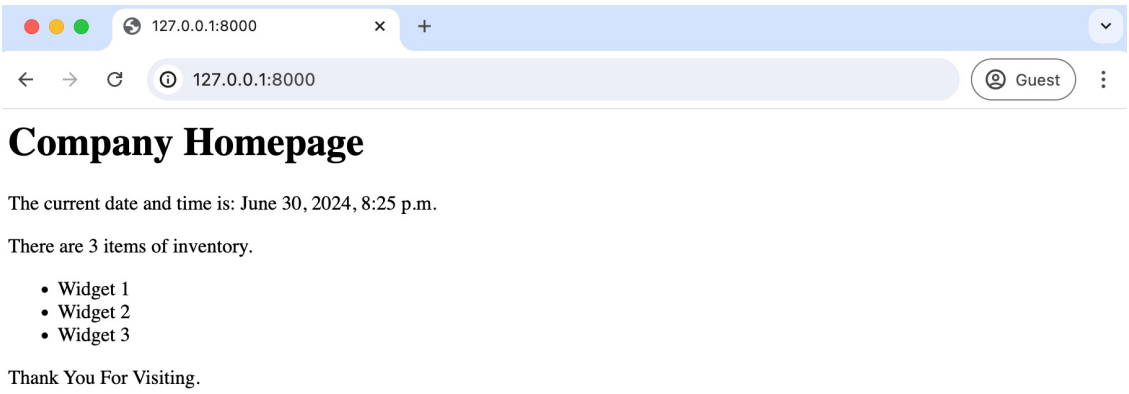
Update the `home.html` template file with the following code. The now tag displays the current date and/or time using DATE_FORMAT, one of several display options. Next, display the number of items in the `inventory_list` using the length filter, which works on both strings and lists.

Then, loop over each item in the for tag. The general syntax is `{% for item in item_list %}` where `item` is a variable name representing the current item in the loop and `item_list` is the sequence we are looping over. It is also critical to include an `{% endfor %}` tag to conclude any `for` loop. In this case, the sequence is named `inventory_list`. We could name the variable anything we like, but a description such as `item` is a common choice that makes the code easier to reason about. Finally, we use the title filter to convert a string into titlecase where each word starts with an uppercase letter followed by lowercase letters.

Code

```
<!-- templates/home.html -->
<h1>Company Homepage</h1>
<p>The current date and time is: {% now "DATETIME_FORMAT" %}</p>
<p>There are {{ inventory_list|length }} items of inventory.</p>
<ul>
 {% for item in inventory_list %}
 <li>{{ item }}</li>
 {% endfor %}
</ul>
<p>{{ greeting| title }}</p>
{% comment %}Add more content here!{% endcomment %}
```

The local web server should still be running in the background with the `runserver` command, so you only need to refresh the web page to see the changes.



Company Homepage with Context

The goal here is not to overwhelm you with many new Django features you need to memorize. There are way too many tags and filters to remember. Instead, it is to emphasize that for almost any web development task you have in mind, Django probably has a built-in solution, which is why the official documentation is indispensable and a daily part of development for a professional Django developer.

Class-Based Views and Generic Class-Based Views

The closest thing to a religious debate within the Django community concerns function-based views–what we've been working with so far–and class-based views. Early versions of Django only shipped with function-based views, which are arguably simpler to understand than their class-based counterparts because they mimic the HTTP request/response cycle. This book starts off using only function-based views for this very reason.

Function-based views do have their drawbacks. They lack an easy means of inheritance, meaning developers must repeat the same code snippets repeatedly in each view. That violates Django's general DRY (Don't Repeat Yourself) approach. But even when not repeating the same code, function-based views typically become lengthy in real-world projects and, therefore, difficult to reason about. It is common to see views with ten, twenty, or even more lines of logic, which becomes difficult to reason about.

Generic function-based views were introduced early in Django's development to abstract common patterns and avoid code duplication. Examples include:

- Write a view that displays a single template (as we've just done here)
- Write a view that lists all objects in a database model.
- Write a view that displays only one detailed item from a model.
- Write a view to create, update, or delete an object

The problem with generic function-based views was that there was no easy way to extend or customize them. As projects grow, this becomes more and more of an issue.

Django added class-based and generic class-based views to help with code reusability while retaining function-based views. Classes are a fundamental part of Python that rely on object-oriented programming (OOP) and inheritance, so one class can inherit attributes and methods from another. That means we *don't* have to include all the logic for a view in one place but rather can abstract common patterns and then customize or extend them as needed. A thorough discussion of Python classes and OOP is beyond the scope of this book. Still, if you need an introduction or refresher, I suggest reviewing the official Python docs, which have an excellent tutorial on classes and their usage.

Once you have used generic class-based views 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. However, this comes 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. An entire website, Classy Class-Based Views, is dedicated to helping Django developers decipher generic class-based views.

The Django codebase itself was shifted to primarily class-based and generic class-based views. Generic function-based views were deprecated in Django 1.3 and removed entirely in version 1.5.

The result of these changes to Django over the years is that there are now three different ways to write a view in Django: function-based, class-based, or generic class-based. This is understandably very confusing to beginners.

Earlier editions of this book focused entirely on generic class-based views, but this version includes both. A Django developer needs to understand how each

approach works, even if they will undoubtedly have a personal preference over time.

TemplateView

Let's create a second web page for our Company site, this time using TemplateView, a generic class-based view. This will be for an about page that also takes advantage of the template context and the Django Templating Language.

At the top of the `views.py` file, import `TemplateView` from the `django.views.generic` module. Then create a class, `AboutPageView`, that extends `TemplateView` and specifies a template, `about.html`. In Python, the convention for naming classes is to use "CamelCase," where the first letter in a word is capitalized, and there are no underscores between words.

Code

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


def home_page_view(request):
    context = {
        "inventory_list": ["Widget 1", "Widget 2", "Widget 3"],
        "greeting": "THAnk you FOR visitING.",
    }
    return render(request, "home.html", context)


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

Next, update the `pages/urls.py` file to display the new view. We import `AboutPageView` and set a route to `about/` while specifying `AboutPageView` as the view.

Code

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

from .views import home_page_view, AboutPageView  # new

urlpatterns = [
    path("about/", AboutPageView.as_view()),  # new
    path("", home_page_view),
]
```

Note the addition of the as_view() method that returns a callable view. The only fundamental difference in configuring URLs for class-based versus function-based views is that this method must be added.

The last step is creating our template file, `about.html`. With your text editor, add this new file within the existing `templates` directory with the following code:

Code

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

Now make sure the local server is running and navigate to `127.0.1:8000/about/` in your web browser.

Company About Page

Easy enough, right?

get_context_data()

One of the most powerful, useful, and commonly used methods in Django is get_context_data(). It is the recommended approach for updating the template context in a generic class-based view. Let's use it now to add context data to the About page.

Code

```
# pages/views.py
...
class AboutPageView(TemplateView):
    template_name = "about.html"

    def get_context_data(self, **kwargs):  # new
        context = super().get_context_data(**kwargs)
        context["contact_address"] = "123 Main Street"
        context["phone_number"] = "555-555-5555"
        return context
...
```

To begin with, we override the existing `get_context_data()` method. The first parameter is `self`, and the second is `**kwargs`, allowing us to pass in keyword arguments. This is how we can add a key/value to the context.

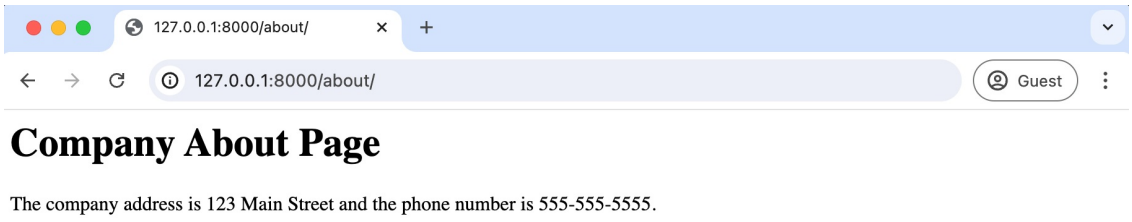
The next step is to set a variable called `context` with the existing value of our context. How do we do that? Call `super()` on `get_context_data` and include any keyword arguments. Then we add two keys, `contact_address` and `phone_number`, along with their corresponding values. The last step is always to explicitly return the now-updated context.

To render context variables in our template, we use double brackets, `{{ }}`.

Code

```
<!-- templates/about.html -->
<h1>Company About Page</h1>
<p>The company address is {{ contact_address }} and the phone number is
  {{ phone_number }}.</p>
```

Refresh the About page in your web browser to see this information displayed.



Company About Page with Context

Using generic class-based views might seem unnecessary at the moment, but their true power will become apparent in the next chapter when we start working with a database.

Template Inheritance

This chapter is all about templates and views. We've already covered a lot of information: template context, template tags and filters, and class-based views. However, one more powerful feature 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!

Within the `templates` directory, create a `base.html` file containing a header with links to our home page and about pages. This is our parent template, which all other child templates will inherit from. To define what areas can be overridden, we'll use block tags in the syntax `{% block content %}` and `{% endblock %}`. Anything within the block tags can be overridden in a child template.

Code

```
<!-- templates/base.html -->
<header>
 <a href="/">Home</a> |
 <a href="/about">About</a>
</header>

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

The extends tag allows us to establish parent/child relationships between our templates by specifying the parent template. Add it to the top of the `home.html` and `about.html` templates. Then define our child template content with `{% block content %}` and `{% endblock %}` tags.

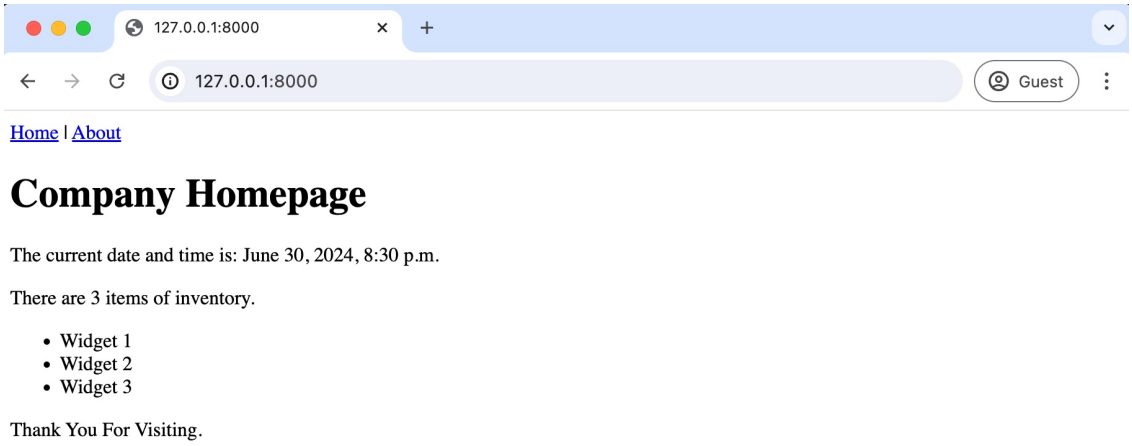
Code

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

{% block content %}
<h1>Company Homepage</h1>
<p>The current date and time is: {% now "DATETIME_FORMAT" %}</p>
<p>There are {{ inventory_list|length }} items of inventory.
<ul>
 {% for item in inventory_list %}
 <li>{{ item }}</li>
 {% endfor %}
</ul>
<p>{{ greeting| title }}</p>
{% comment %}Add more content here!{% endcomment %}
{% endblock %}
```

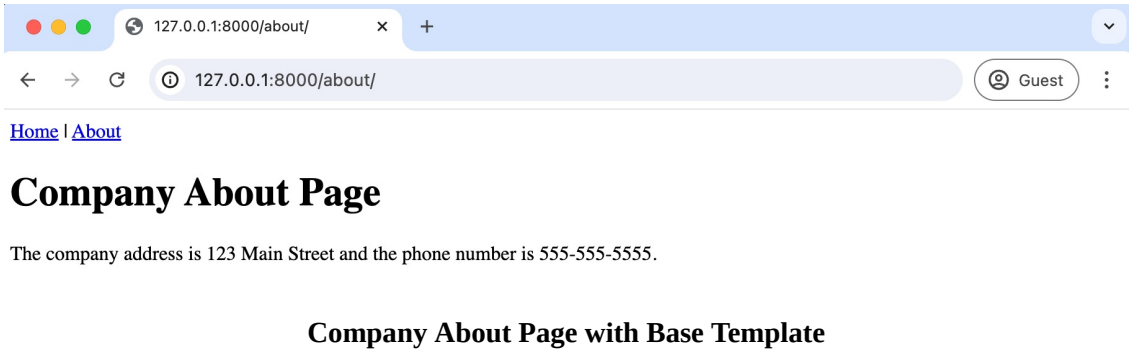
Code

```
<!-- templates/about.html -->
{% extends "base.html" %}
{% block content %}
<h1>Company About Page</h1>
<p>The company address is {{ contact_address }} and the phone number is
  {{ phone_number }}.</p>
{% endblock %}
```

Refresh each web page in your browser to see the results:



Company Home Page with Base Template



Company About Page with Base Template

Each page now contains the `base.html` header with navigation links for Home and About pages.

Named URLs

Experienced web developers may have noticed a problem with our current approach to page links. We have hardcoded URL paths in the `views.py` and `urls.py` files. In each place, we specified `/` for the Homepage and `about/` for the About page. What happens if we change the URL path in one place but not another? We'll get a 404 error for a page not found.

Django takes seriously the idea of having a canonical location for logic. In this case, we want to reference a URL and its associated view once and only once. To

do that, we can add a name to our URLs. The path() function accepts the following arguments: `path(route, view, kwargs=None, name=None)`. By default, `kwargs` and `name` are set to `None`, but we can update `name` here.

Code

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

from .views import home_page_view, AboutPageView

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

Now, whenever we want to refer to a specific URL path, we can do so using the named URL in our template via the built-in url template tag. Update the `base.html` file with the code below.

Code

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

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

If you refresh your website in the browser, both pages and their links work as before. The URL path is now set in only one location–in the `urls.py` file. Named URLs make your projects easier to maintain and modify, as changes to URL patterns don't require changes in multiple places in our code. They are a best practice that should be used in all Django projects.

Tests

In the previous chapter, we wrote a single unit test for each webpage, checking that it returned an HTTP 200 code. Let's do that again as a quick recap before adding more robust tests to our website. Update the `pages/tests.py` file with the code below:

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)
```

Then, run the tests by quitting the local web server with `Control+c` and typing `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
```

So far, so good. What's changed in the Company Website versus the previous chapter's Personal Website? We now have URL names for each URL route, so we should check that they work as expected. We can use the handy Django utility function reverse. Instead of going to a URL path, it looks for the URL name. It is generally a bad idea to hardcode URLs, especially in templates. We can avoid this by using `reverse`. See a further explanation here in the docs, but we will use `reverse` more in coming chapters.

Open the existing `pages/tests.py` file in your text editor and at the following code:

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)
```

At the top, we have imported `SimpleTestCase` since we are not working with a database and then the `reverse` function. There are two classes of tests corresponding to each web page. `HomepageTests` checks that the homepage at `/` returns a 200 status code and then checks that calling `reverse` on the named URL of "home" does the same. The pattern for both tests is repeated for the About page in the `AboutpageTests` class.

Run 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 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 contain the expected text of `"<h1>Company Homepage</h1>"` and `"<h1>Company 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>Company 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>Company 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 it is repetitive. For example, we set the `response` each time for all eight tests. Generally, following DRY (Don't Repeat Yourself) coding is a good idea, but unit tests work best when they are self-contained and highly verbose. As a test suite expands, combining multiple assertions into a smaller number of tests might make more sense for performance reasons.

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
```

Then, create a `.gitignore` file instructing Git on what not to track. We will focus on three areas: the `.venv` directory with virtual environments, the `__pycache__` directory with compiled bytecode, and the database file, `db.sqlite3`.

.gitignore

```
.venv/
__pycache__/
db.sqlite3
```

The next step is to create a `requirements.txt` file listing the contents of our virtual environment.

Shell

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

The final step is to run `git status` again to confirm that `requirements.txt` is included, but the three items in the `.gitignore` file are ignored. Then, add all intended files and directories accompanied by 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 `company-website` 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 “��r 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/company-website.git
(.venv) $ git branch -M main
(.venv) $ git push -u origin main
```

Conclusion

Congratulations on building and deploying your third Django project! This time, we used both function-based views and generic class-based views to build out our website. We incorporated template inheritance, named URLs, and added more advanced tests. 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 5: Message Board Website

In this chapter, we will create our first database-backed website, a Message Board application. We will learn about relational databases, write a Django model, perform queries, and manipulate using the powerful built-in admin interface. We will also write function-based and class-based views before ending with more advanced tests to ensure everything works properly.

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 performing the following actions:

- 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~=5.0.0
(.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 now present with our local database and Django default tables. Spin up our local server to confirm everything works correctly.

Shell

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

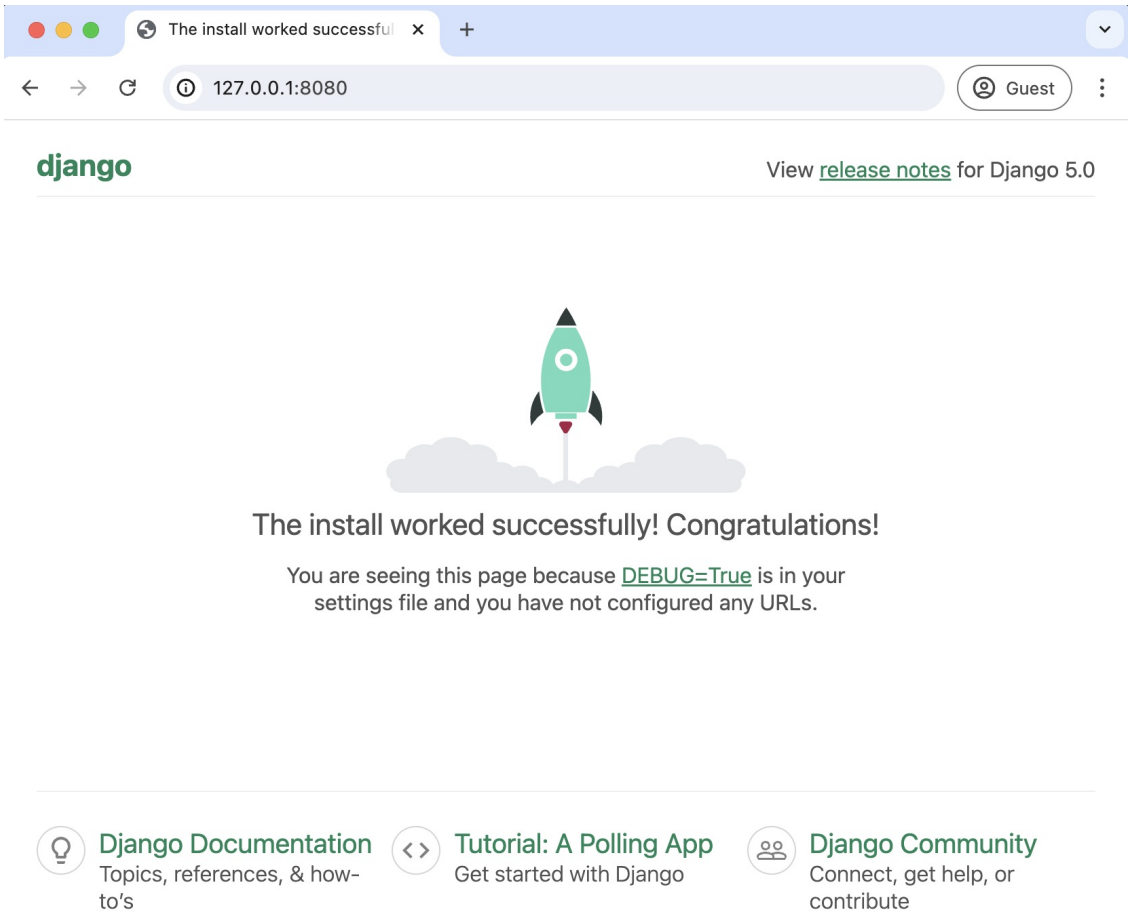
In your web browser, navigate to `http://127.0.0.1:8000/` to see the familiar Django welcome page.

A major design philosophy of Django, which you will see repeatedly in this book, is that it allows for customization. For example, we don't have to use port 8000 for local development. We could change it to any other available port, such as 8080. Stop the server with `Ctrl + c` and restart it by passing in the desired port number.

Shell

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

If you refresh `http://127.0.0.1/:8000/`, you will see an error message, but switching to `http://127.0.0.1:8080/` displays our Hello, World! greeting.



Django Welcome Screen Port 8080

We won't need to move off port 8000 in this book, but it is only a command-line customization away if you need a different port number in a more complex Django project.

Databases

Before implementing our Message Board model, it is worth reviewing 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 and 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 is 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 we will focus on that.

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, which 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 to implement them correctly. For example, before writing any actual code, let's look at structuring the data in our Message Board model. We will have only one table called "Post" and a single field, "text," containing the contents of a message. If we drew this out as a simple schema, it would look something like this:

Post Schema

```
Post
--------
TEXT
```

The actual database table with rows of actual messages would look like this:

Post Database Table

```
Post
--------
TEXT
My first message board post.
A 2nd post!
A third message.
```

Database Model

Now that we know how our database table should look, let's use Django's ORM to define it using Python. 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 that 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) `django.db.models.Model` and then add our fields. Type in the following code, which we will review below:

Code

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


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

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. We will explore these later. For now, we have written our first model!

Activating Models

After creating a model, the next step is activating 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 record any changes to the database models, which means we can track changes over time and debug errors as necessary.
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, Django will detect the changes and generate separate migration files for each app that has modifications. But if you

want more granular control, such as here, specify *which* app should have a new migrations file for relevant changes.

Django Admin

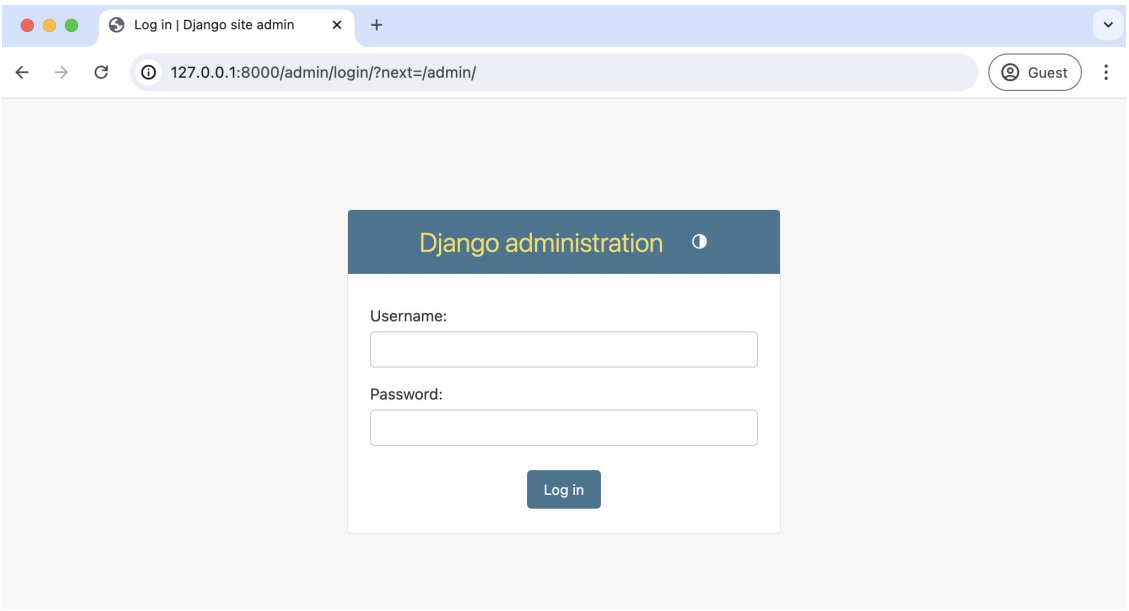
One of Django's killer features is its robust admin interface that visually interacts with data. It came about because Django started as a newspaper CMS (Content Management System). The idea was that journalists could write and edit their stories in the admin without needing to touch "code." Over time, 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

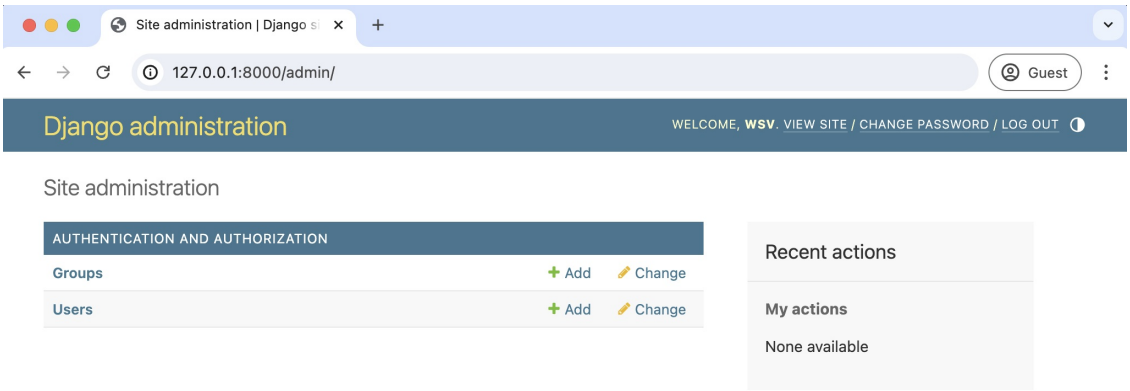
```
(.venv) $ python manage.py createsuperuser
Username (leave blank to use 'wsv'): wsv
Email: will@learndjango.com
Password:
Password (again):
Superuser created successfully.
```

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 languages, 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`.

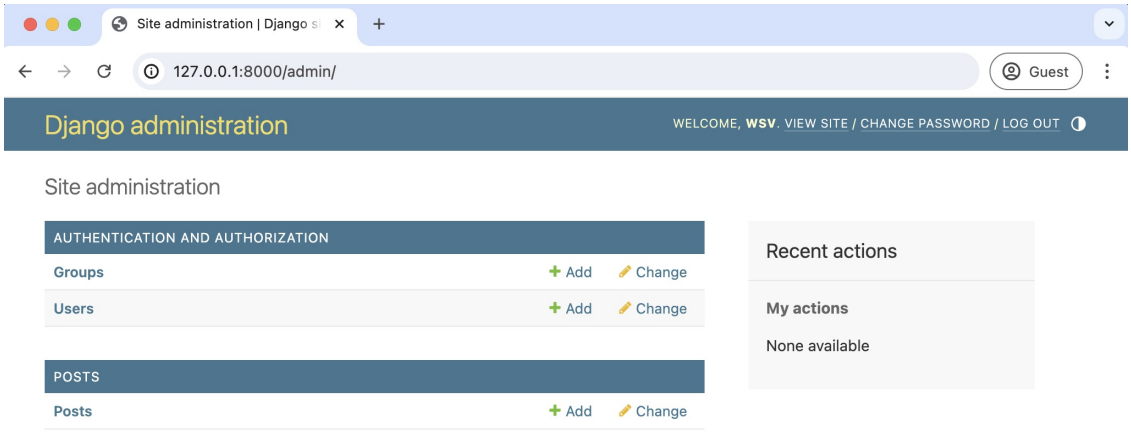
But where is our `posts` app since it is not displayed on the main admin page? Just as we must explicitly add new apps to the `INSTALLED_APPS` config, we must also update an app's `admin.py` file for it to appear in the admin.

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

Code

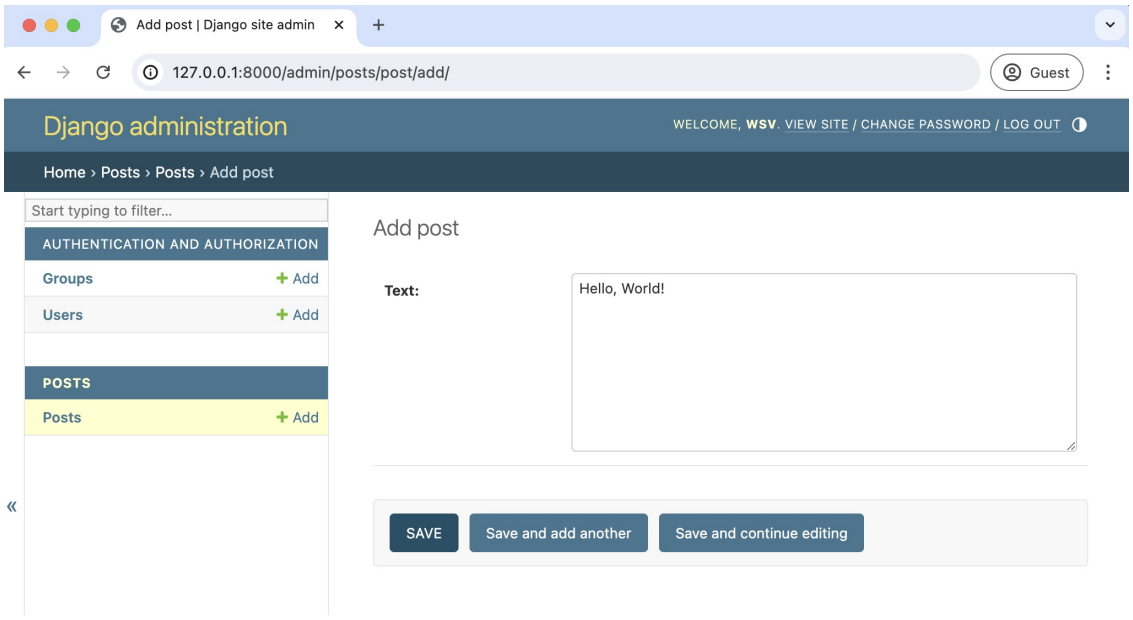
```
# posts/admin.py
from django.contrib import admin

from .models import Post

admin.site.register(Post)
```

Django 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:



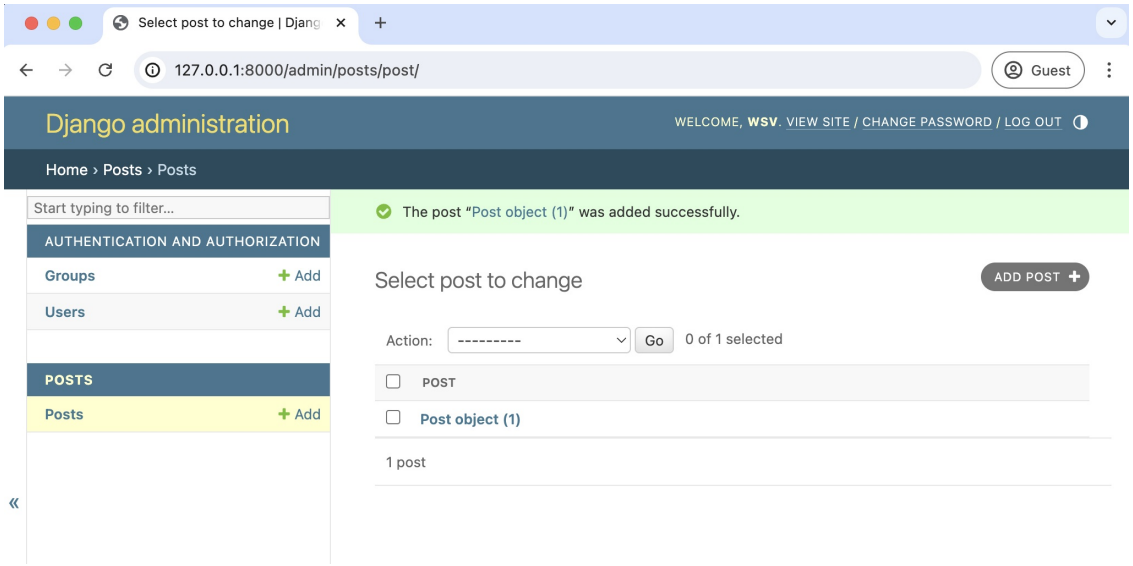
Admin Updated Homepage

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.



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!



Admin Posts List

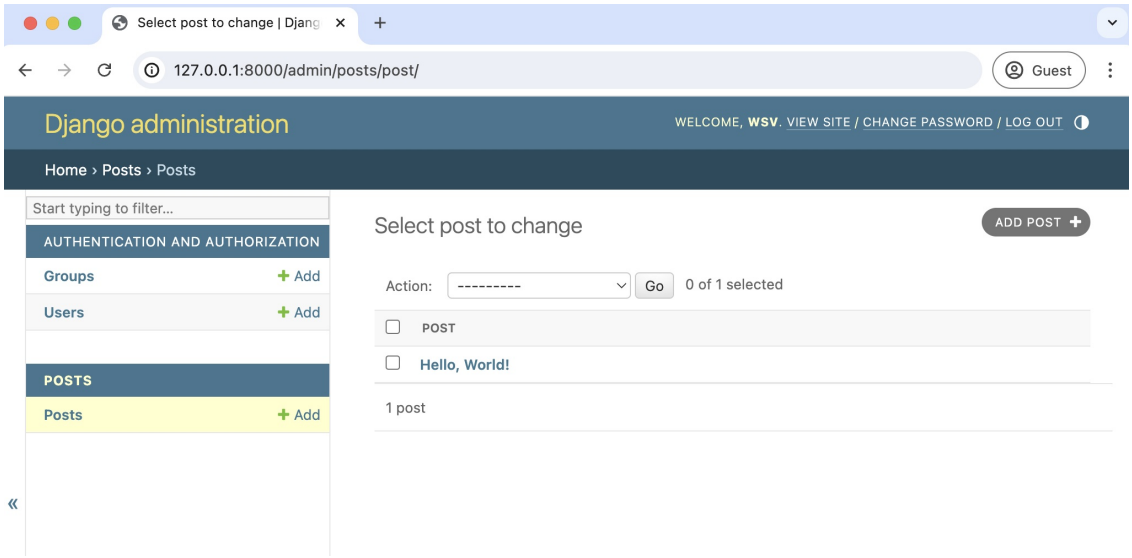
Let’s change that. Within the `posts/models.py` file, add a new method called `__str__`, which provides a human-readable representation of the model. In this

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]
```

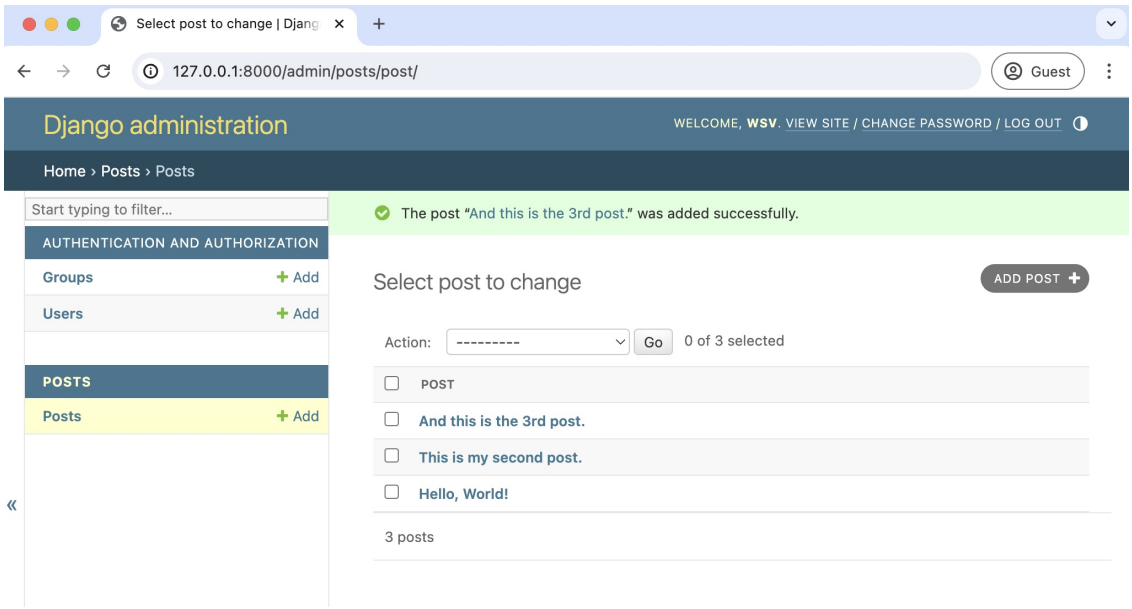
If you refresh your Admin page in the browser, you'll see that it now represents our database entry in a much more descriptive and helpful way.



Admin Readable Post

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

Let's add two more entries using the same method, so we have three total to work with in the next section. You can use the "Add Post +" button in the upper right corner.



Admin with Three Posts

Function-Based View

To display our message board posts on the homepage we have to wire up a view, template, and URL. This pattern should start to feel familiar now.

Let's begin with the view. We'll initially write a function-based view and switch to a generic class-based view. In the `posts/views.py` file, replace the default text and enter the Python code below:

Code

```
# posts/views.py
from django.shortcuts import render
from .models import Post


def post_list(request):
    posts = Post.objects.all()
    return render(request, "post_list.html", {"posts": posts})
```

On the first line, we import the render() shortcut function, which combines a template with a context dictionary and returns an `HttpResponse` object. Then, we import our database model, `Post`, from the `models.py` file.

We define a function, `post_list`, and name the request object "request" per Django convention. Then, we set a variable, `posts`, to a database query

containing all Post objects. Then we use `render()` to return the `request` object, define the template as the second argument, and then in the third argument, define a context dictionary called "posts" that matches the value of the `posts` variable we set on the previous line.

Let's examine `Post.objects.all()` in more detail as it is the first example of a Database query via the Django ORM.

1. `Post`: This refers to our model class, which represents a table in the database where each row corresponds to an instance of the `Post` model
2. `objects`: This is the default manager for the `Post` model. A manager provides a way to interact with the database and perform queries. By default, Django adds a Manager named "objects" to every Django model class.
3. `all()`: This is a method provided by the manager that returns a QuerySet containing all instances of the `Post` model from the database. A QuerySet is a collection of database queries to retrieve objects.

The QuerySet API reference is an invaluable resource in the official documentation that covers all available methods. You are not expected to memorize all of these. Rather, the traditional approach is to have a problem with querying data and then search for a built-in method. You will likely find one already exists.

Templates and URLs

We already have a model and view, which means only a template and URL are left to configure. 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 Django can look in this new templates directory.

Code

```
# django_project/settings.py
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [BASE_DIR / "templates"],  # new
```

```
        "APP_DIRS": True,
        ...
    },
]
```

In your text editor, create a new file called `templates/post_list.html`. Our template context contains a dictionary called `posts` which we need to loop over via the 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/post_list.html -->
<h1>Message Board Homepage</h1>
<ul>
 {% for post in posts %}
 <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 post_list

urlpatterns = [
    path("", post_list, name="post_list"),
]
```

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

Homepage with Three Posts

If you navigate to the Django admin and add or delete message board posts, the homepage will be updated to reflect the changes.

ListView

In the previous chapter, we wrote a function-based view and then switched to the built-in generic TemplateView to display a template file on our homepage. Listing out all items in a database model is so common that a generic class-based view exists for this, too, called ListView. We will now switch over to it for educational purposes. There is no right or wrong answer regarding function-based versus generic class-based views: it is a matter of preference.

In the `posts/views.py` file, comment out the existing code for our function-based view and add new code for the class-based view implementation. Import on the top line `ListView` and then create a 'PostList' class that extends it. We define the desired model, `Post`, and then the template name, `post_list.html`.

Code

```
# posts/views.py
# from django.shortcuts import render
# from .models import Post


# def post_list(request):
#     posts = Post.objects.all()
#     return render(request, "post_list.html", {"posts": posts})

from django.views.generic import ListView  # new
from .models import Post


class PostList(ListView):  # new
    model = Post
    template_name = "post_list.html"
```

The current template uses a context dictionary called `posts`. By default, ListView returns a context variable called `<model>_list`, where `<model>` is our model name. Since our model is named `post`, we need to loop over `post_list` with our `for` loop. The rest of the template remains the same.

Code

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

The final step is to update `posts/urls.py` with the new name for our view, `PostList`. We can comment out or delete the previous code. Remember that we must also add the `as_view()` method to return a callable view.

Code

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

# from .views import post_list
from .views import PostList  # new

urlpatterns = [
    # path("", post_list, name="post_list"),
    path("", PostList.as_view(), name="home"),  # new
]
```

And that's it! If you refresh your home page, it should work just as before. We have written a new view, updated the name of the context variable in the template, and updated the URL file.

Initial Commit

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 tracked.

.gitignore

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

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

Shell

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

Tests

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, we can make a separate test database, fill it with sample data, and 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 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 the 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. Only functions starting with `test` are considered unit tests by the `python manage.py test` command.

It is time to check our URLs, views, and templates. We 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 “post_list.html”
- Homepage content matches what we expect in the database

Since this project has only one webpage, we can include all of these tests in our existing `PostTests` class. 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, "post_list.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 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 and 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, "post_list.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. 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. Since you should already have a GitHub account from previous chapters, 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 built and tested our first database-driven app and learned how to create a database model, update it with the admin panel, and write tests. We also looked at both function-based and class-based approaches for the views. In the next chapter, we will build a more complex *Blog* application with user accounts for signup and login, allowing users to create/edit/delete their posts and then add CSS for styling.

Chapter 6: Blog Website

In this chapter, we will begin building a *Blog* application that allows users to read, create, edit, and delete posts. This functionality, CRUD (Create-Read-Update-Delete), is the dominant pattern for most websites. If you think about Facebook, Instagram, or Reddit, all you do is read posts and sometimes create, edit, or delete them. That's what we'll implement here with a blog app with a homepage listing all posts and an individual page for each post. We'll also introduce CSS for styling, learn about static files, and write more advanced tests to ensure everything works as expected.

Initial Set Up

The setup for this project is similar to past examples in this book:

- 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, add 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`, create a new app called `blog`, and `migrate` the initial database.

Shell

```
(.venv) $ python -m pip install django~=5.0.0
(.venv) $ python -m pip install black
(.venv) $ django-admin startproject django_project .
(.venv) $ python manage.py startapp blog
(.venv) $ python manage.py migrate
```

Regarding app names, it is generally a best practice to use the plural form, such as `pages` or `posts`, unless your app name doesn't make sense as a plural, like `blog`, so we use the singular form here. 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
```

You should see the friendly Django welcome page if you navigate to `http://127.0.0.1:8000/` in your web browser.

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.

Blog Post Models

Before we write the code for our Django model, let's take a moment to visualize how we want the information in our database to be structured. In our previous example, the Message Board app, we only had a single field for content. Here,

we want a database table called Post with three fields: Title, Author, and Body. The actual database table with columns and rows should look like this:

Post Database Table

```
Post
--------
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.
```

Remember that the `models.py` file is the single, definitive source of information about our data, containing the necessary fields and behaviors of the stored data. We can write Python in a `models.py` file, and the Django ORM will translate it into SQL. Write the following code in 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 the `models` module and then create a class, `Post`, that extends it. The `Post` class has three fields (think of them as columns) for `title`, `author`, and `body`. Each field must have an appropriate field type. The first two use `CharField`, meaning a Character Field with a maximum character length of 200, while the third uses `TextField`, 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 file and migrate the change to apply it 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
Migrations for 'blog':
  blog/migrations/0001_initial.py
    - Create model Post
(.venv) $ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, blog, contenttypes, sessions
Running migrations:
  Applying blog.0001_initial... OK
```

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

Primary Keys and Foreign Keys

We could hop over to the Django admin and add data to our blog post model. However, two more important concepts—primary and foreign keys—must be covered before building the rest of the Blog app.

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 Post model, 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. With 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 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 corresponding to a user. 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

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
```

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 choosing `CASCADE` is typically safe, as we do here.

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
Migrations for 'blog':
  blog/migrations/0002_alter_post_author.py
    - Alter field author on post
(.venv) $ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, blog, contenttypes, sessions
Running migrations:
  Applying blog.0002_alter_post_author... OK
```

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 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.
```

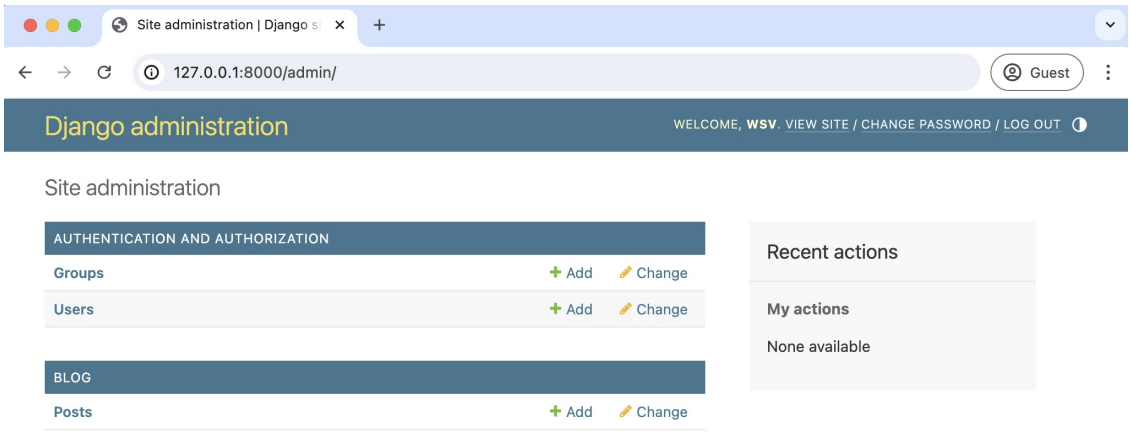
Now rerun the Django server with the command `python manage.py runserver` and navigate to the admin at `127.0.0.1:8000/admin/`. Log in with your new superuser account.

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)
```

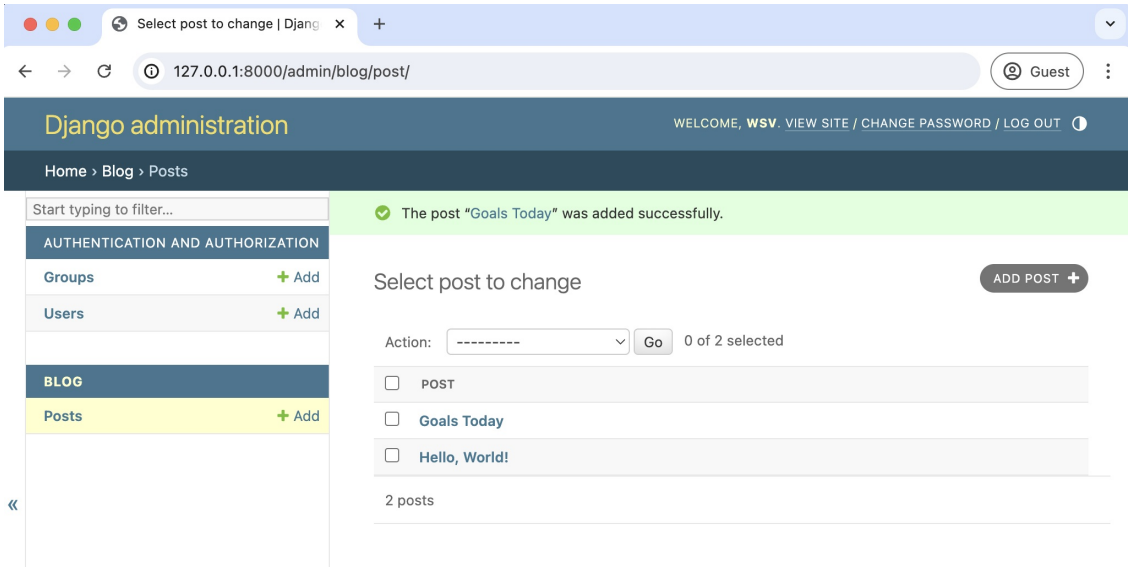
If you refresh the page, you'll see the update.



Admin Homepage

Let's add two blog posts so we have some sample data. Click 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. To change this, we could add field options to our model to make a given field optional or default to a specified value.



Admin Change List with Two Posts

Although our model has three fields, the Django admin defaults to displaying whatever is in the `__str__` method in the list view. In our case, that is the `title`

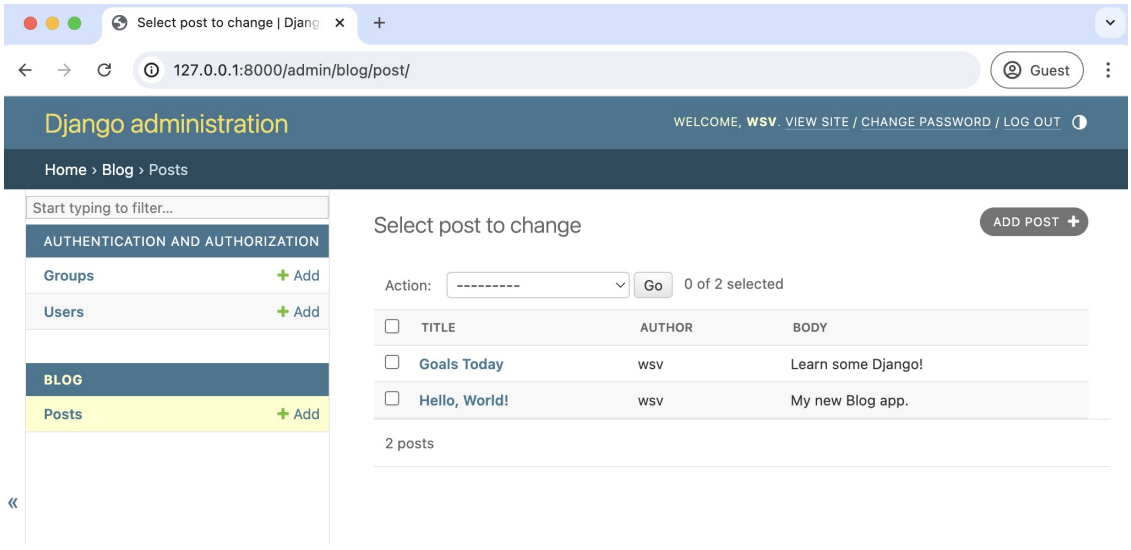
field. However, it is quite straightforward to customize the admin further to display all three fields.

To do this, we can extend ModelAdmin by creating a new class, `PostAdmin`. Within it, we can set list_display to control what is shown on the admin change list page. We also must register both the model, `Post`, and the extended `ModelAdmin` class we've created, `PostAdmin`, at the bottom of the file.

Code

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


class PostAdmin(admin.ModelAdmin):  # new
    list_display = (
        "title",
        "author",
        "body",
    )


admin.site.register(Post, PostAdmin)  # new
```

All three model fields will be visible if you refresh the admin change list page.



Admin Change List with All Fields

Now that our database model is complete, we must create the necessary view, URL, and template to display the information in our web application.

Views

Our view needs to list all available blog posts. We can write it as a function-based view; the code is identical to what we used in the last chapter for our Message Board. In web development, we are often performing similar tasks over and over again. This is what led to the development of generic class-based views!

Code

```
# blog/views.py
from django.shortcuts import render
from .models import Post


def post_list(request):
    posts = Post.objects.all()
    return render(request, 'home.html', {'posts': posts})
```

As a brief recap, we import the `render()` shortcut function and the `Post` model at the top of the file. Then we create a function, `post_list`. The first parameter is named `request` by convention and represents an instance of the `HttpRequest` object that triggered the view. Then, we set a variable, `posts`, equal to a QuerySet containing all Post objects in the database using the default model manager name of `objects` and the built-in `all()` method. Finally, we use `render` to return the `request` object, specify the template, and add a context dictionary named `posts` equal to the variable `posts` containing all Blog posts.

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 post_list

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

At the top, we import the `path` module and our view, `post_list`. Then we set a single route at the empty string, `""`, which matches the route URL of our website. We pass in the view, `post_list`, as the second argument and then add an optional “home” name that will come in handy shortly in our template.

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.

Templates

With our URLs and views now complete, we’re only missing the third piece of the puzzle: templates. Let’s use template inheritance to avoid repeating code, starting with a `base.html` file and a `home.html` file that inherits from it. Later, when we add templates for creating and editing blog posts, they can also be inherited from `base.html`.

Start by adding 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>
```

The link to `url 'home'` means that we expect a URL with the name "home" to power our homepage. The code between `{% block content %}` and `{% endblock content %}` is designed to be filled by other templates. Speaking of which, here is the code for `home.html`.

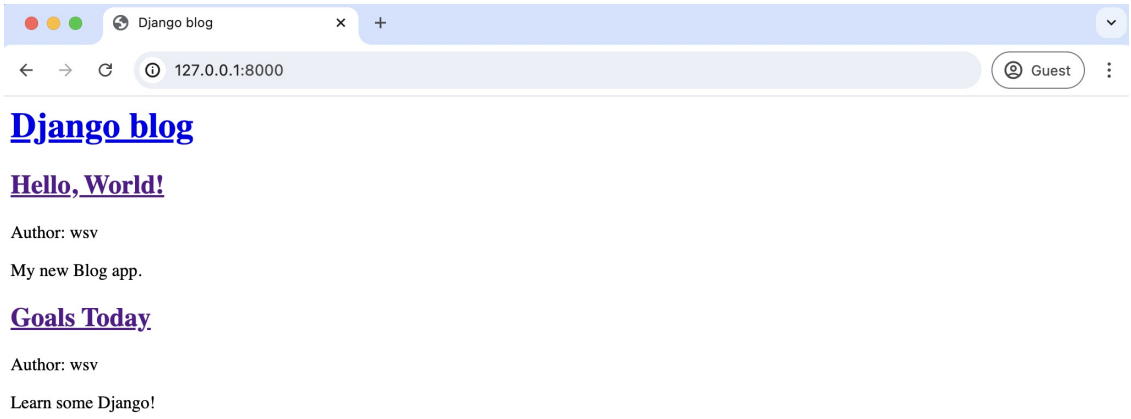
Code

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

{% block content %}
{% for post in posts %}
<div class="post-entry">
 <h2><a href="">{{ post.title }}</a></h2>
 <p>Author: {{ post.author }}</a>
 <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. We are looping over the context dictionary `posts`, which we defined in our view, and naming each item as `post`.

Then, we can use dot notation to display fields like `post.title`, `post.author`, and `post.body`.

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 by adding some styling.

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 to our project yet, 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.

In production, things are more complex, which we will cover properly in the deployment section of this book. The central concept to understand 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/"
```

For local development, we don't have to worry about static files because the web server–run via the `runserver` command–will automatically find and serve them for us. It is worth mentioning that modern web browsers implement quite aggressive caching, so if you have made changes and don't see the on your local web project it is possible you need to clear your web browser cache. A good way to check if this is the issue is to try a different web browser, for example Safari rather than Chrome, if you fail to see a static file change materialize on the page.

The first question when adding a new static file is where to place it. 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
```

STATICFILES_DIRS defines the additional locations the built-in staticfiles app will traverse looking for static files beyond an `app/static` folder. We need to add our project-level `static` folder to this configuration.

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
```

Add a new file within your text editor in 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;
}
```

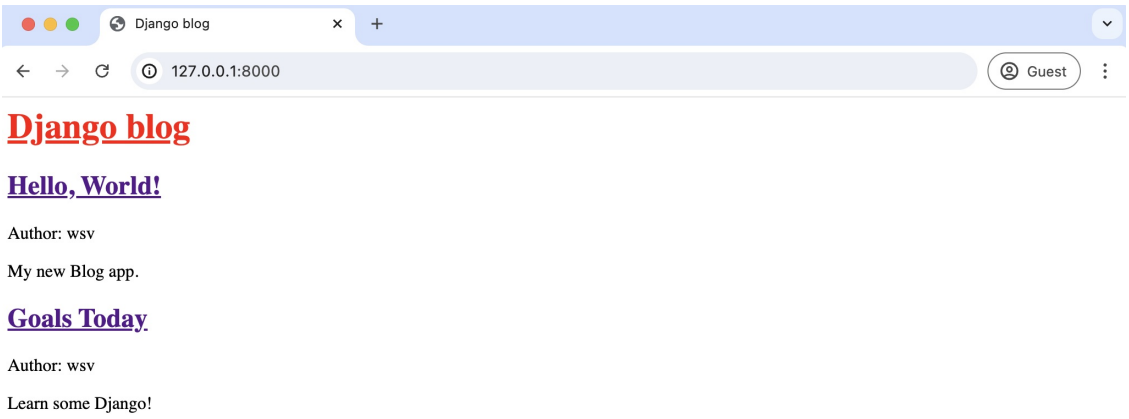
The last step is adding the static files to our templates by adding `{% load static %}` to the top of `base.html`. Because our other templates inherit from `base.html`, we only have to add this once. Include a new line at the bottom of the `<head></head>` code that explicitly references our new `base.css` file.

Code

```
<!-- templates/base.html -->
{% load static %}
<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. 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 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.

Blog Homepage with CSS

Individual Blog Pages

Now, we can add functionality for individual blog pages. We need to create a new view, URL, and template to do that. I hope you're noticing a pattern in Django development!

Start with the view. At the top of our views file, import the shortcut function get_object_or_404(), which calls the get QuerySet method to return an object or raises a Http404 error if unsuccessful.

We will name the view function `post_detail` as it represents the detailed view of a blog post. It accepts two parameters: the first, `request`, is an instance of the `HttpRequest` object; the second, `pk`, is a parameter extracted from the URL that identifies the specific blog post to be displayed by its primary key.

Let's take a moment to focus on that last sentence, as it gets at the heart of understanding this function. To specify an individual blog post, we need a way to say that out of the list of all blog posts in the database, we should choose this specific one. The easiest and default way is to use the primary key ID associated with each record. For example, the first entry has a primary key (PK) of `1`, the second of `2`, and so on automatically set by the Django ORM.

Within the body of the `post_detail` function there are two lines. First, we call `get_object_or_404` on the `Post` model and specify that the `pk` primary key

matches the parameter `pk` from the URL. If we want the URL `/1`, our function will call the blog post with a primary key of `1`. When you see this in action, it will make more sense. We set the variable `post` to equal this specific blog post. Then we use the `render()` function to return a response, where the first argument is the `request` variable, the second is the template `post_detail.html`, and the third is the template context where we create a variable `post` that matches the `post` from the previous line.

Code

```
# blog/views.py
from django.shortcuts import render, get_object_or_404  # new
from .models import Post


def post_list(request):
    posts = Post.objects.all()
    return render(request, 'home.html', {'posts': posts})


def post_detail(request, pk):  # new
    post = get_object_or_404(Post, pk=pk)
    return render(request, "post_detail.html", {"post": post})
```

This pattern for accessing a list of information from the database or a single item is repeated repeatedly in web development with Django, so don't worry if it's a little confusing at the moment.

With our view created, the next two steps are adding a URL route and a template. In the `blog/urls.py` file, import the new view, `post_detail`, and add a URL route at `post/<int:pk>/`. This pattern will look a bit strange initially as it is our first use of a Django path converter. Up to this point, we have hardcoded our URL routes, but it is more common to use variables. In this case, we specify that individual posts will start at `post/`, but then we use `int` to specify that the captured value from the URL should be treated as an integer and the variable's name passed to the view is `pk`. The second argument is the view name, `post_detail`, and we add the optional third argument of a name that is also `post_detail`.

Code

```
# blog/urls.py
from django.urls import path
from .views import post_list, post_detail  # new

urlpatterns = [
    path("post/<int:pk>/", post_detail, name="post_detail"),  # new
    path("", post_list, name="home"),
]
```

The last step is adding a new template file 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`. The template context variable `post` contains information from this particular blog post, and we can display individual fields using `post.title` and `post.body`.

If you start 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/`.

Blog Post One Detail

Woohoo! You can also go to `http://127.0.0.1:8000/post/2/` to see the second entry.

Blog Post Two Detail

To make our lives 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="">` with `<a href="{% url 'post_detail' post.pk %}">`.

Code

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

{% block content %}
{% for post in posts %}
<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()

Currently, we are using the `url` template tag, which means that every time we want to display an individual blog post in this template or other templates, we

must repeat the pattern `{% url 'post_detail' post.pk %}`. If the URL pattern changes, we need to update every template where the URL is constructed, which increases the risk of errors.

A better approach is to use the built-in get_absolute_url() method, which tells Django how to calculate the canonical URL for our model object. In the `blog/models.py` file, add a new method for getting the `get_absolute_url`.

Code

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


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

At the top, we import reverse(), a utility function for our URLs. Then, we define `get_absolute_url` using `self` as the first parameter referring to the model instance on which the method is called. This is a standard practice in Python for instance methods. The `reverse()` function accepts the URL name and keyword arguments or "kwargs." In this case, we are setting the variable `pk` to equal the primary key of our model instance.

We don't need to update the migrations files because we aren't changing the database schema here. Migrations are only required when changes to the model affect the database schema, such as adding or removing fields, changing field types, or modifying relationships between models.

In the template file, update the `href` link to now use `post.get_absolute_url`.

Code

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

{% block content %}
{% for post in posts %}
```

```
<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 a project's lifetime. 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 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. There is quite a lot to test!

First, we can set up our test data and check the content of the `Post` model. 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 setup 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 each of 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 the 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 must 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 mindlessly 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 mistake 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. Initialize our directory and review all the added content by checking the status.

Shell

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

Oops, we do not want to include the .venv directory 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__/
db.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, which will display a list of all posts on the homepage and individual pages for each post. We looked at static files for the first time and added some styling with CSS. We also learned about the best practice of using `get_absolute_url` in our templates and made substantial progress on our test suite.

In the next section, we'll switch to generic class-based views and add forms to create, update, and delete blog posts 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 switching to class-based views and 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. Any time you accept user input, there are significant security concerns since users are uploading information to your database. All forms must be properly rendered, validated, and saved to the database. Writing this code by hand would be time-consuming and complex, so Django comes with powerful built-in Forms and generic editing views for common tasks like displaying, creating, updating, or deleting a form.

ListView and DetailView

Currently, we are using function-based views for our page listing all blog posts and powering individual posts. We could continue down this page and create function-based views for create, edit, and delete functionality; however, doing so requires a lot more code and is error-prone compared to using generic class-based views designed explicitly for the job. For this reason, we will switch to using generic class-based views throughout the rest of the book.

Let's look at the current `blog/views.py` file.

Code

```
# blog/views.py
from django.shortcuts import render, get_object_or_404  # new
from .models import Post


def post_list(request):
    posts = Post.objects.all()
    return render(request, 'home.html', {'posts': posts})


def post_detail(request, pk):  # new
    post = get_object_or_404(Post, pk=pk)
    return render(request, "post_detail.html", {"post": post})
```

Switching over to generic class-based views is relatively straightforward. Here is the updated code for the views:

Code

```
# blog/views.py
from django.views.generic import ListView, DetailView
from .models import Post


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


class BlogDetailView(DetailView):
    model = Post
    template_name = "post_detail.html"
```

At the top, we import the generic views `ListView` and `DetailView`, along with our model, `Post`. We used `ListView` previously in the Message Board app, but DetailView is new. It is used for detail pages and has a format extremely similar to `ListView`. We pass in the generic class-based view and define the model and template for both views. That's it.

Next up is the URLs file. We change the views imports to match our new ones, `BlogListView` and `BlogDetailView`. Then, we update the second argument in both routes, specifying the view name and adding `as_view()` to transform the class into a callable view function.

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"),  # new
]
```

The final update is to our template. Previously, in our function-based list and detail views, we named the context object `posts` and `post` respectively. But for the GCBV `ListView`, you need to know that the default naming pattern is `<model>_list`; since our model is `Post`, the context object will be `post_list`. The updated code looks as follows:

Code

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

{% block content %}
{% for post in post_list %}  <!-- new -->
<div class="post-entry">
 <h2><a href="{{ post.get_absolute_url }}">{{ post.title }}</a></h2>
```

```
 <p>{{ post.body }}</p>
</div>
{% endfor %}
{% endblock content %}
```

Do you want to change the context object name? You can! Django almost always provides a way to customize behavior. The attribute to update is called context_object_name. The update below would switch it back to `posts` as opposed to `post_list`. We won't implement this change in the book, but you are welcome to do so on your own.

Code

```
# blog/views.py
...
class BlogListView(ListView):
    model = Post
    template_name = "home.html"
    context_object_name = 'posts'  # Change the context object name to 'posts'
...
```

The context object name in a `DetailView` also has a default value of either the model name (so `post` in this case) or `object`. That means we could leave our current template as is.

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 %}
```

Or we could change `post` to `object` like this:

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 refresh an individual blog post page in your web browser, you'll see that both work. Which naming pattern you prefer is a personal preference. I like to use the model name, so we'll use `post` going forward in our detail view templates. If you don't like these built-in names for the template context object, you can override the existing context_object_name variable in your `DetailView`.

Mixins

If you were paying close attention, you noticed that the `context_object_name` attribute for ListView was part of MultipleObjectMixin whereas for DetailView it was part of SingleObjectMixin. Why the difference?

A mixin is a type of multiple inheritance where you can add specific functionality to a class by including additional classes. Mixins are used in class-based views (CBVs) and generic class-based views (GCBVs) to add reusable behaviors or functionality without duplicating code. But you need to know what you're doing to use them effectively.

Covering Django mixins fully is beyond the scope of a beginner book, but it is worth briefly explaining how the inheritance structure works. To do this, we will rely on Classy Class-Based Views, a website dedicated to describing the full methods and attributes for each Django GCBV. It is an invaluable resource if you decide to work with GCBVs. Looking at the entry for ListView, you can see its Ancestors listed at the top of the page. The hierarchy of classes used by `ListView` starts from the bottom.

1. ListView
2. MultipleObjectTemplateResponseMixin
3. TemplateResponseMixin
4. BaseListView
5. MultipleObjectMixin
6. ContextMixin
7. View

The base class, View, is used by all class-based views. You don't need to memorize all these mixins to use GCBVs. But over time as attempt to customize their behavior, you will likely find yourself looking for the specific attribute or method that needs to be overridden.

A proponent of function-based views would say that this is ridiculous; you should have all the code you are using visible. A fan of class-based views would counter that it is ridiculous to write the same boilerplate code repeatedly when you only want to change one or two lines. I fall in the latter camp, but you can certainly understand why some very experienced Django developers prefer function-based views.

CreateView

We now need a view where users can create a new blog post. CreateView is a generic class-based view designed for exactly this purpose. Here is the code we need in our views file.

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"]
```

At the top we import `CreateView` from the `django.views.generic.edit` module. Then we create a class, `BlogCreateView`, that extends `CreateView`, specifying the model, `Post`, the template name, `post_new.html`, and the database fields we want visible on the form, which are `title`, `author`, and `body`. That's it! The function-based version of this view is much, much longer.

Then, update the URLs file to add a URL route for the create page.

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. We import the new view, `BlogCreateView` and set the new path. It is located at `post/new/`, uses the view `BlogCreateView`, and has a URL name of `post_new`.

The only thing left is our template. Make a new 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 }}
  <input type="submit" value="Save">
</form>
{% endblock content %}
```

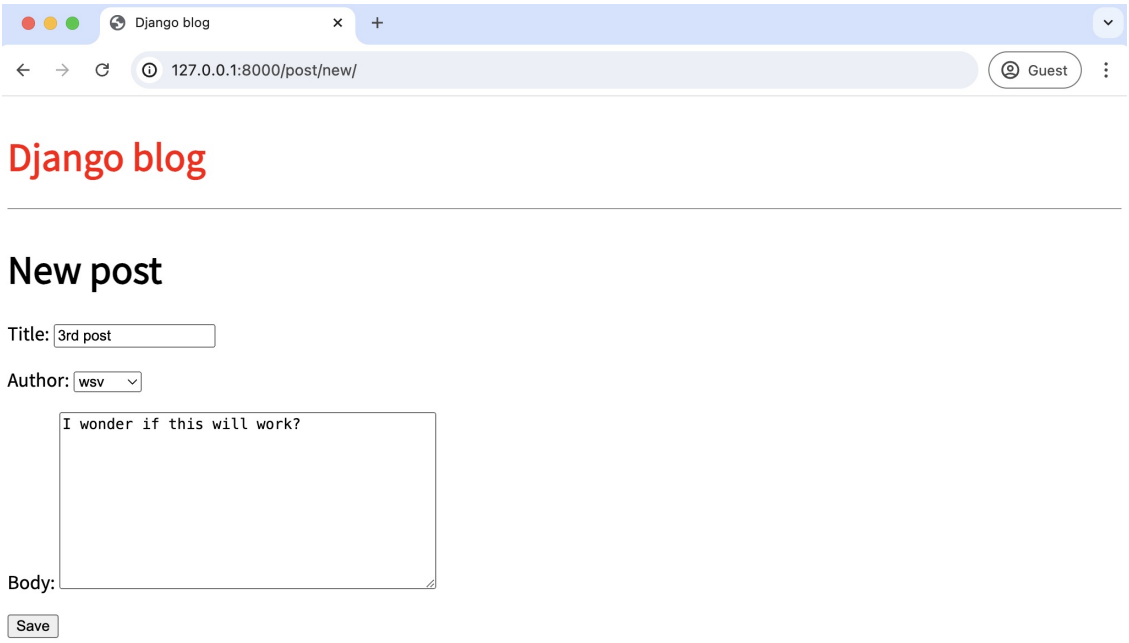
Let's break down what we've done:

- 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 }}` to render the specified fields.
- 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 create a new page at `http://127.0.0.1:8000/post/new`.

Blog New Page

Try to create a new blog post and submit it by clicking the “Save” button.



Blog Third Post Save

Upon completion, it will redirect to a detail page with the post at `http://127.0.0.1:8000/post/3/`. Success!

Blog Third Post Page

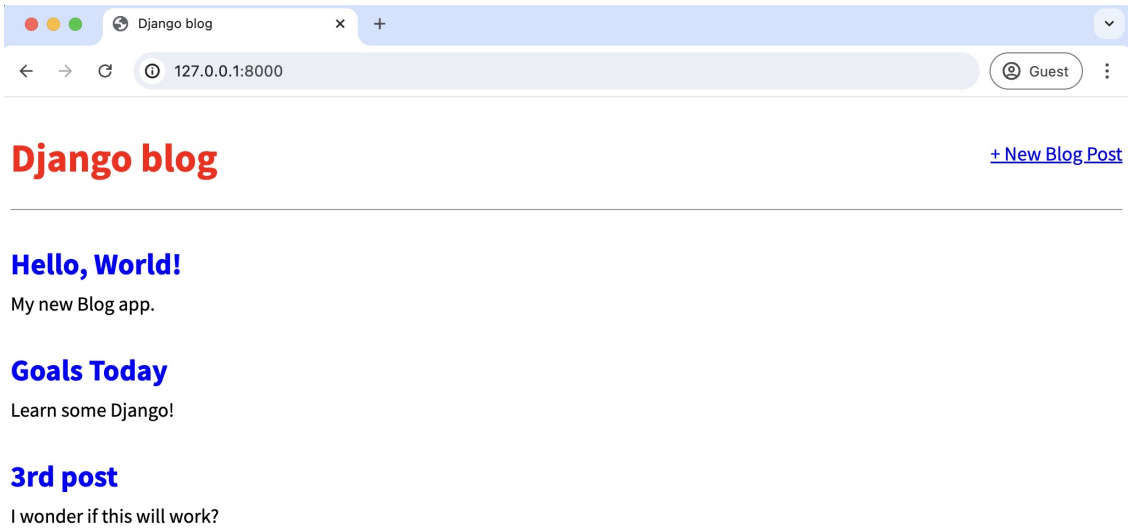
Rather than making a user guess where the create new post page is, let's add a link to our base template. It will take the form `<a href="{% url 'post_new' %}"></a>` where `post_new` is the name for our URL.

Your updated `templates/base.html` 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>
```

Navigate to the homepage and the “+ New Blog Post” link in the upper right-hand corner will be visible.



Homepage with New Button

Clicking it will redirect to the create new post page at `http://127.0.0.1:8000/post/new/`.

If we were to create this functionality using function-based views, we’d need a dedicated `blog/forms.py` file and a much longer view explaining how to handle the form, validate the data, and save it to the database. By using `CreateView` instead, we rely on built-in code to handle all these issues.

UpdateView

Our next task is to add a page with a form for editing existing blog posts. It turns out that the generic class-based view, UpdateView, is designed explicitly for this. We will add it to our Blog, and the pattern of adding a new view, then a new URL path, and finally, a new template should feel familiar.

Let’s begin with the view. At the top of the `blog/views.py` file import `UpdateView` and then create a new class, `BlogUpdateView`, that extends it. Just as with CreateView, we only have to define three things: the model, template name, and fields we want displayed on the form.

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 that we are not including “author” in the `fields` because we assume that the author of a post will not change during editing. We only want the title and body text to be available to update.

Next we need a new URL path for this update view. Update `blog/urls.py` by importing `BlogUpdateView` at the top and then create a new route with the URL pattern of `/post/pk/edit`. It will use `BlogUpdateView` as the view and have a URL named `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"),
]
```

The third step is adding a new template for the update page. We know from our view that it should have the name `post_edit.html`, so add that file to your templates directory. It should have the following code.

Code

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

{% block content %}
<h1>Edit post</h1>
<form action="" method="post">{% csrf_token %}
  {{ form }}
  <input type="submit" value="Update">
</form>
{% endblock content %}
```

At the top, we use `extends` on the base template, `base.html`, and then sandwich this page's content between the `content` blocks. We again use HTML `<form> </form>` tags, Django's `csrf_token` for security, and give it the value "Update" on the submit button.

We could go directly to the URL for a specific post, for example, `127.0.0.1:8000/post/1/edit/` for the first blog post. However, a better approach is to add a link to the individual blog page in the `post_detail.html` template.

Code

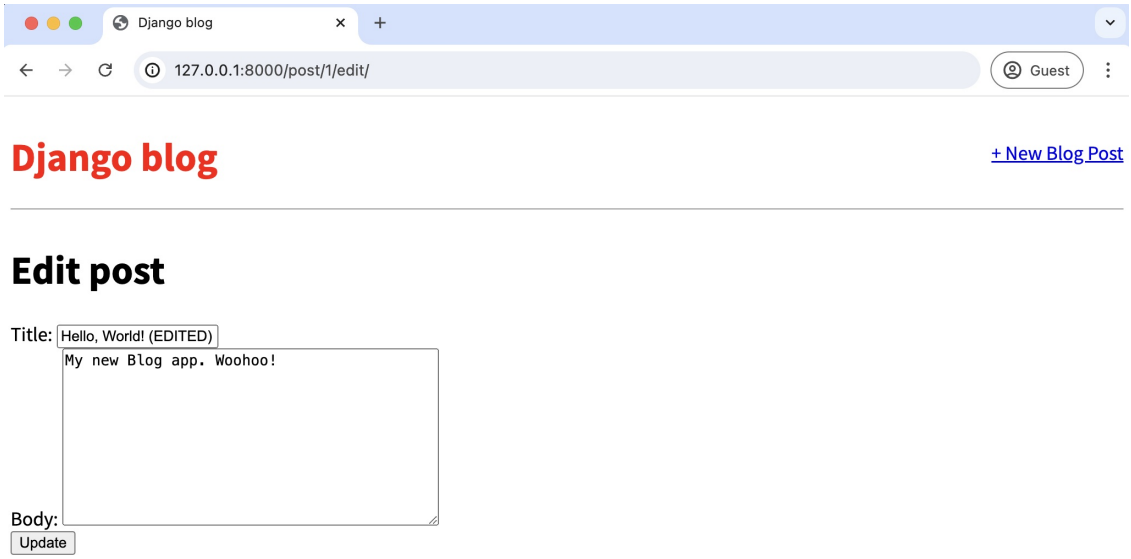
```
<!-- templates/post_detail.html -->
{% extends "base.html" %}

{% 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 `{% 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`.

Now, if you click on a blog entry, you'll see our new `+ Edit Blog Post` hyperlink.

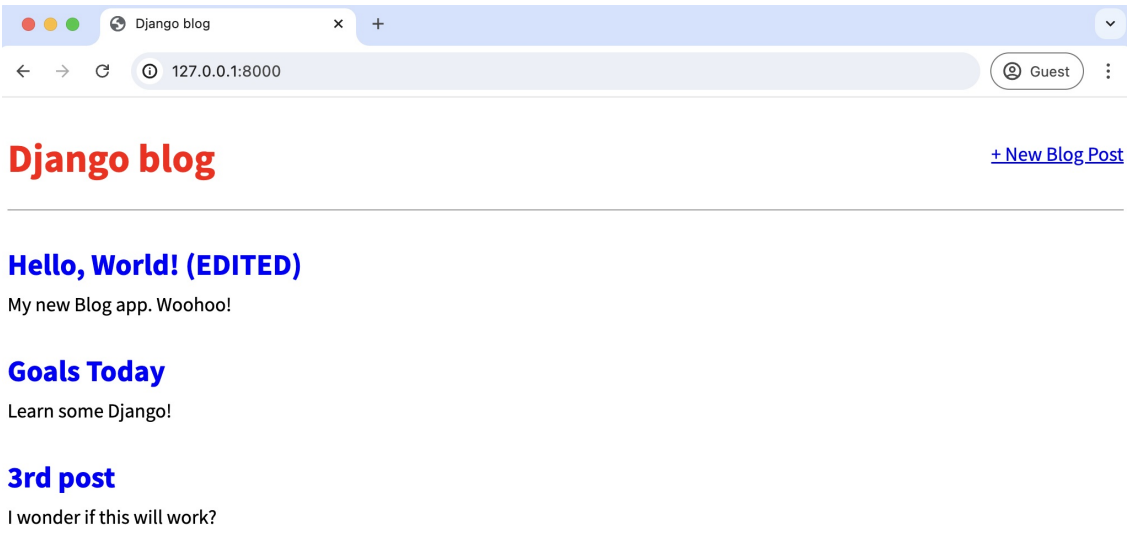
Blog Page with Edit Button

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…



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.



Blog Homepage with Edited Post

DeleteView

The final feature to add in this chapter is the ability to delete blog posts. We'll use yet another generic class-based view, DeleteView, to do this and create the necessary view, URL, and template.

To begin, update the `blog/views.py` file by importing `DeleteView` and reverse_lazy at the top and then create a new view that subclasses `DeleteView`. 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 URL dispatcher until the moment it is needed, not when our class `BlogDeleteView` is being evaluated.

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 BlogDeleteView(DeleteView):  # new
    model = Post
    template_name = "post_delete.html"
    success_url = reverse_lazy("home")
```

DeleteView requires a model, template name, and success URL. We have supplied all three here. Remember, *after* a blog post is deleted, we have to *redirect* the user somewhere. In this case, that is our homepage at the URL name of "home".

An astute reader might notice that both CreateView and UpdateView also have redirects yet we did not have to specify a `success_url`. Why is this? If available, Django, by default, will use the `get_absolute_url()` on the model object, a handy feature, but you would only know about it from reading a book like this or deeply reading the docs. More likely, with experience, you have used these GCBVs before and vaguely recall something about the redirect, consult the docs on model forms and then implement. There is no way to memorize everything you need to know as a Django developer; instead, with time, you'll start to see the broad patterns and be able to look up the documentation for implementation details.

With our view complete, we can turn to the URL. It's a similar pattern where we import the view, `BlogDeleteView`, set a URL path, specify the view, and include a URL name. A convention is to add `/delete/` to your URL path as we've done here.

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"),
]
```

Lastly, we need to add a template file confirming the user's wish to delete the blog post. 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 %}
```

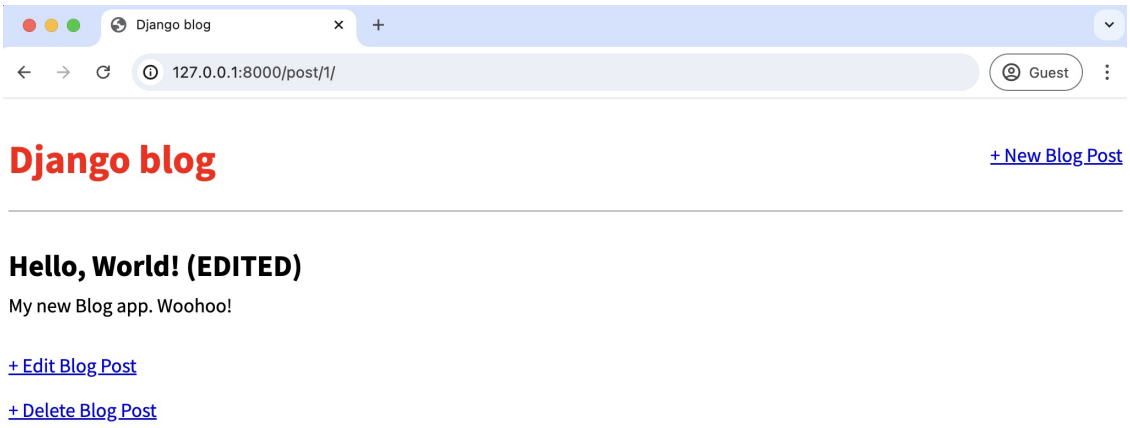
Note we are using `post.title` here to display the title of our blog post. We could use `object.title` as it is also provided by `DetailView`.

We can add a link to delete blog posts on the 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>
  <p><a href="{% url 'post_edit' post.pk %}">+ Edit Blog Post</a></p>
  <!-- new HTML below here... -->
  <p><a href="{% url 'post_delete' post.pk %}">+ Delete Blog Post</a></p>
</div>
{% endblock content %}
```

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.

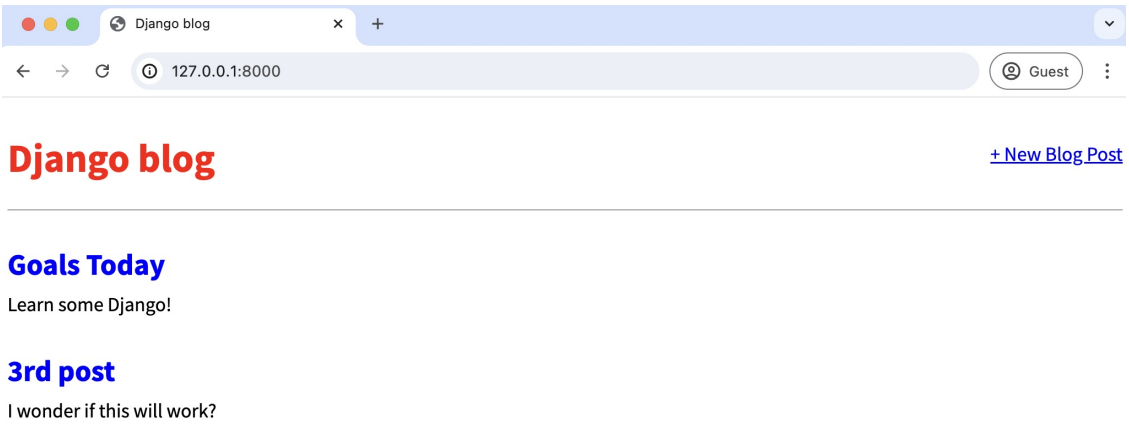


Blog Delete Post

Clicking on the link takes us to the delete page for the blog post, which displays the blog post's title.

Blog Delete Post Page

If you click the "Confirm" button, it redirects you to the homepage where the blog post has been deleted!



Homepage with Post Deleted

Tests

It's 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, such as for the templates, but at least we have some coverage of 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. 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

One major area still needs to be added to our *Blog* application: user accounts. We have an `author` field but no way, currently, for a user to sign up, log in, log out, and so on. Implementing user authentication from scratch is famously hard and not advised! It is one area where you really don't want to make a mistake because of all the resulting security implications. Fortunately, Django has a powerful, battle-tested, 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 is a URL pattern for the auth system, a login template, and a minor update to our `django_project/settings.py` file. Note that this code exists at the project-level and not within the `blog` app.

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 within `templates` 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
```

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 }}
  <button type="submit">Log In</button>
</form>
{% endblock content %}
```

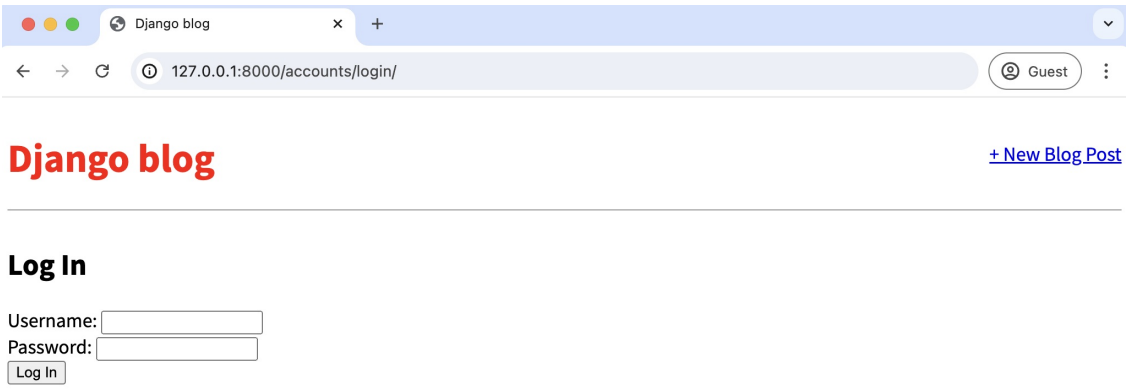
We're using HTML `<form></form>` tags and specifying the POST method since we're sending data to the server (we'd use GET if we were requesting data, such as in a search engine form). We add `{% csrf_token %}` for security concerns to prevent a CSRF Attack and then include a "submit" button.

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 automatically provided both for us. Thanks, Django!

Updated Homepage

Let's update our `base.html` template so we display a message to users regardless of whether they are logged in. We can use the is_authenticated attribute for this.

For now, 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 -->
...
<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.

Homepage Logged In

It worked! My superuser name is `wsv`, which I see on the page.

Log Out Link

One of the major changes to Django 5.0, as noted in the release notes, is the removal support for logging out via `GET` requests. Previously, you could add a logout link such as Log Out to a template file. But now a POST request via a form is required.

Add a logout button to the `base.html` file under the "Hi {{ user.username }}!` section.

Shell

```
<!-- templates/base.html-->
...
{% if user.is_authenticated %}
<p>Hi {{ user.username }}!</p>
```

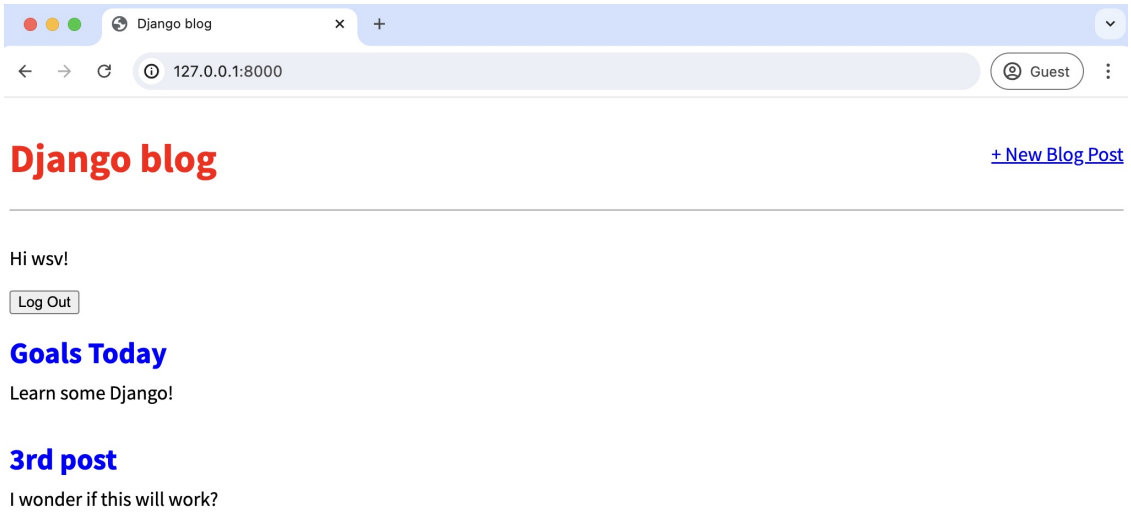
```
<form action="{% url 'logout' %}" method="post">
  {% csrf_token %}
  <button type="submit">Log Out</button>
</form>
{% else %}
...
```

The Django `auth` app provides us with the necessary view, so all we need to do is specify where to redirect a user when logging out. Update `django_project/settings.py` to provide a redirect link called `LOGOUT_REDIRECT_URL`. We can add it right next to our login redirect, so the bottom of the file should look as follows:

Code

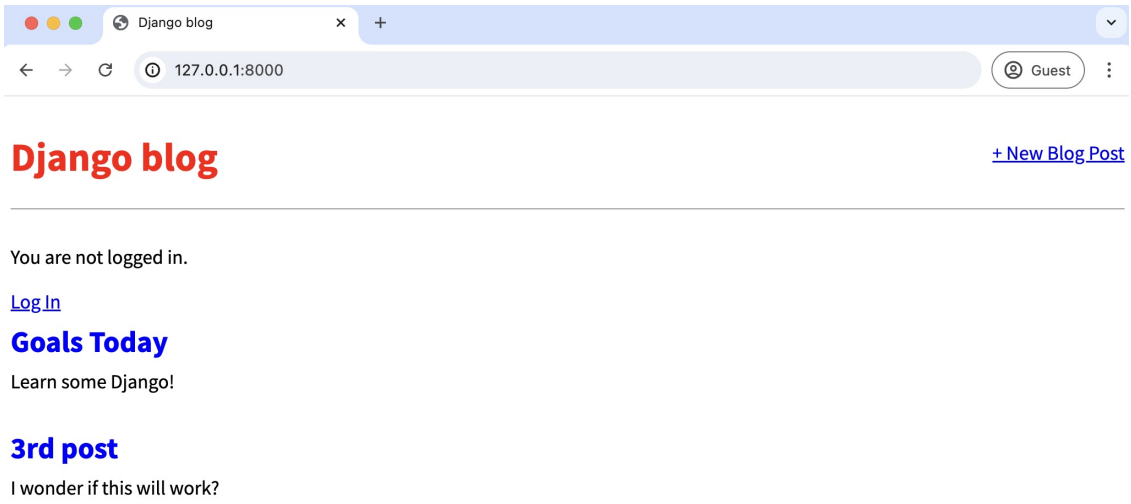
```
# django_project/settings.py
LOGIN_REDIRECT_URL = "home"
LOGOUT_REDIRECT_URL = "home"  # new
```

If you refresh the homepage, you'll see it now has a "log out" button for logged-in users.



Homepage Log Out Link

And clicking it takes you back to the homepage with a `login` link.



Homepage Logged Out

Try logging in and out several times with your user account.

Sign Up

Django does not provide a built-in view, URL, or template for signup. A common approach is to create a dedicated app, called `accounts` here, but also sometimes `users` in other projects, for this. That way, if we need to make further customizations to the user authentication process in the future, it is only contained in one place.

On the command line, stop the local server with `Control+c` and create a dedicated new app, `accounts`.

Shell

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

Then, add the new app to the `INSTALLED_APPS` setting in our `django_project/settings.py` file so it is registered with Django.

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/`.

For the view, Django has a built-in form class, UserCreationForm, that comes with three fields: `username`, `password1`, and `password2`. We can use it with `CreateView` to create our signup page. 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 specifying `UserCreationForm`, 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 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.

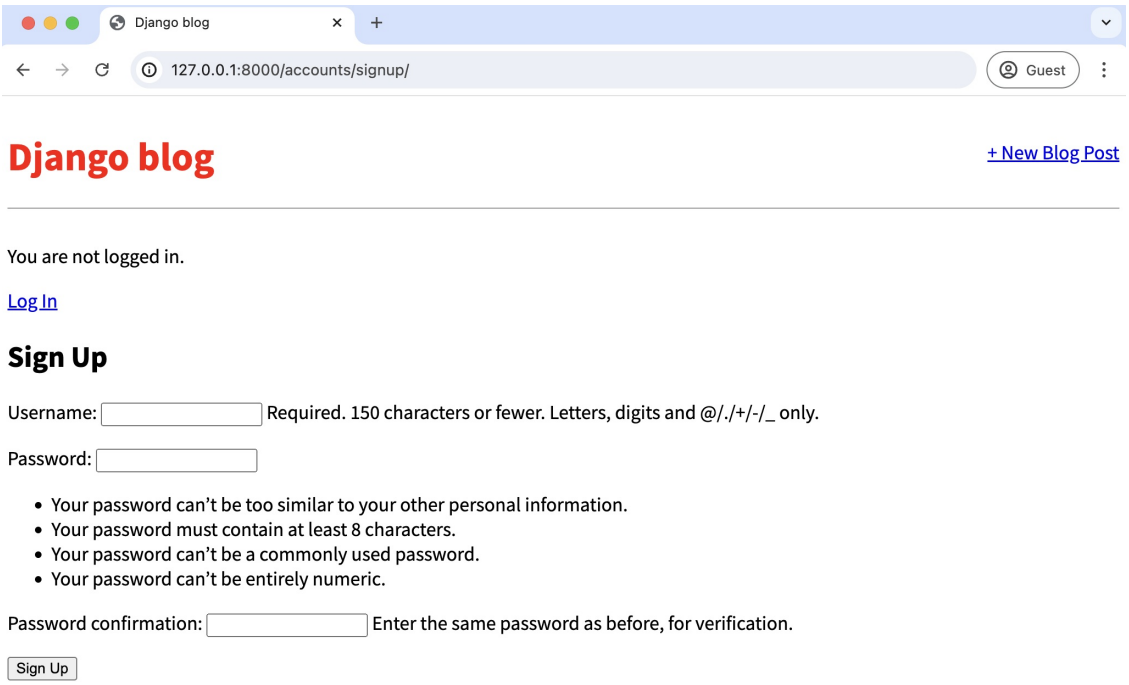
Create the file `templates/registration/signup.html` in your text editor and populate it with the code below.

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 with the added `as_p` to render it with paragraph tags, use the `csrf_token` for security, and include a submit button.

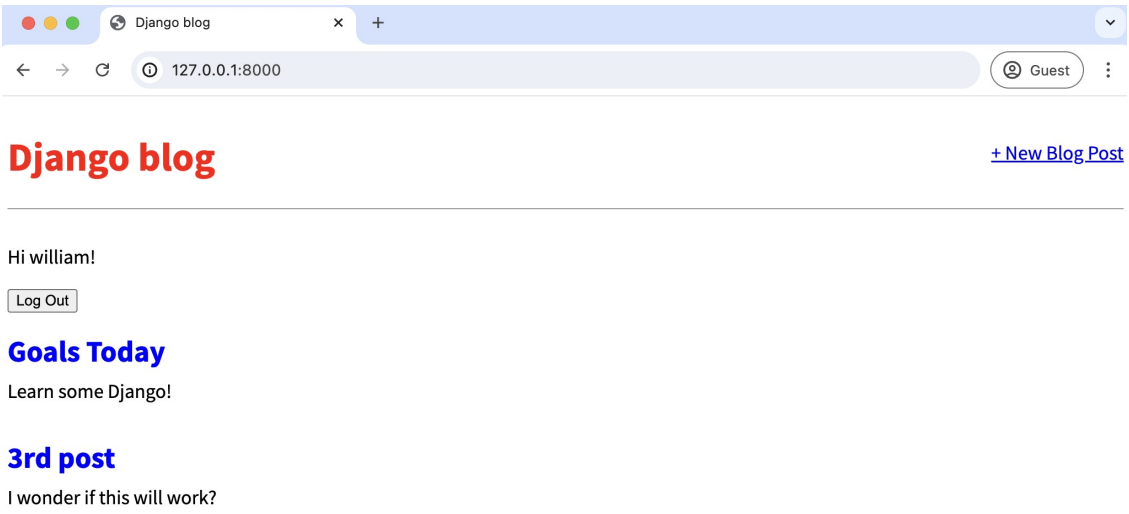
We're done! To test it out, start the local server with the `python manage.py runserver` command and navigate to `http://127.0.0.1:8000/accounts/signup/`.



Django Signup Page

Notice the large amount 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 created a new user called "william" and was redirected to the login page upon submission. After logging in successfully with my new username and password, I was redirected to the homepage with our personalized "Hi username" greeting.



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.

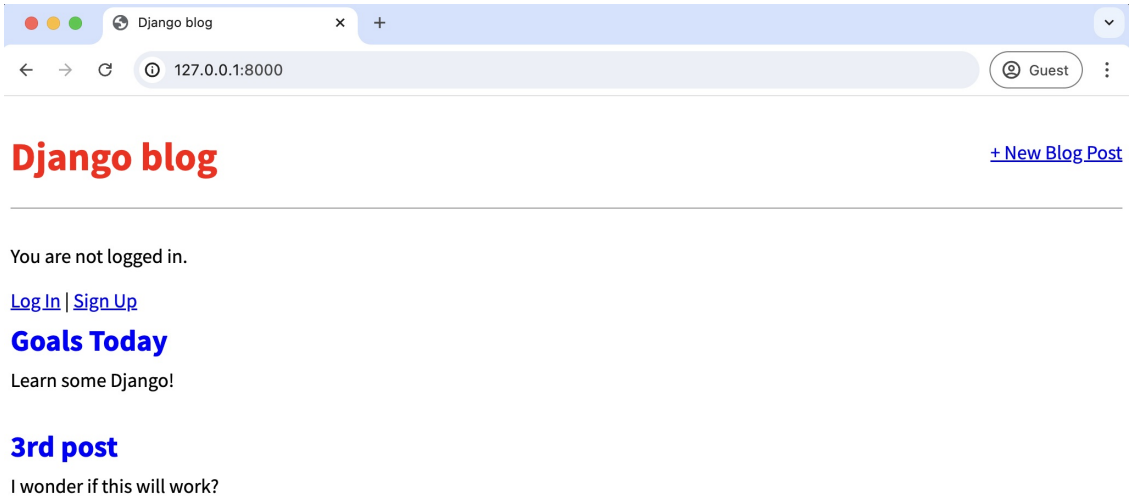
Add the link for "Sign Up" just below the existing link for "Log In" as follows:

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>
...
```

The signup link will be visible if you navigate to the logged-out homepage. Much better!



Signup Link

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; 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 a small amount of code, we have added a robust user authentication flow to our website: login, logout, and signup. 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. The *Blog* website is now complete. It is intentionally minimalist in design, but the combination of user authentication and full CRUD functionality is present in most websites.

In the next chapter, we will start the final project in the book: a *Newspaper* website that incorporates even more features.

Chapter 9: Newspaper Project

This chapter and the remaining portion of the book will focus on building a production-ready *Newspaper* website. The project choice is an homage to Django's roots as a newspaper CRM. It provides the opportunity to introduce even more features, including advanced user authentication and styling, complex data models, permissions, deployment, and more.

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~=5.0.0
(.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. Given how tightly connected the user model is to the rest of Django, it's important to wait until **after** we've created our new custom user model before doing so.

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, the `__pycache__` directory, and the SQLite database should not be included in Git, so create a project-level `.gitignore` file in your text editor.

.gitignore

```
.venv/
__pycache__/
db.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`; my username is `wsvincent`. Make sure to use your repo name and username using the command below.

Shell

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

All done!

User Profile vs 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 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 is to create a custom user model that extends `User` but allows for additional user information to be added. The Django documentation recommends using a custom user model when starting a new project as it makes later customization far easier than using the default `User` model. A custom user model can be created using AbstractUser, which behaves identically to the default `User` model but allows for customization.

It is possible to implement a hybrid approach combining a custom user model and a user profile model. But for this project, we will stick to a basic custom user model using `AbstractUser`.

AbstractUser

We can create a custom user model in four steps:

- update `django_project/settings.py`
- add a new `CustomUser` model
- add 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
```

Next update `accounts/models.py` with a new User model called `CustomUser`, which extends the existing `AbstractUser`. We will 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, you will see that 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 mistake 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 briefly, 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 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

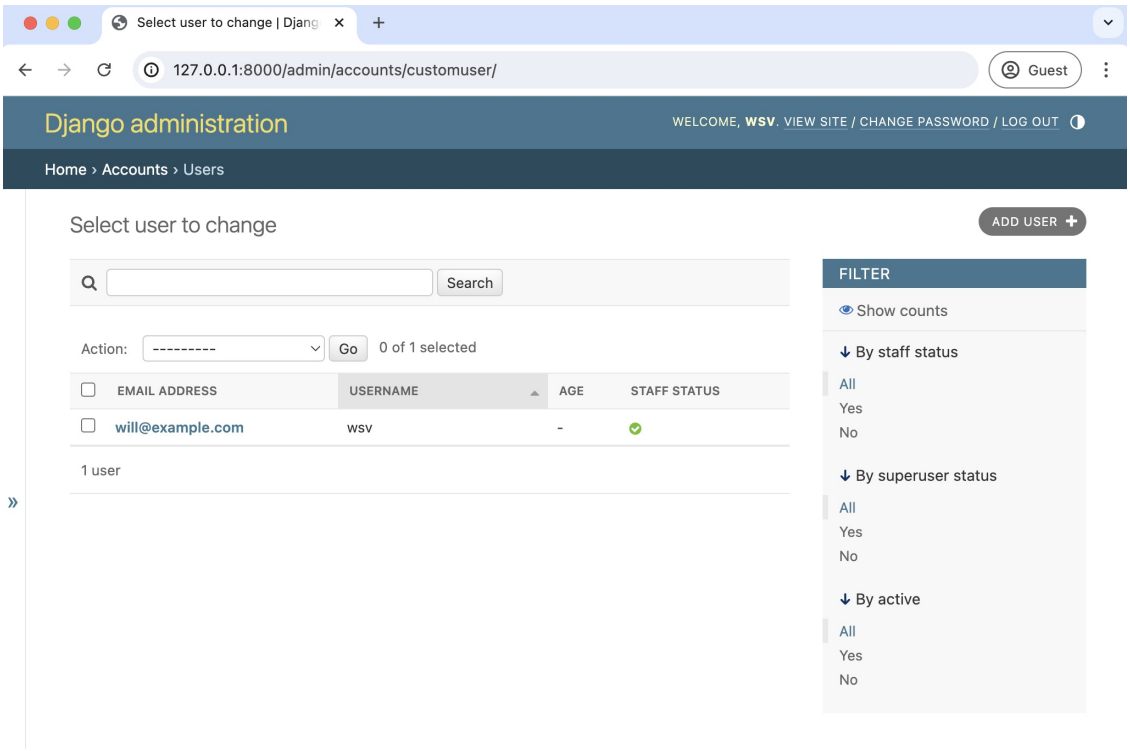
```
(.venv) $ python manage.py createsuperuser
```

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

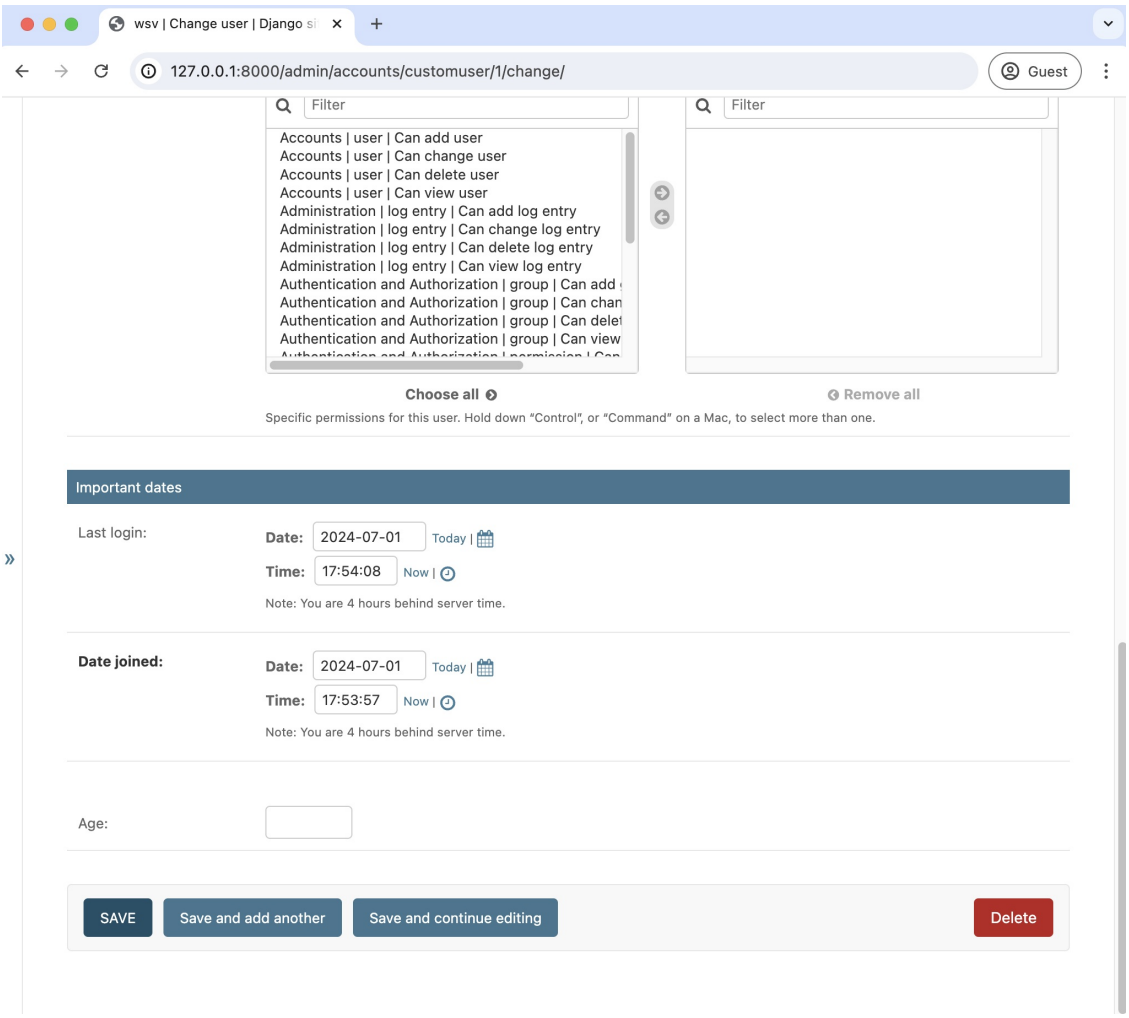
```
(.venv) $ python manage.py runserver
```

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: Email Address, Username, Age, and Staff Status. These were set in `list_display` in our `admin.py` file.



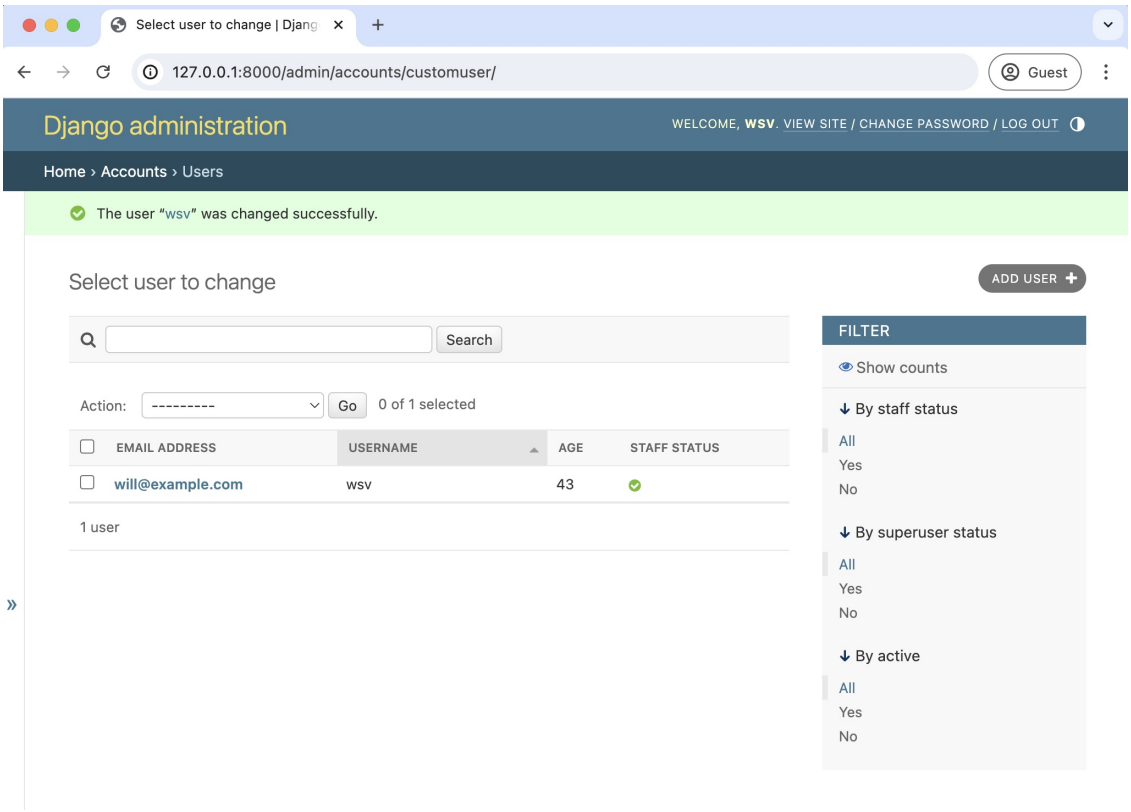
Admin Select User to Change

The `age` field is empty because we have yet to set it. However, you can set your age now because we set the `fieldsets` section. Click on the highlighted link for your superuser's email address to bring up the edit user interface. If you scroll to the bottom, you will see that we added the `age` field. Go ahead and enter your age. Then click on "Save."



Admin Edit Age

It will redirect back to the main Users page listing our superuser. Note that the age field is now updated.



Admin Updated Age

Tests

It is a good idea to add tests every time we make code changes that alter core functionality. 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 several 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. We also import `TestCase` since these tests touch the database.

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 adding a custom user model and an `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 10: 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 must 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. For example, the structure `accounts/templates/accounts/home.html` would be needed for a `home.html` template within the `accounts` app. However, a single project-level `templates` directory approach is cleaner and scales better, so that's what we'll use.

Create a new project-level `templates` directory and a `registration` directory within 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 %}
<p>Hi {{ user.username }}!</p>
```

```
<form action="{% url 'logout' %}" method="post">
  {% csrf_token %}
  <button type="submit">Log Out</button>
</form>
{% 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 %}
```

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 }}
  <button type="submit">Log In</button>
</form>
{% 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. The related URLs and views are still to go.

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 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.

Homepage Logged In

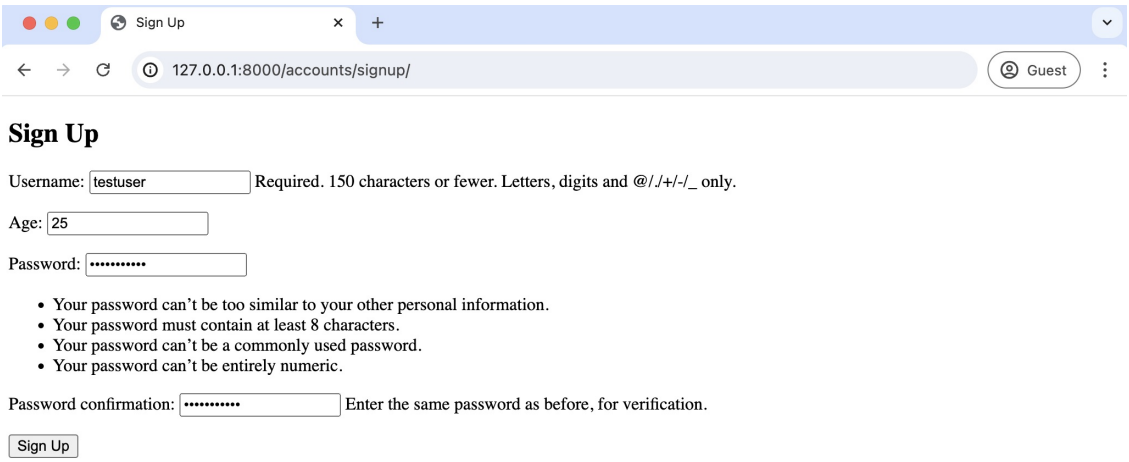
We logged in to the admin in the previous chapter, so you should see a personalized greeting here. Click on the “Log Out” link.

Homepage Logged Out

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. 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`.



Signup Page

After successfully submitting the form, you'll be redirected to the login page. Log in with your new user, and you'll be redirected again 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 -->
<form action="{% url 'logout' %}" method="post">
  {% csrf_token %}
  <button type="submit">Log Out</button>
</form>
{% 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 %}
```

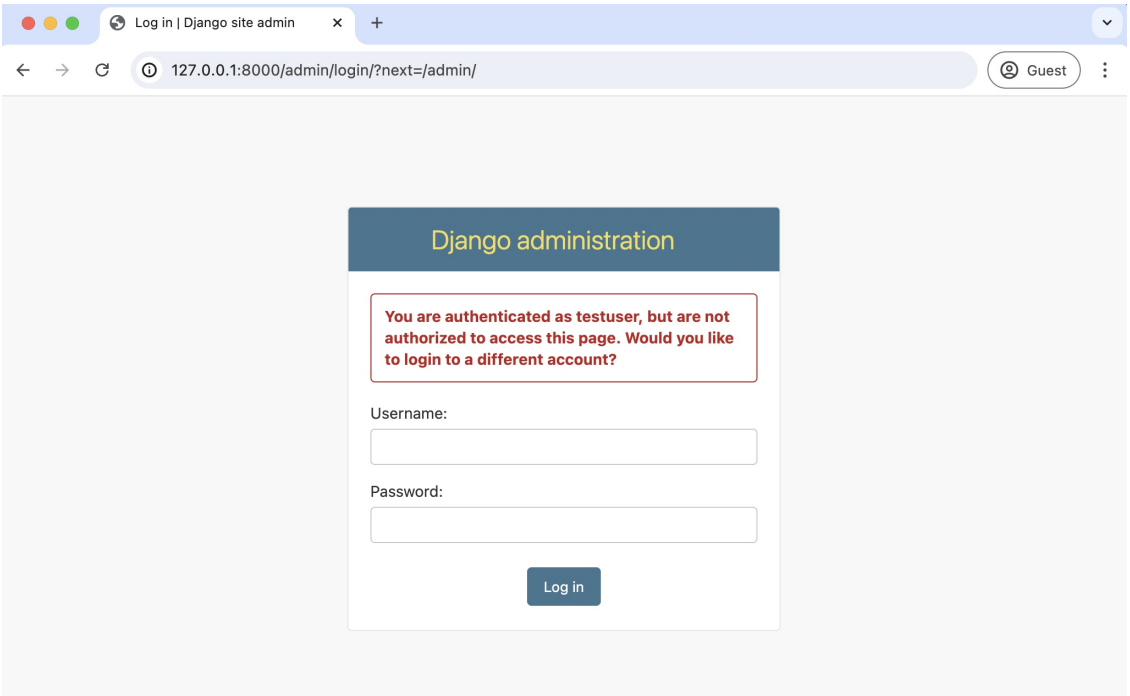
Save the file and refresh the homepage.

Homepage for testuser

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.

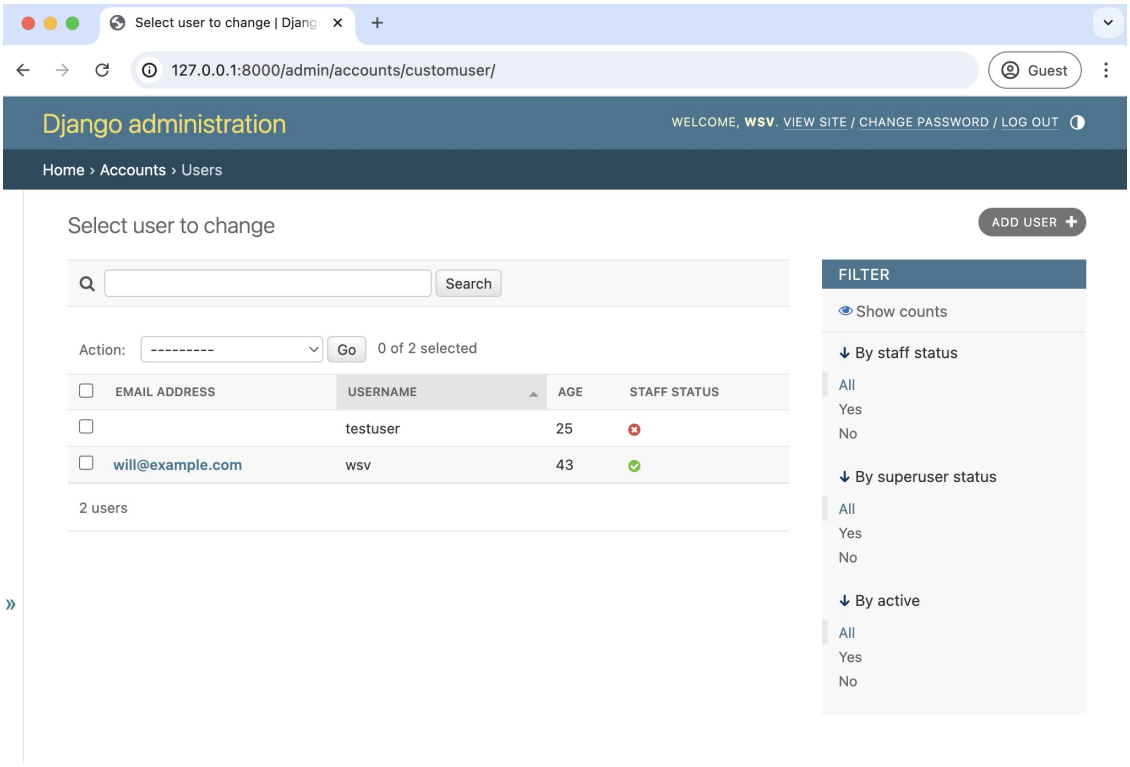


Wrong Admin Login

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.

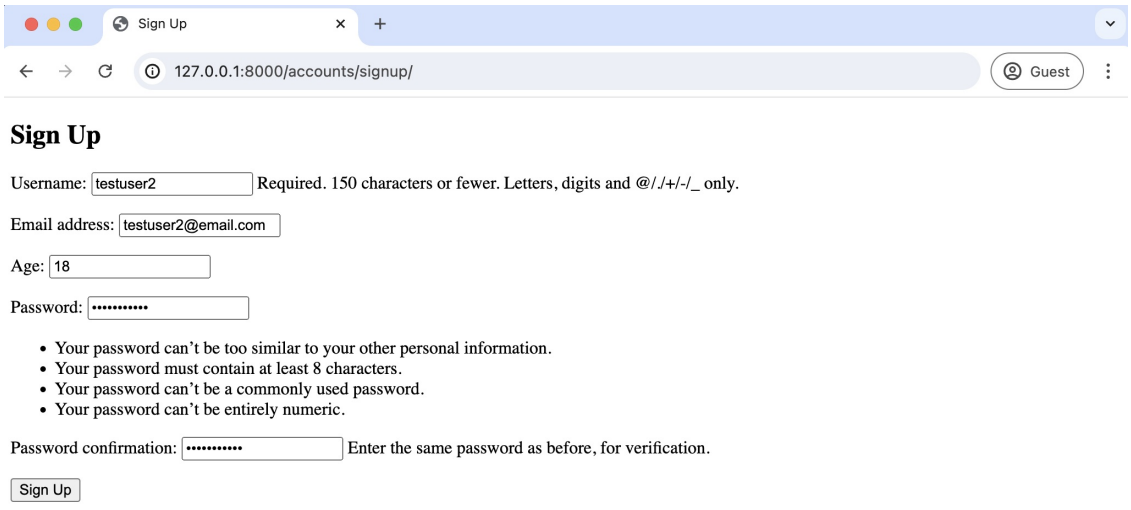
Currently, in `accounts/forms.py` under `fields`, we're using `Meta.fields`, which displays the default settings of username/password, and then we explicitly added our custom field, `age`, too. But we can also explicitly set which fields we want to be displayed, so let's update it to ask for a username/email/age/password 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
```

Log out of your superuser account and 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`.

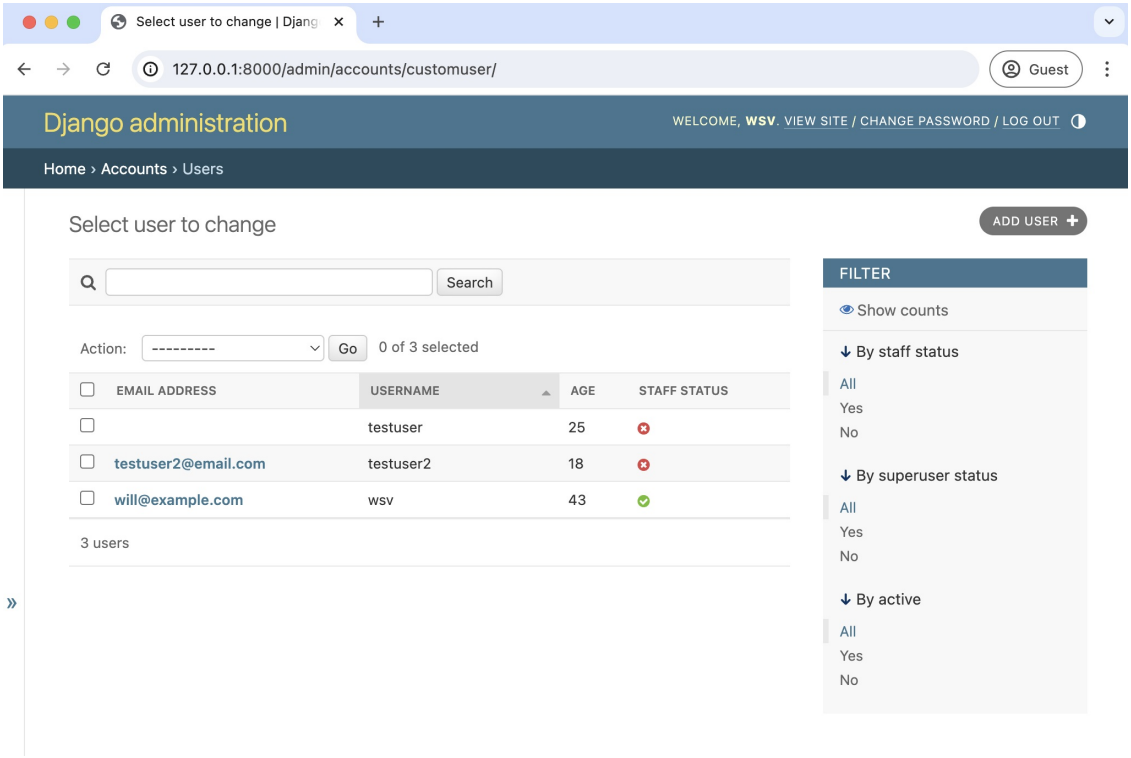


New Signup Page

Click the "Sign Up" button and continue to log in. You'll see a personalized greeting on the homepage.

testuser2 Homepage Greeting

Switch back to the admin page, log in using our superuser account, and all three users are on display.



Three Users in the Admin

Django's user authentication flow requires some 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 it.

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 11: Bootstrap

Web development requires many skills. Not only do you have to program the website correctly, but users also expect it to look good. Adding all the necessary HTML/CSS for a beautiful site can be overwhelming when creating everything from scratch.

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. We'll use Bootstrap for our project, 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.

On the command line, use the `startapp` command to create our new `pages` app. If the server is still running, you must 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. The code should look like this 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 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-suited for many unit tests and automatic regression tests, so developers can be confident in the consistency of their projects.

Bootstrap

It's now 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 we'll use it 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

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">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>{% block title %}Newspaper App{% endblock title %}</title>
  <!-- Bootstrap CSS -->
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/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@5.3.3/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 5.3 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 and 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. If there are any formatting issues, you can view the official GitHub repository for reference.

Code

```
<!-- templates/base.html -->
...
<body>
  <nav class="navbar navbar-expand-lg bg-body-tertiary">
    <div class="container-fluid">
      <a class="navbar-brand" href="{% url 'home' %}">Newspaper</a>
      <button class="navbar-toggler" type="button"
        data-bs-toggle="collapse" data-bs-target="#navbarSupportedContent"
        aria-controls="navbarSupportedContent" aria-expanded="false"
        aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
      </button>
      <div class="collapse navbar-collapse" id="navbarSupportedContent">
        <ul class="navbar-nav me-auto mb-2 mb-lg-0">
          <li class="nav-item">
            <a class="nav-link active" aria-current="page" href="#">Home</a>
          </li>
          {% if user.is_authenticated %}
          <li><a href="#" class="nav-link px-2 link-dark">+ New</a></li>
        </ul>
        <div class="mr-auto">
          <ul class="navbar-nav">
            <li class="nav-item dropdown">
              <a class="nav-link dropdown-toggle" href="#" role="button"
                data-bs-toggle="dropdown" aria-expanded="false">
                {{ user.username }}
              </a>
              <ul class="dropdown-menu dropdown-menu-end">
                <li><a class="dropdown-item" href="{% url 'password_change' %}">
                  Change password</a></li>
                <li>
                  <hr class="dropdown-divider">
                </li>
                <li>
                  <form method="post" action="{% url 'logout' %}"
                    style="display:inline;">
                    {% csrf_token %}
                    <button type="submit" class="btn btn-link nav-link"
                      style="display:inline; cursor:pointer;">Logout</button>
                  </form>
                </li>
              </ul>
            </li>
            {% else %}
          </ul>
        </div>
        <div class="mr-auto">
```

```
          <form class="form d-flex">
            <a href="{% url 'login' %}" class="btn btn-outline-secondary">Log in</a>
            <a href="{% url 'signup' %}" class="btn btn-primary ms-2">Sign up</a>
          </form>
        </div>
        {% endif %}
      </div>
    </div>
  </nav>
  <main>
    <div class="container">
      {% block content %}
      {% endblock content %}
    </div>
  </main>
...
```

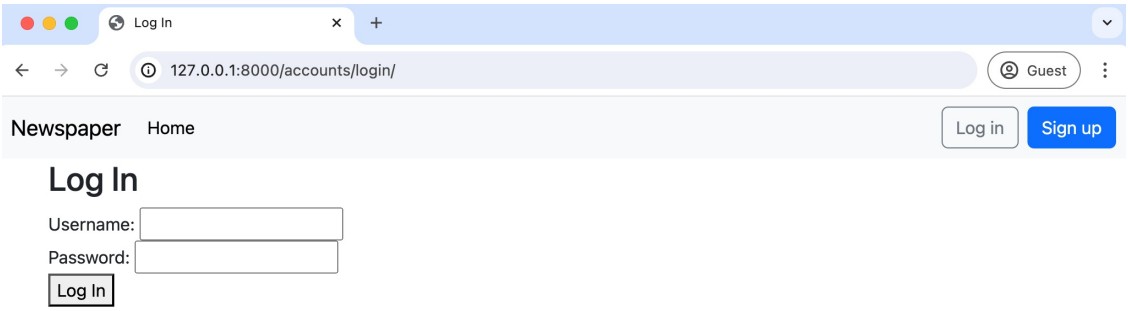
If you refresh the homepage at `http://127.0.0.1:8000/`, our new navbar has magically appeared! Note that there are no actual links yet for new articles "+ New"; a placeholder is represented in the code by `href="#"`. We will add that later on. Also, note that our logged-in username is now in the upper 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.



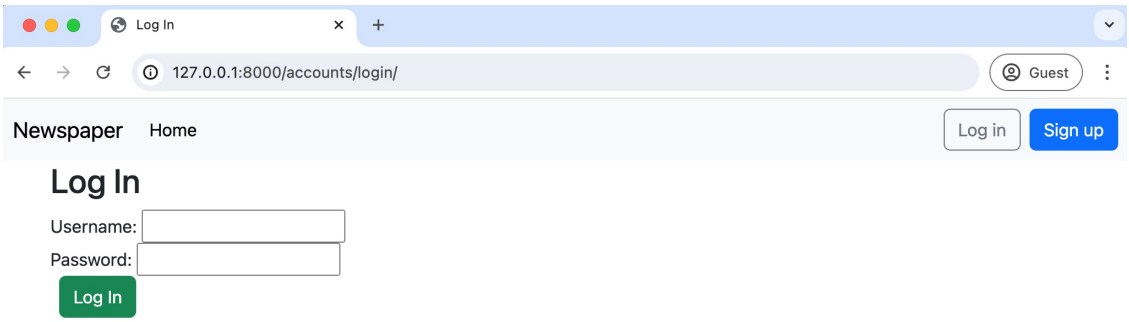
Bootstrap Login

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 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 }}
  <!-- new code here -->
  <button class="btn btn-success ms-2" type="submit">Log In</button>
  <!-- end new code -->
</form>
{% endblock content %}
```

Now refresh the page to see our new button in action.



Bootstrap Login with New Button

Signup Form

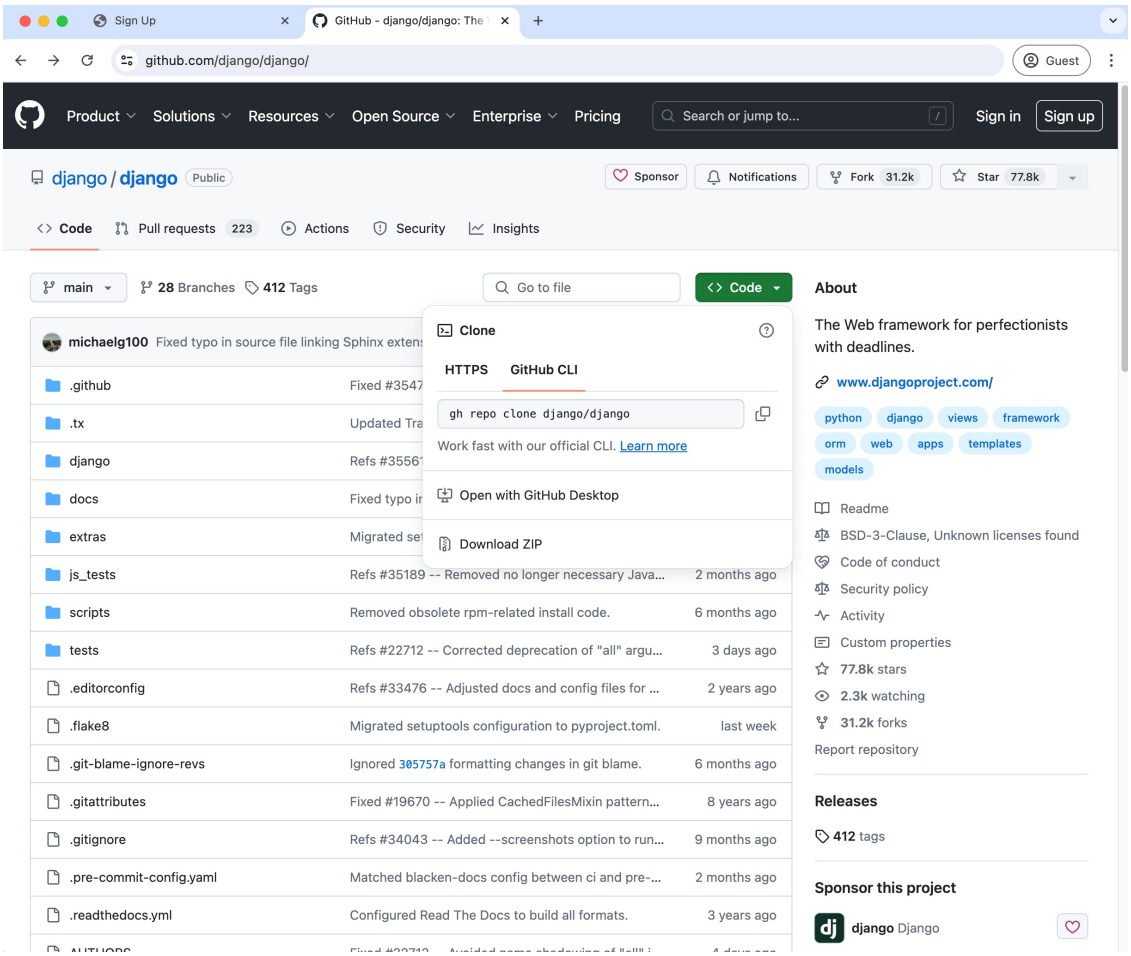
If you click on the link for “Sign Up” you’ll see that the page has Bootstrap stylings and distracting helper text. For example, after “Username” it says “Required. 150 characters or fewer. Letters, digits, and @/./+/-/_ only.” So, where did that text come from? Whenever something feels like “magic” in Django, rest assured that it is decidedly not. If you did not write the code, 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. Let’s navigate there now since we’ve been using a `code` folder on the desktop.

Shell

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

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

On the GitHub website, you’ll see a green “<> Code” button. Click it and select “GitHub CLI” to view the command line instructions to download the repo.



GitHub Django Repo

On the command line, type `gh repo clone django/django` to copy the Django source code onto your computer in a new directory called `django`.

Shell

```
$ gh repo clone django/django
```

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. 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 are valuable tools, 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, press the keys `command + shift + f` in the VS Code text editor to do a "find" search in all files. Type in a search for "150 characters or fewer" and you'll find the top link to the page for `django/contrib/auth/models.py`. The specific code is on line 350, 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. In the future, 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](https://github.com/django-crispy-forms/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](https://github.com/django-crispy-forms/crispy-bootstrap5).

Shell

```
(.venv) $ python -m pip install django-crispy-forms==2.2
(.venv) $ python -m pip install crispy-bootstrap5==2024.2
```

Add the new apps to our `INSTALLED_APPS` list in the `django_project/settings.py` file. As the number of apps starts to grow, it can be helpful to distinguish between "3rd party" apps and "local" apps. 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

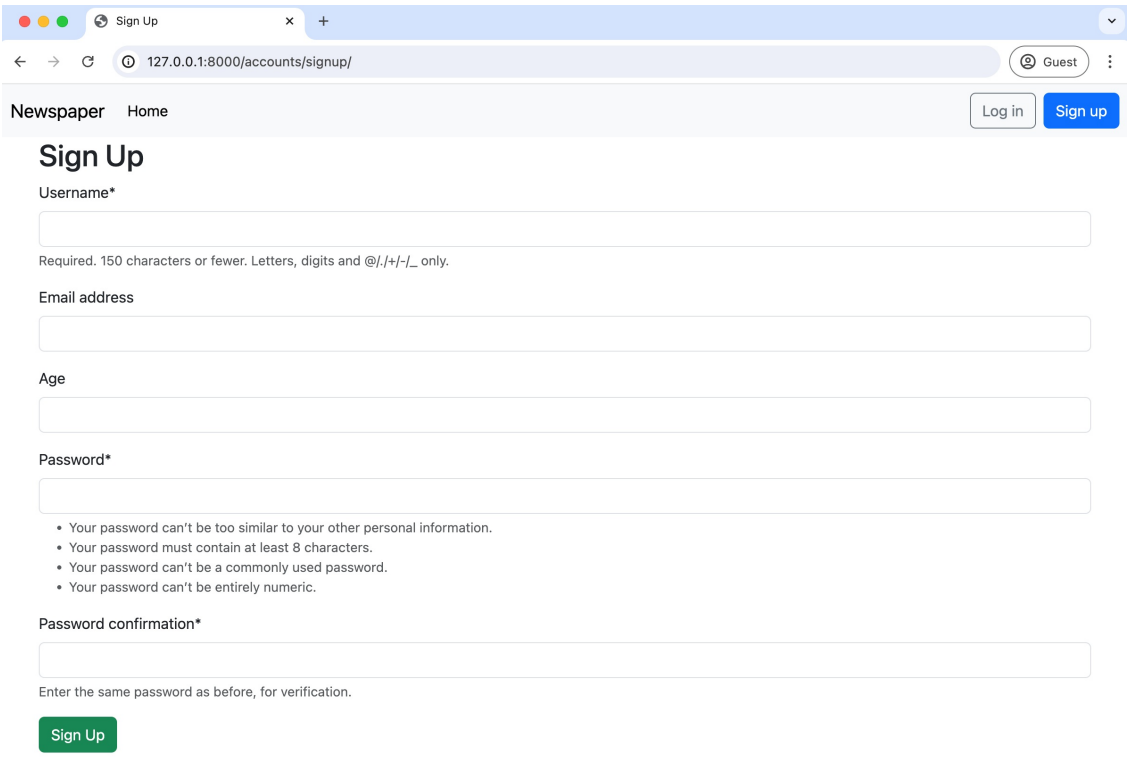
```
# django_project/settings.py
CRISPY_ALLOWED_TEMPLATE_PACKS = "bootstrap5"  # new
CRISPY_TEMPLATE_PACK = "bootstrap5"  # new
```

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 }}` 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.



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 ms-2" type="submit">Log In</button>
</form>
{% endblock content %}
```

Refresh the login page, and the update will be visible.

Crispy Login Page

Git and requirements.txt

We have now added several packages to our Django project so it is a good time to create a `requirements.txt` file.

Shell

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

At the moment, this is how mine looks:

Code

```
# requirements.txt
asgiref==3.8.1
black==24.4.2
click==8.1.7
crispy-bootstrap5==2024.2
Django==5.0.6
django-crispy-forms==2.2
mypy-extensions==1.0.0
packaging==24.1
pathspec==0.12.1
platformdirs==4.2.2
sqlparse==0.5.0
```

Then we can add a quick Git commit 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

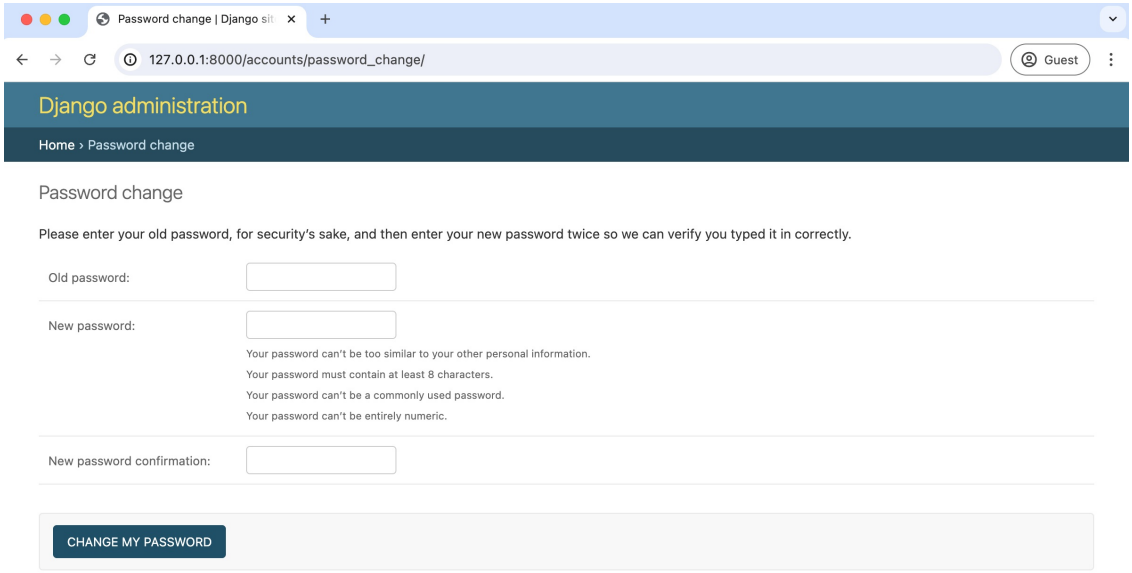
Our *Newspaper* app looks good. To improve the look of our forms, we added Bootstrap and Django Crispy Forms to our website. The last step of our user authentication flow is configuring password change and reset. Again, Django has taken care of the heavy lifting, requiring a minimal amount of code on our part.

Chapter 12: 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. We'll implement Django's built-in views and URLs for password changes and resets before customizing them with our own Bootstrap-powered templates.

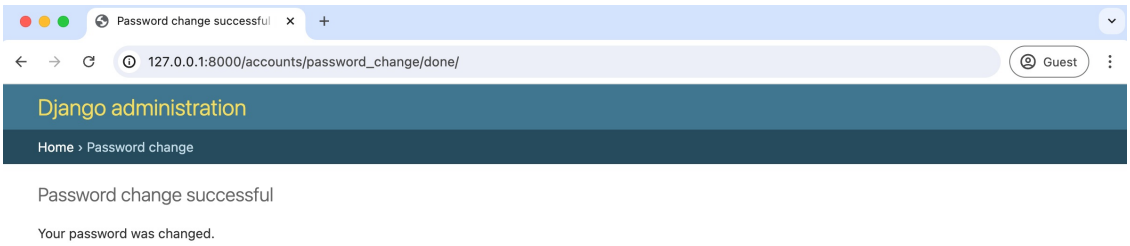
Password Change

Many websites allow users to change their passwords, 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/`.



Password Change

Enter 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.



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:

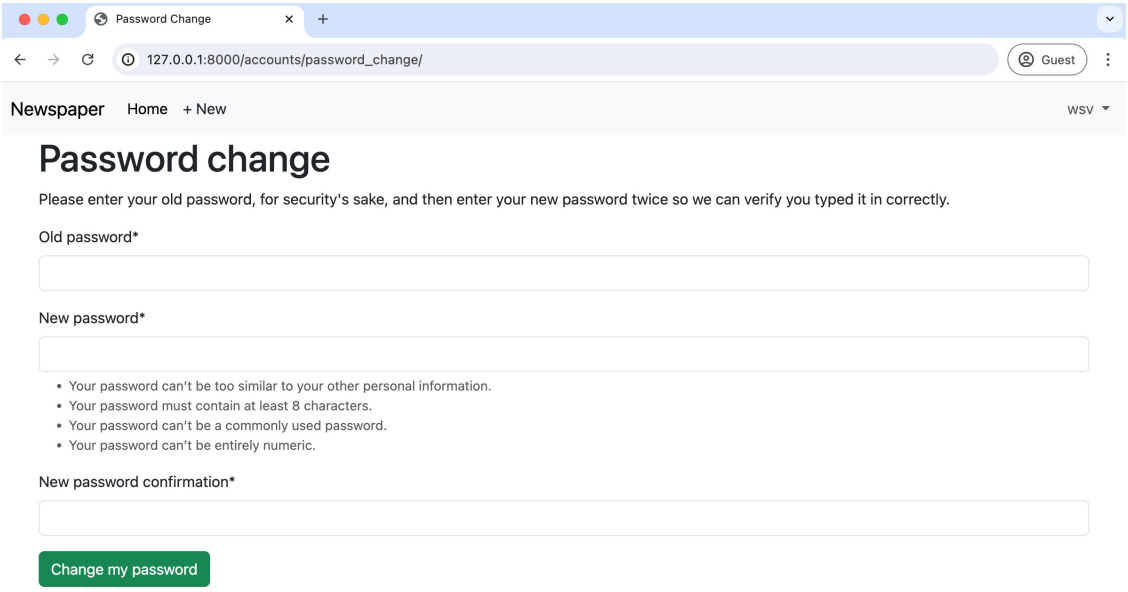
- `templates/registration/password_change_form.html`
- `templates/registration/password_change_done.html`

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.



New Password Change Form

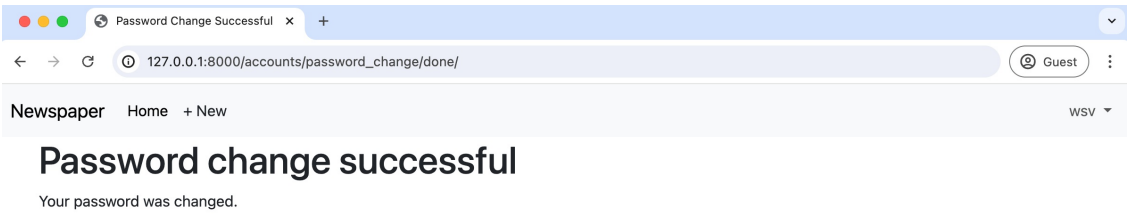
Next up is the `password_change_done` template. It also extends `base.html` and includes a new meta title; however, there's no form on the page, just new text.

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/`



New 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 typical 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 we will customize the templates to match the look and feel of the rest of our site.

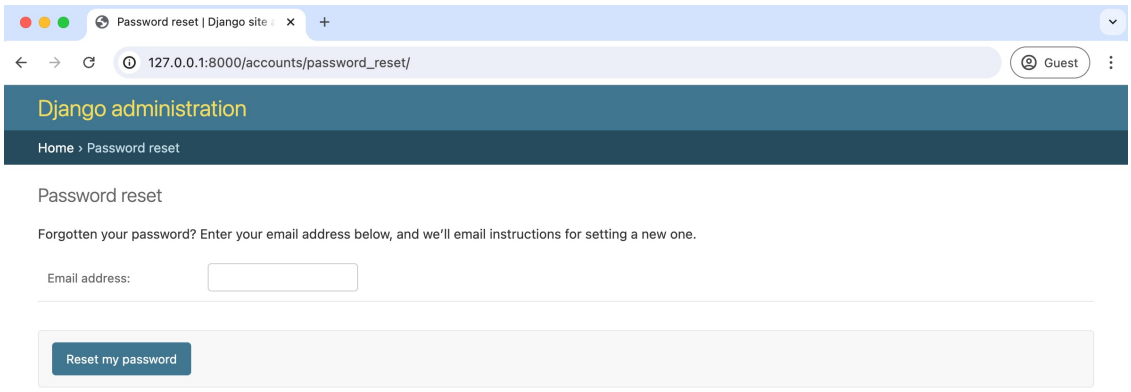
The only configuration required is telling Django how to send emails. After all, a user can only reset a password if they can access the email linked to the account. For testing purposes, we can rely on Django's console backend setting, which outputs the email text to our command-line console instead.

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
```

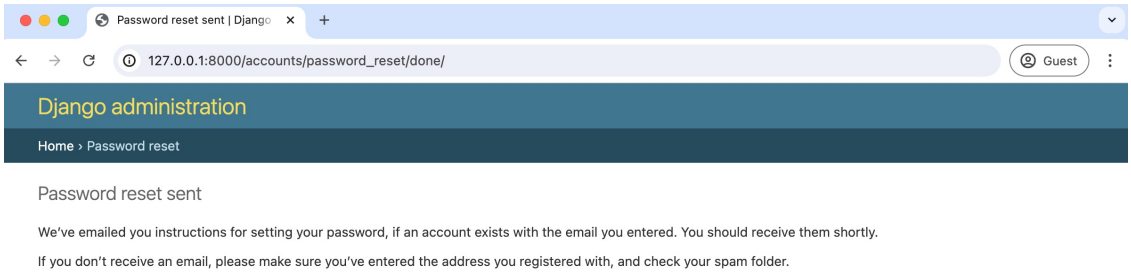
And we're all set! Django will take care of all the rest for us. Let's try it out. Navigate to `http://127.0.0.1:8000/accounts/password_reset/` to view the default password reset page.



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/`



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. In my console, I see the following:

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: Fri, 28 Jun 2024 15:36:30 -0000
Message-ID:
 <168235059067.94136.7639058137157368290@1.0.0.0.0....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


----------------------------------------------------------------------------
[11/Jun/2024 15:36:30] "POST /accounts/password_reset/ HTTP/1.1" 302 0
[11/Jun/2024 15:36:30] "GET /accounts/password_reset/done/ HTTP/1.1" 200 3014
```

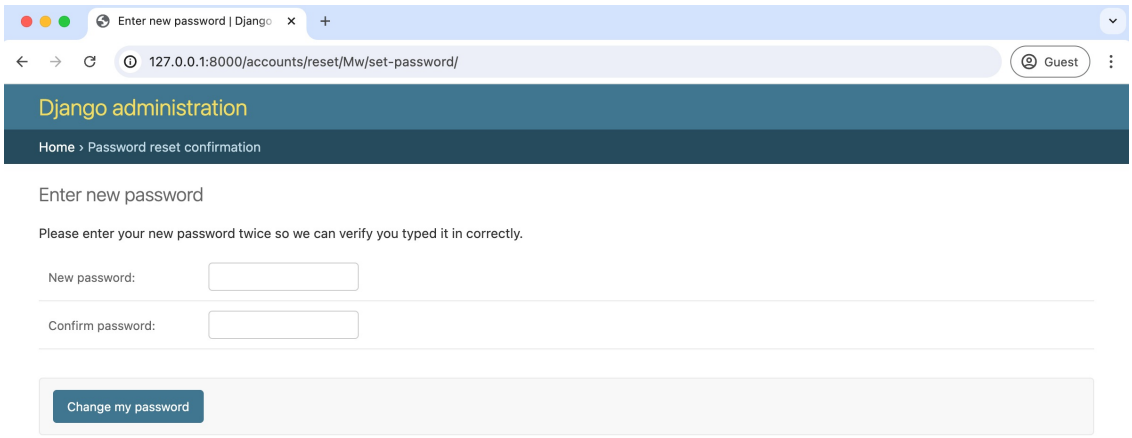
Your email text should be identical except for three lines:

- 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:

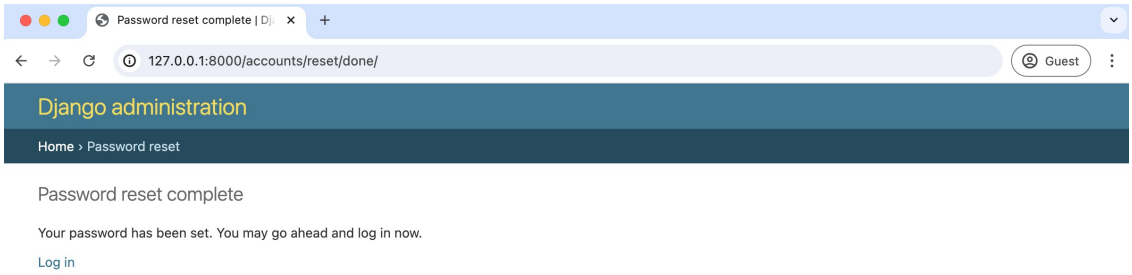
```
http://127.0.0.1:8000/accounts/reset/Mw/bn63cu-
b77b6590a2e38a75c4b2af244c40297e/
```

You’ll be redirected to the “Password reset confirmation” page.



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.



Default Password Reset Complete

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, four separate templates are 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 updated the submit button to green. At this point, updating these template pages should start to feel familiar.

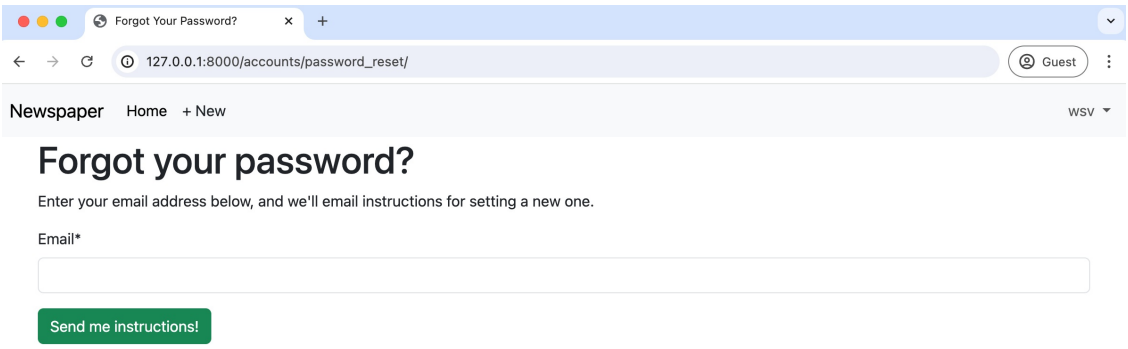
Code

```
<!-- 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.



New Password Reset

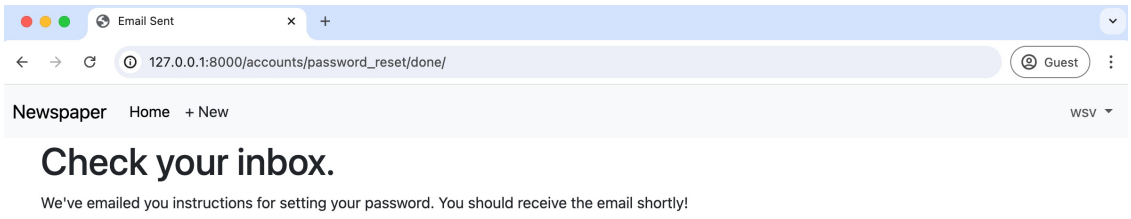
Now we can update the other three pages. Each involves extending `base.html`, setting a new meta title, and adding new content text. When a form is involved, we switch to loading and using crispy forms.

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/`.



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:

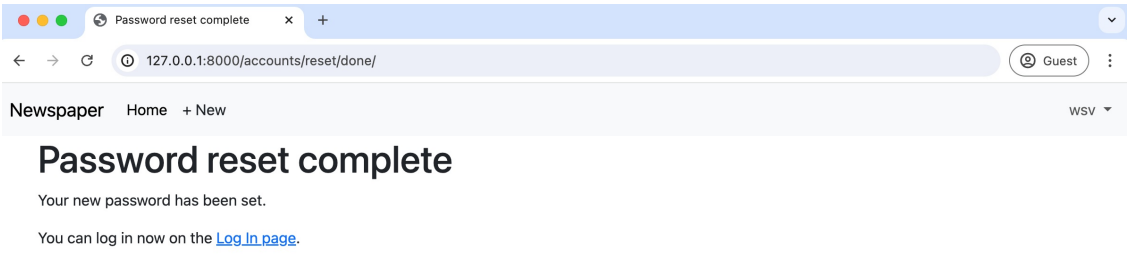
New Set Password

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/`.



New Password Reset Complete

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 ms-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.

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 13: Articles 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, `TIME_ZONE`, lower down in the settings 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
Migrations for 'articles':
  articles/migrations/0001_initial.py
    - Create model Article
(.venv) $ python manage.py migrate
Operations to perform:
  Apply all migrations: accounts, admin, articles, auth, contenttypes, sessions
Running migrations:
  Applying articles.0001_initial... OK
```

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

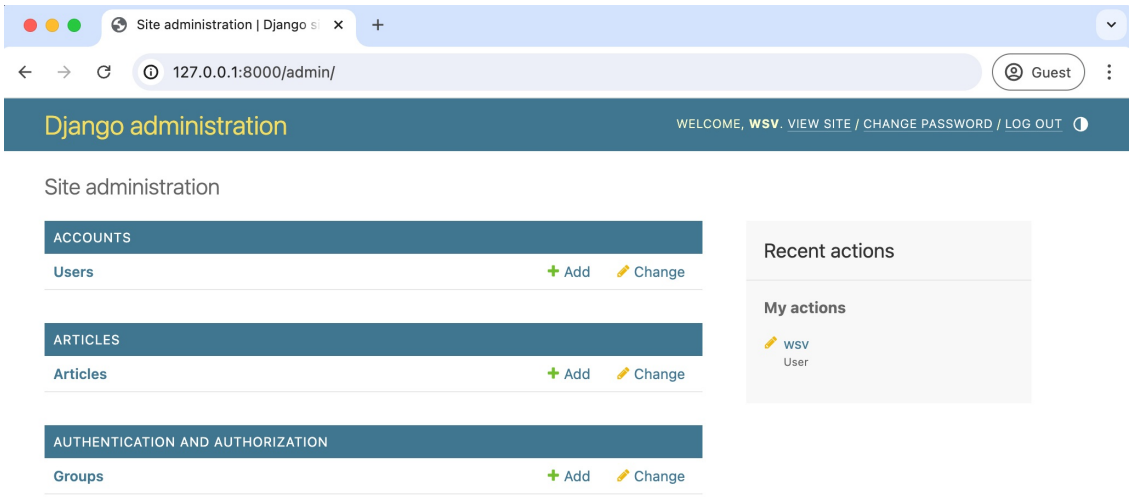
```
# articles/admin.py
from django.contrib import admin
from .models import Article

admin.site.register(Article)
```

Now, we start the server.

Shell

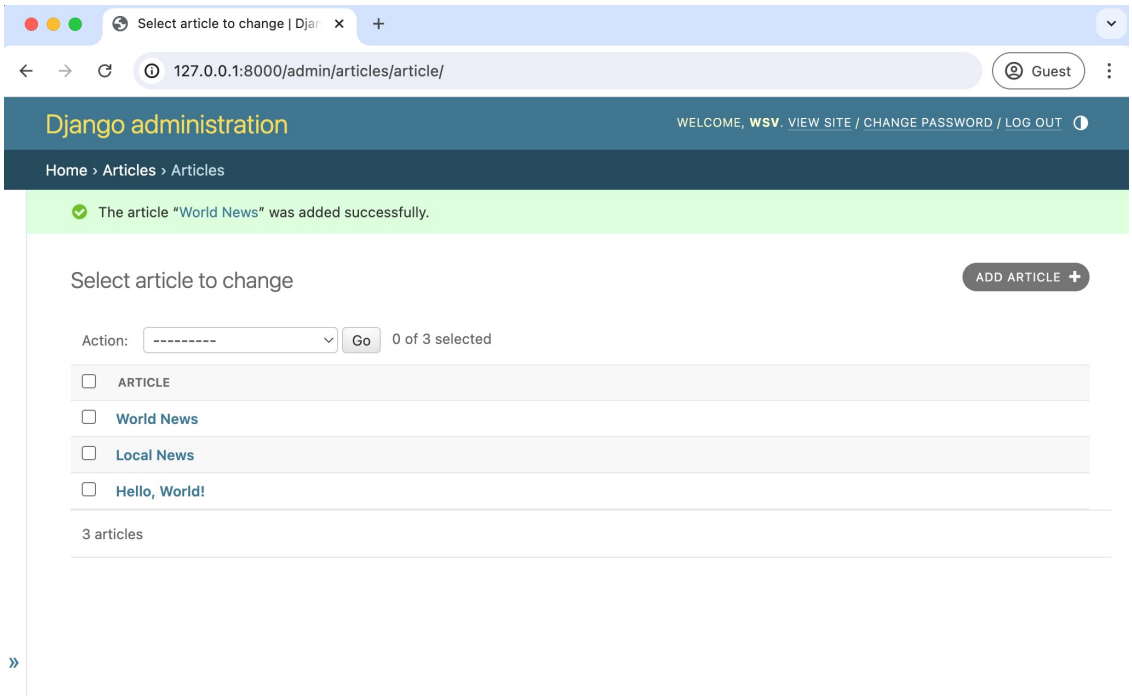
```
(.venv) $ python manage.py runserver
```

Navigate to the admin at `http://127.0.0.1:8000/admin/` and log in.



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.



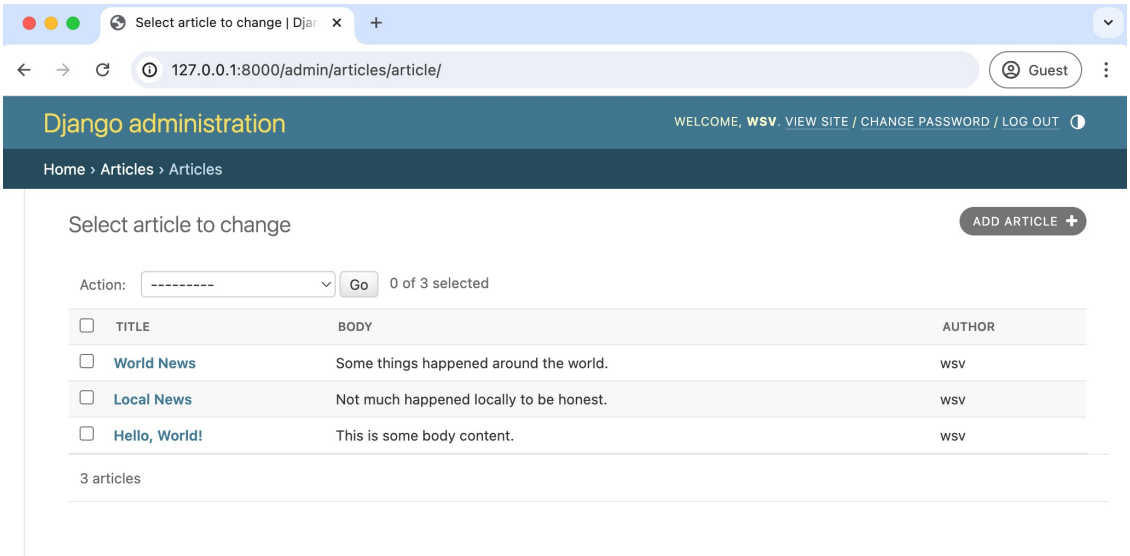
Admin Three Articles

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)
```

We've extended `ModelAdmin`, a class that represents a model in the admin interface, and specified our fields to list with `list_display`. At the bottom of the file we registered `ArticleAdmin` along with the `Article` model 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 web pages.

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"
```

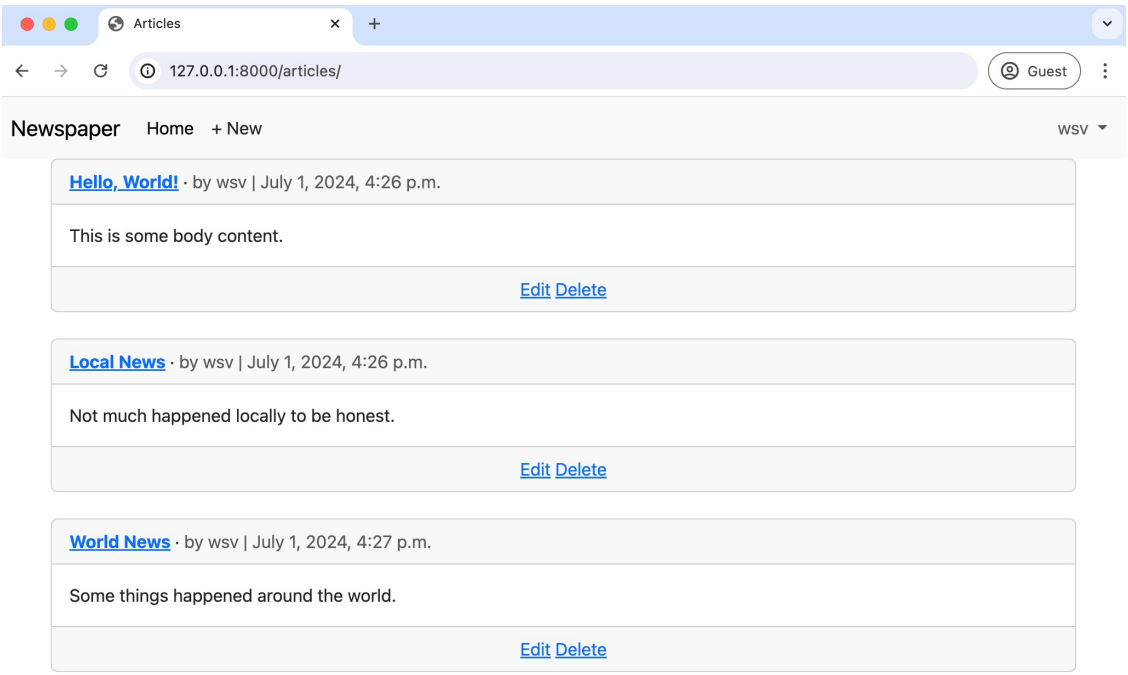
The last step is to create a new template file in the text editor called `templates/article_list.html`. Bootstrap has a built-in component called Cards that we can customize for our individual articles. Recall that `ListView` returns an object with `<model_name>_list` that we can iterate using a `for` 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 `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), 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, which displays the title, date, body, and author and links to edit and delete. It also links back to all articles. `DetailView` automatically names the context object either `object` or the lowercase model name. Recall that the Django templating language's `url` tag wants the URL name, and any arguments are 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 ms-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 ms-2" type="submit">Confirm</button>
</form>
{% endblock content %}
```

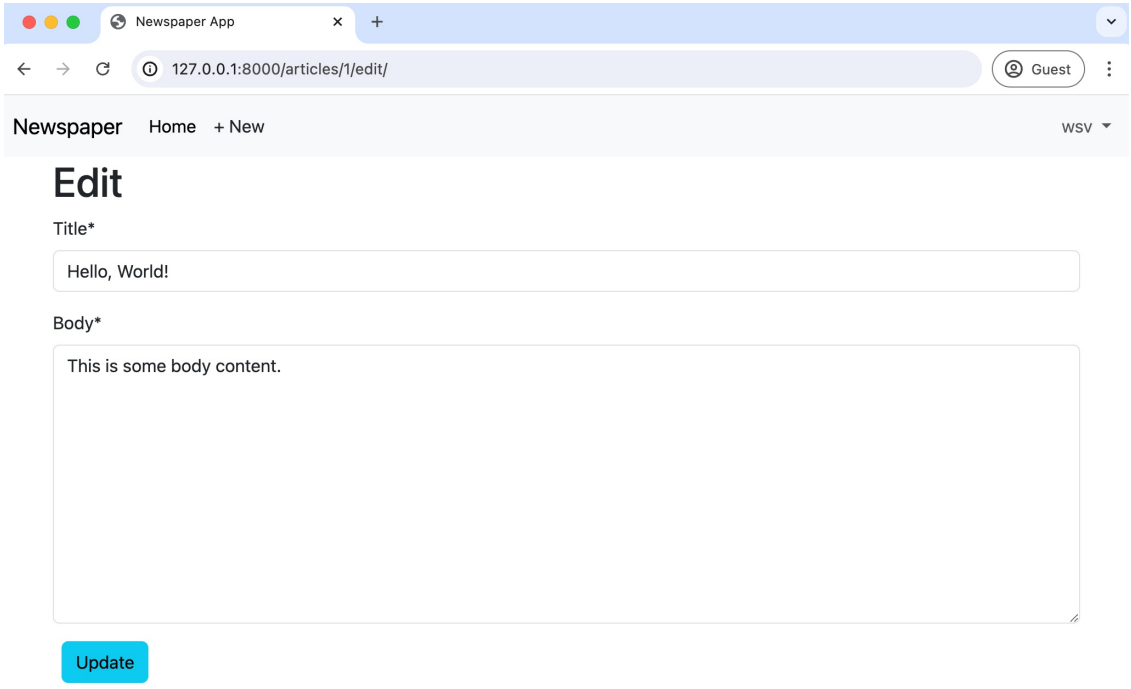
As a final step, in `article_list.html`, we can now add URL routes for detail, edit, and delete pages to replace the existing `<a href="#">` placeholders. We can use the `get_absolute_url` method defined in our model for the detail page. And we can use the url template tag, the URL name, and the `pk` 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/`.

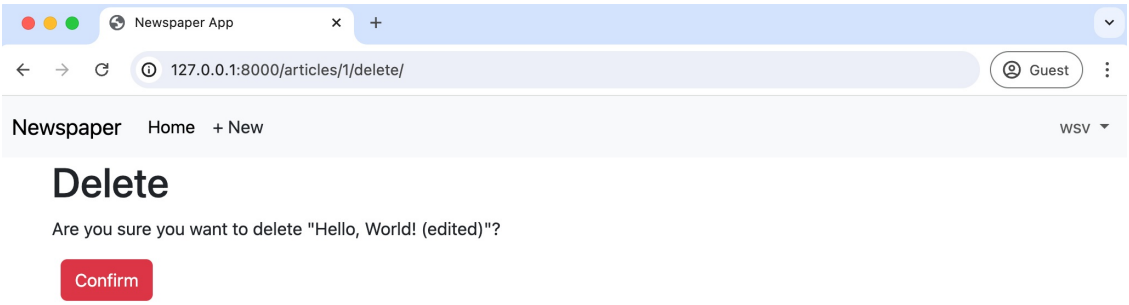


Edit Page

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.

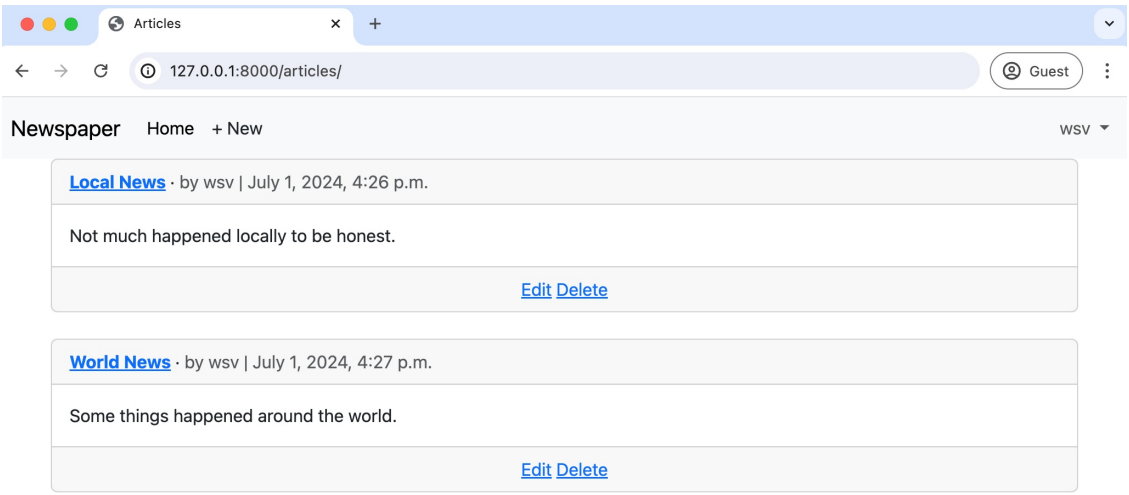
Detail Page

You’ll be redirected to the delete page if you click the “Delete” link.



Delete Page

Press the scary red button for “Confirm.” You’ll be redirected to the articles page, which now has only two entries.



Articles Page Two Entries

Create Page

The final step is to create a 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 create a new class called `ArticleCreateView` at the bottom of the file 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 ms-2" type="submit">Save</button>
</form>
{% endblock content %}
```

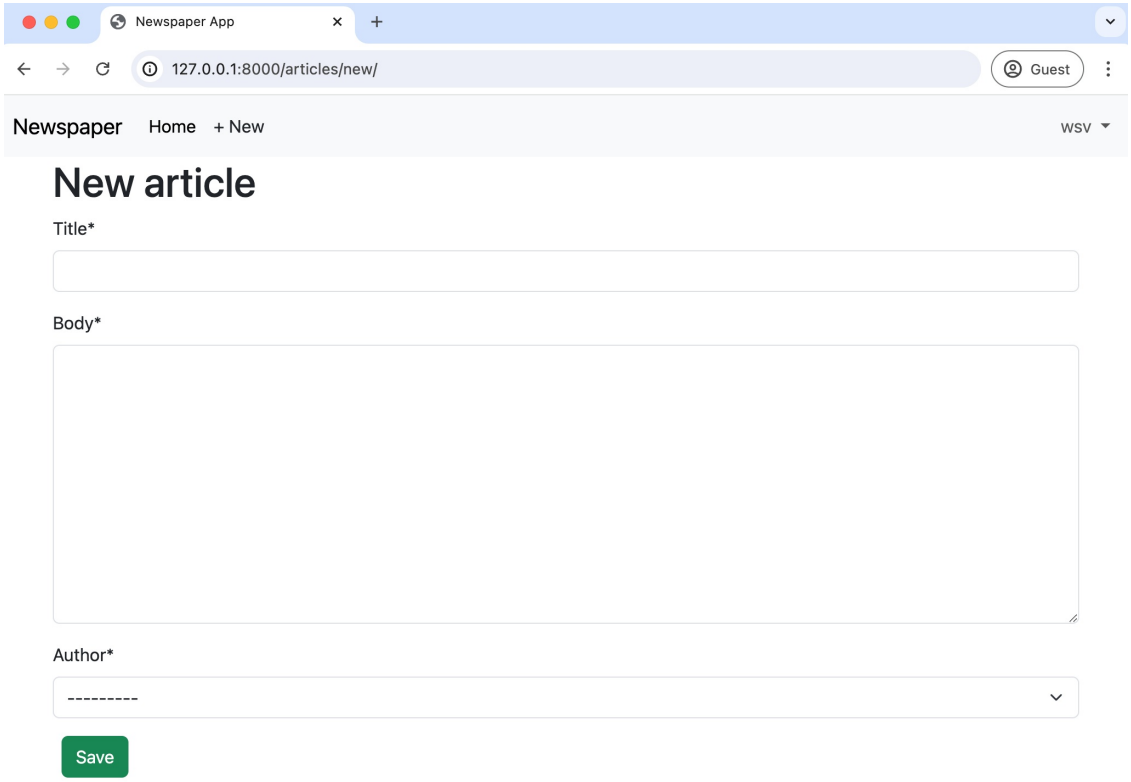
Additional Links

We should add the URL link for creating new articles to our navbar so that logged-in users can access it everywhere on the site.

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.



New Article Page

One final link to add is to make the articles list page accessible from the home page. 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.
<form action="{% url 'logout' %}" method="post">
  {% csrf_token %}
  <button type="submit">Log Out</button>
</form>
<br/>
<p><a class="btn btn-primary" 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.

Homepage 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 edit an existing article or delete one, even one that’s not theirs! In the next chapter, we will add permissions and authorizations to our project to fix this.

Chapter 14: Permissions and Authorization

There are several issues with our current *Newspaper* website. For one thing, we want our newspaper to be financially sustainable. 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, such as requiring users to log in to view articles. As a mature web framework, Django has built-in authorization functionality that we can use to restrict access to the articles list page and add additional restrictions so that only the author of an article can edit or delete it.

Improved CreateView

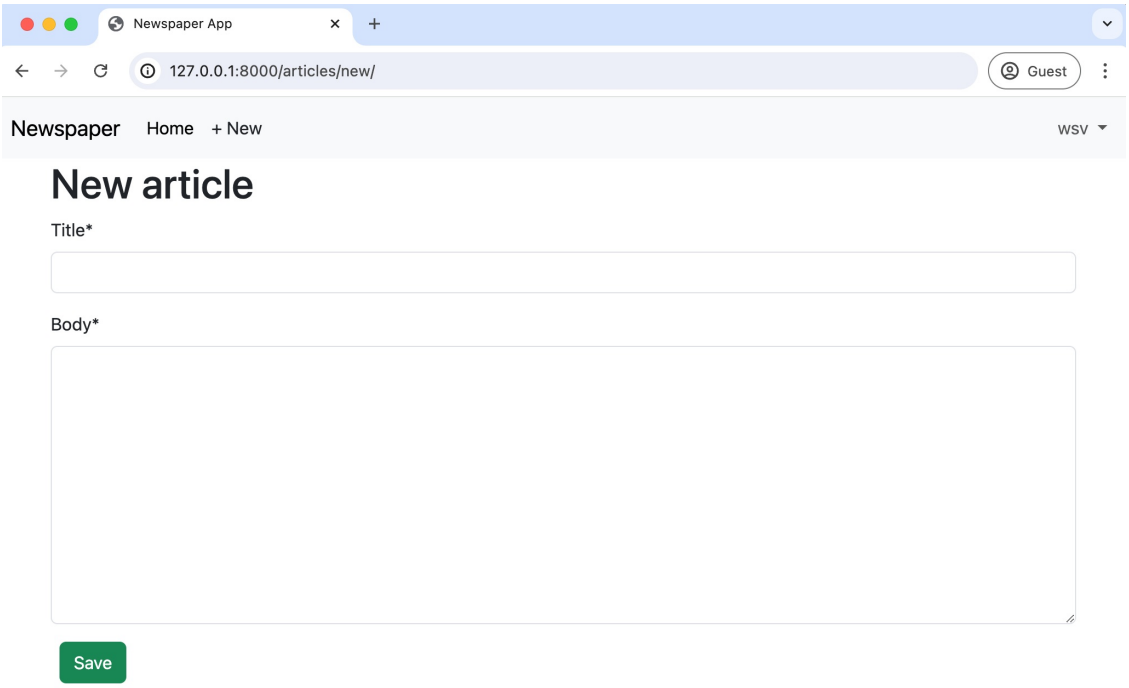
Currently, the `author` on a new article can be set to any existing user. Instead, it should be automatically set to the currently logged-in user. We can modify Django's `CreateView` to achieve this by removing the `author` field and instead setting it automatically via the `form_valid` method.

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 that 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 understand what's happening under the hood. This is the downside of class-based 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.



New Article Link

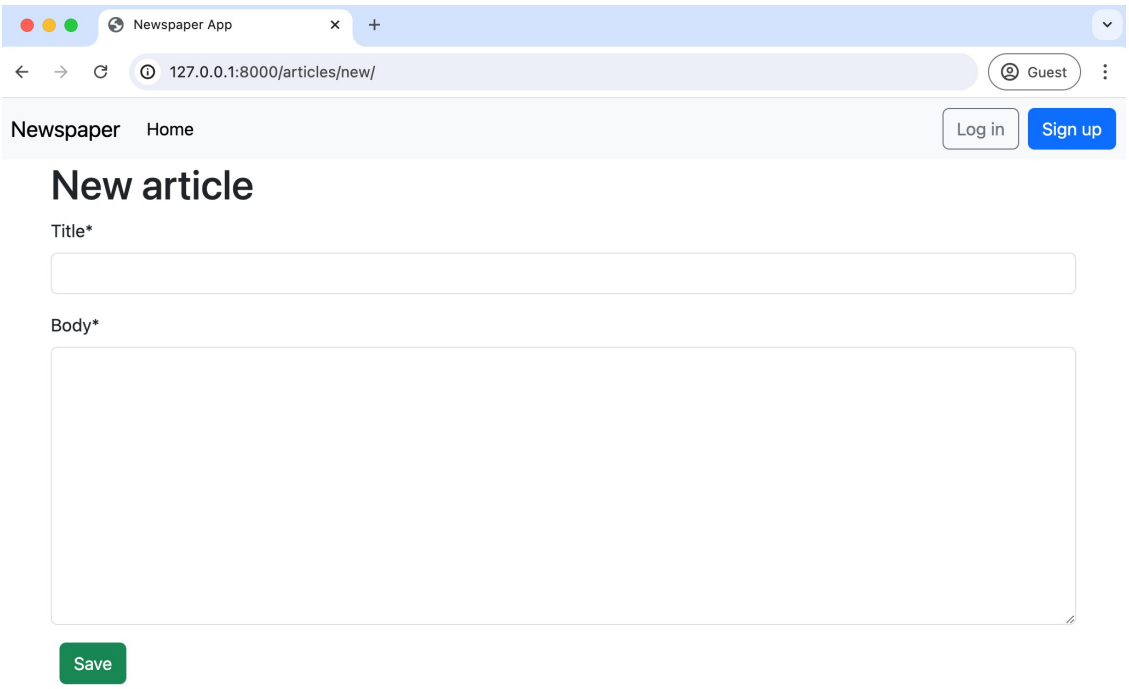
If you create a new article and go into the admin, you will see it is automatically set to the currently logged-in user.

Authorizations

There are multiple issues related to the lack of authorizations for our current project. We want 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.

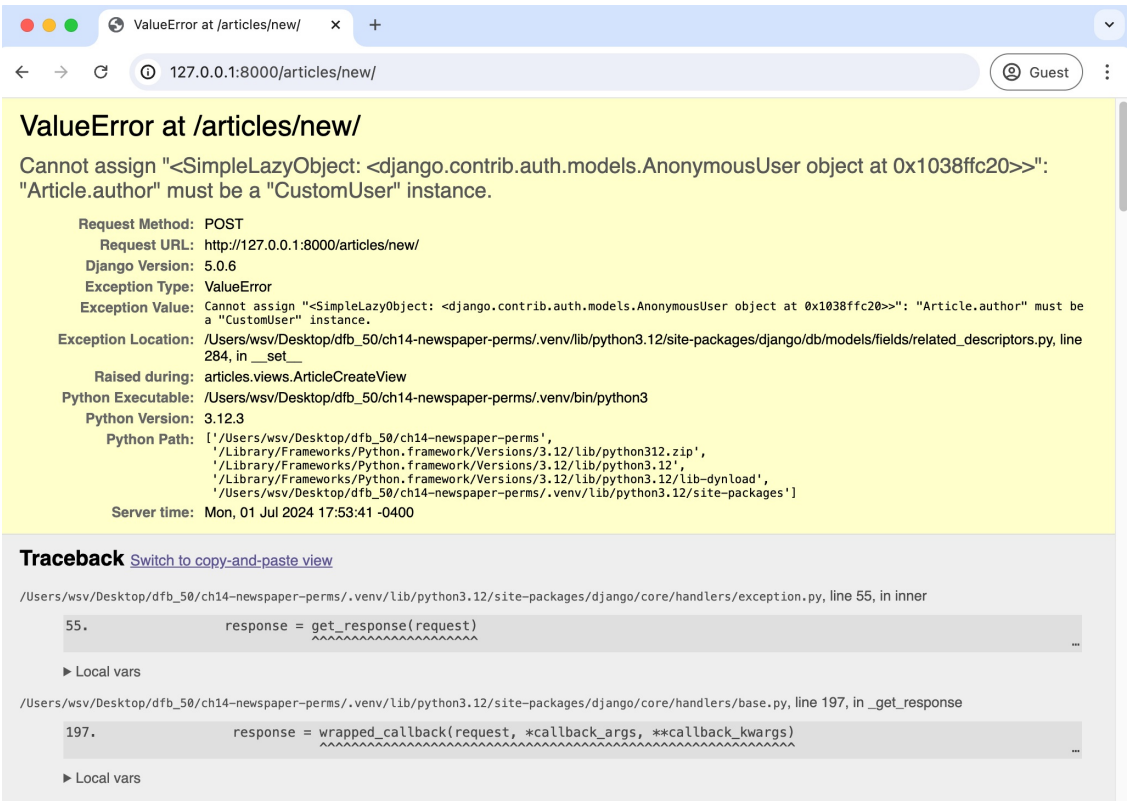
Consider what would happen if a logged-out user tried to create a new article. To try it out, click on your username in the upper right corner of the navbar, then select “Log Out” from the dropdown options. The “+ New” link disappears from the navbar, but what happens if you go to it directly: `http://127.0.0.1:8000/articles/new/`.

The page is still there.



Logged Out New

Now, try to create a new article with a title and body. Then, click on the “Save” button.



Create Page Error

An error! This is because our model **expects** an `author` field that 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.

You'll see mixins used everywhere if you read the Django source code, which is freely available on Github. To restrict view access to only logged-in users,

Django has a LoginRequired mixin that we can use. It's powerful and extremely concise.

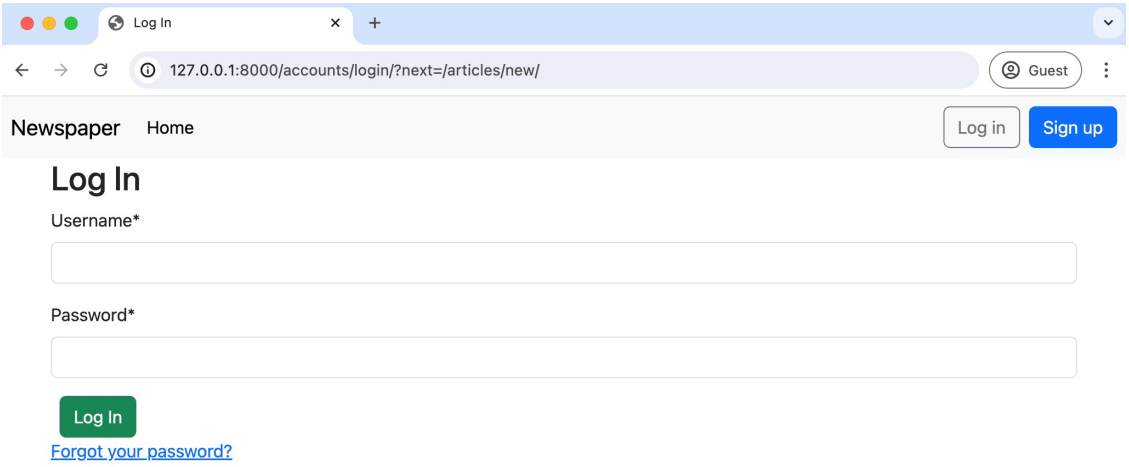
In the `articles/views.py` file, import `LoginRequiredMixin` and add it to `ArticleCreateView`. Make sure that the mixin is to the left of `CreateView` so it will be read first. We want the `CreateView` to know we intend to restrict access.

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)
```

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 progressing, but our edit and delete views are still an issue. Any logged-in user can change any article, but we want to restrict this access so that only the article's author 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
```

The order is critical when using mixins with class-based views. `LoginRequiredMixin` comes first so that we force login, then add `UserPassesTestMixin` for an additional layer of functionality, and finally, either `UpdateView` or `DeleteView`. The code will only work properly if you 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 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 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 author of an article.

We can add simple logic to our `article_list` and `article_detail` template files by using the built-in if filter so that only the article author can see the edit and delete links.

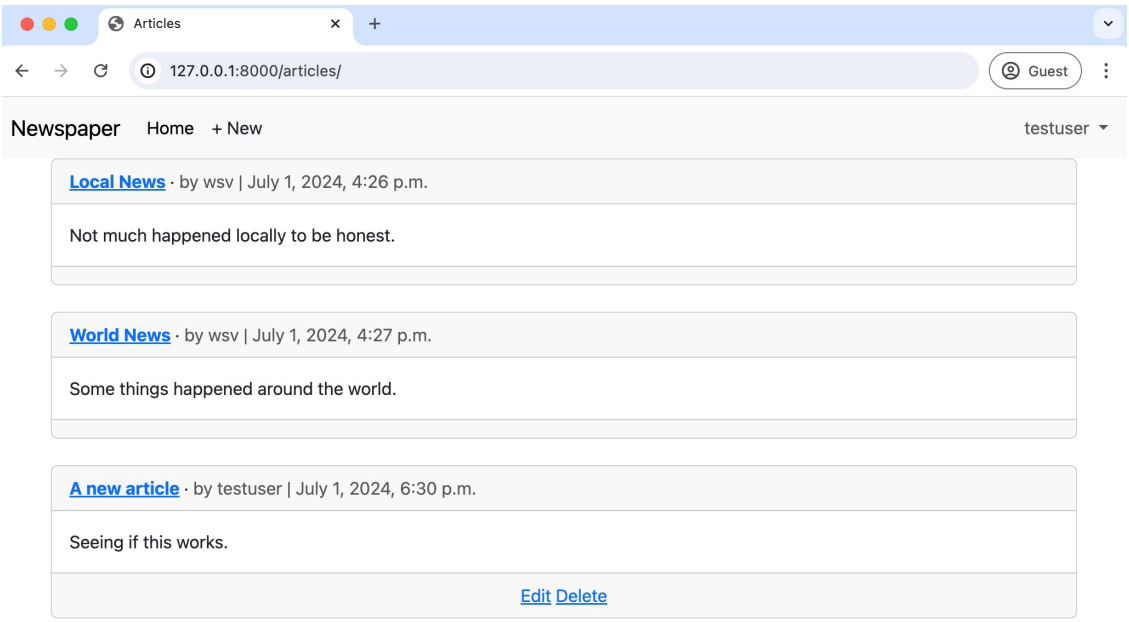
Code

```
<!-- 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 -->
</div>
...
```

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, make sure you are logged in as `testuser` and create a new article using the “+ New” button in the top navbar. If you refresh the all articles webpage, only the article authored by `testuser` should have the edit and delete links visible.



Edit/Delete Links Not Shown

Click on the article name to navigate to its detail page. The edit and delete links are visible.

Edit/Delete Links Shown for testuser

However, if you navigate to the detail page of an article created by superuser, the links are gone.

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 15: 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, forms. We need all four for the final solution, but the order in which we tackle them is largely up to us. 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 vice versa. 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 because consider what would happen 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
Migrations for 'articles':
  articles/migrations/0002_comment.py
    - Create model Comment
(.venv) $ python manage.py migrate
Operations to perform:
  Apply all migrations: accounts, admin, articles, auth, contenttypes, sessions
Running migrations:
  Applying articles.0002_comment... OK
```

Admin

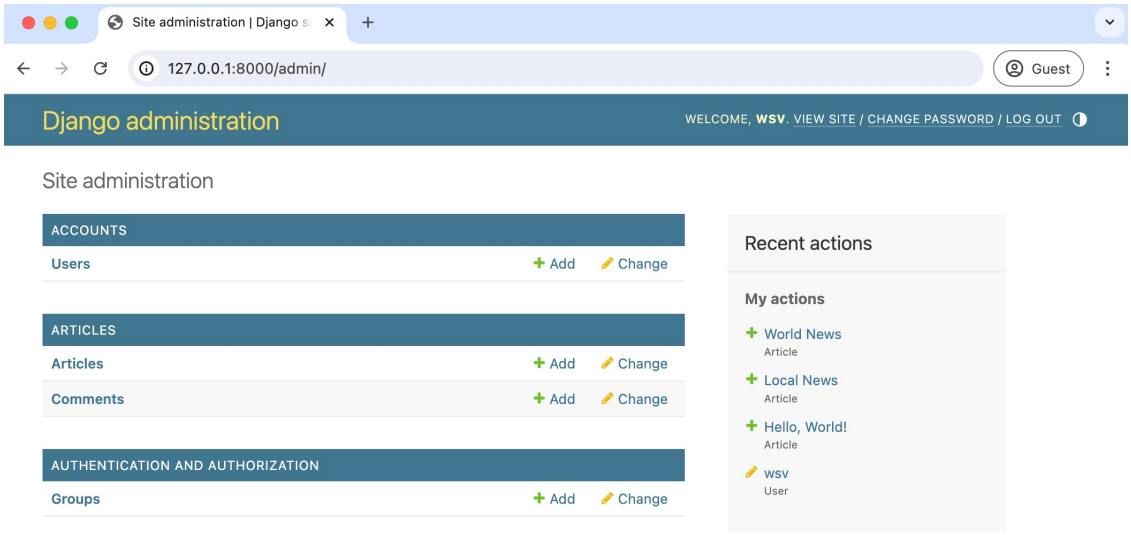
After making a new model, it's a good idea to play around with it in the admin app before displaying it on our website. Add `Comment` to our `admin.py` file so it will be visible.

Code

```
# articles/admin.py
from django.contrib import admin
from .models import Article, Comment  # new


class ArticleAdmin(admin.ModelAdmin):
```

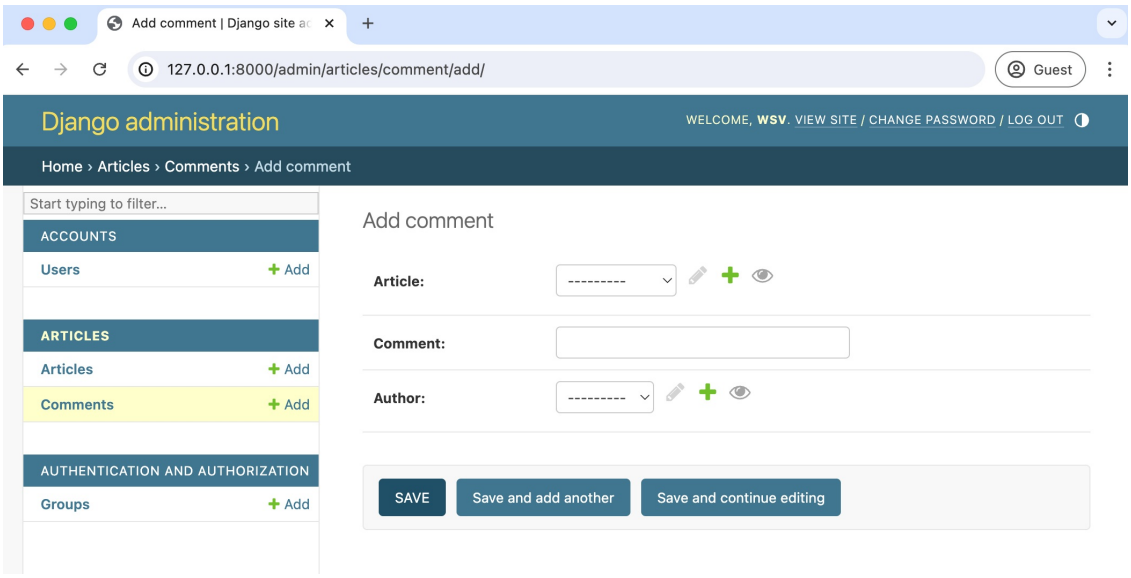
```
    list_display = [
        "title",
        "body",
        "author",
    ]


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/`.



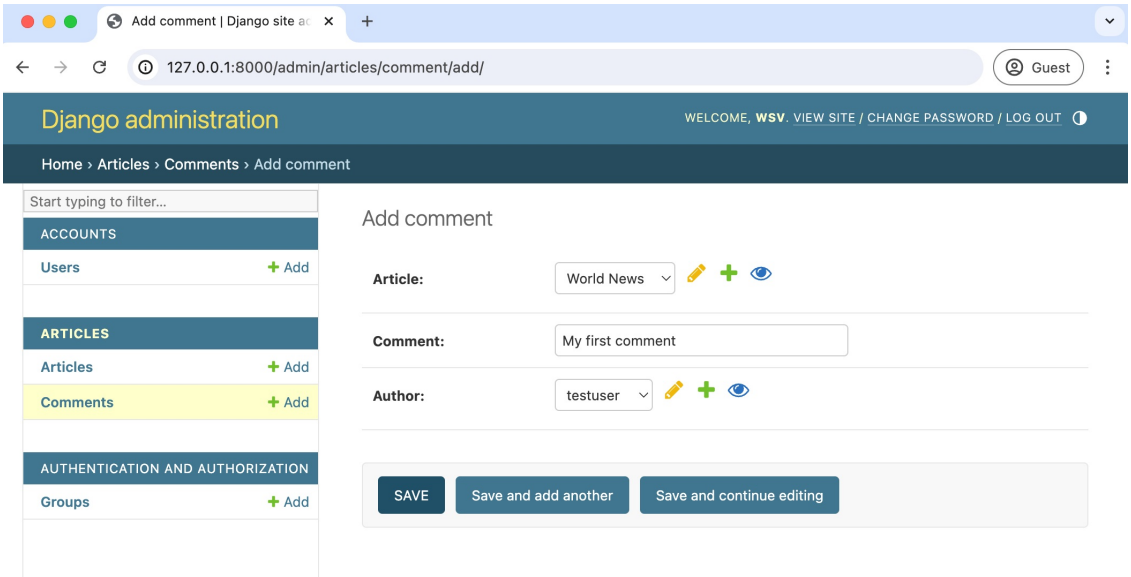
Admin Homepage with Comments

Under the “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.



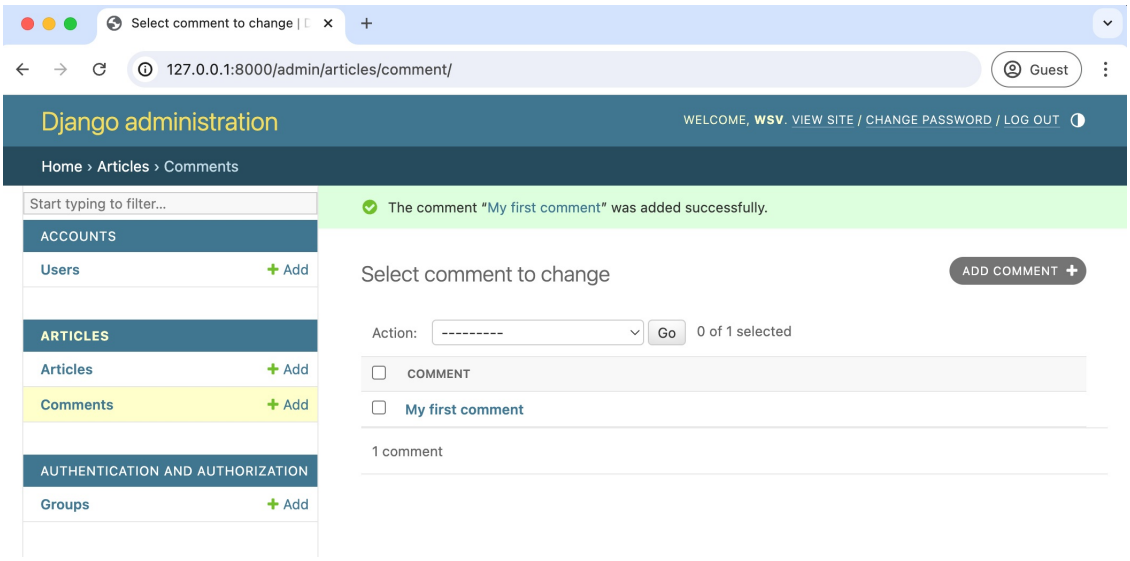
Admin Comments

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.



Admin testuser Comment

You should next see your comment on the admin "Comments" page.



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? 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.

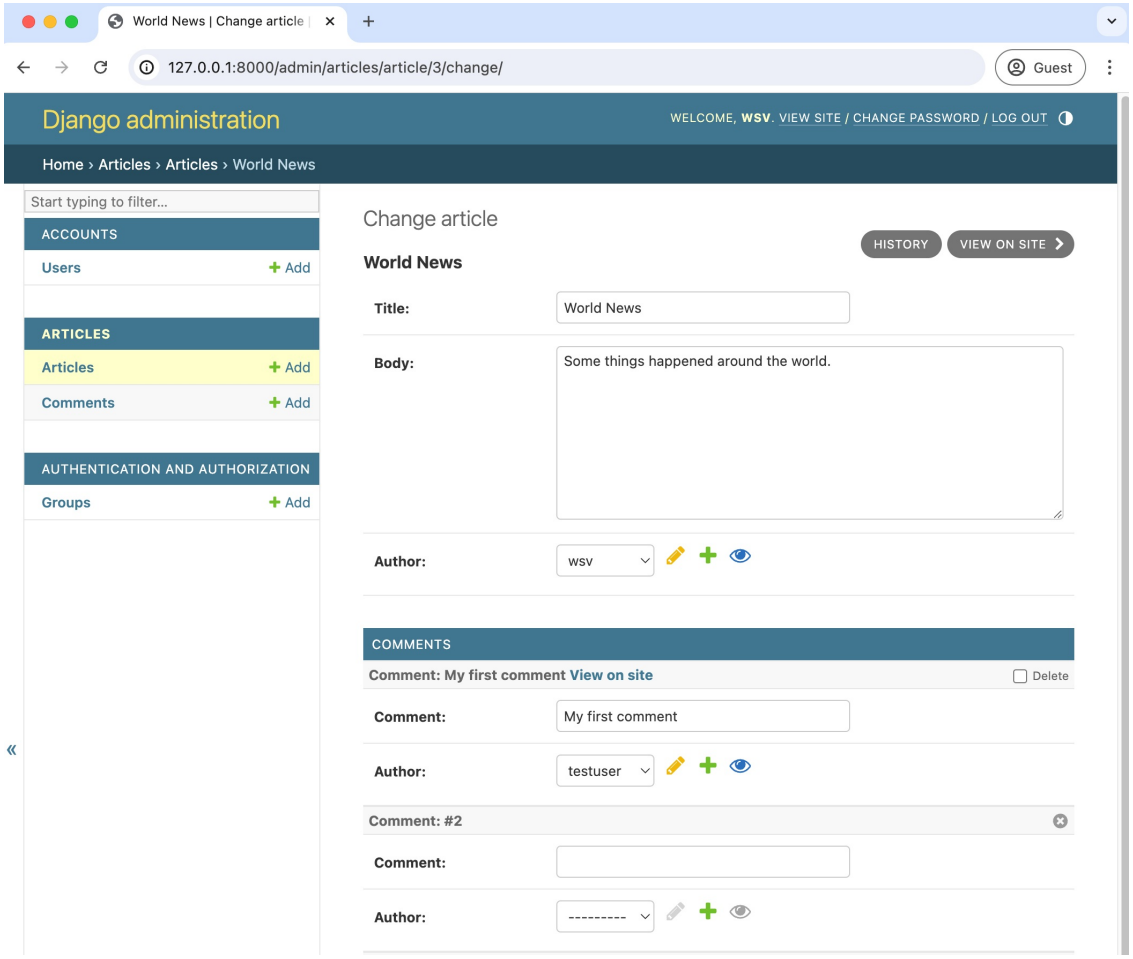
Update `articles/admin.py` in your text editor to add the StackedInline view.

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.



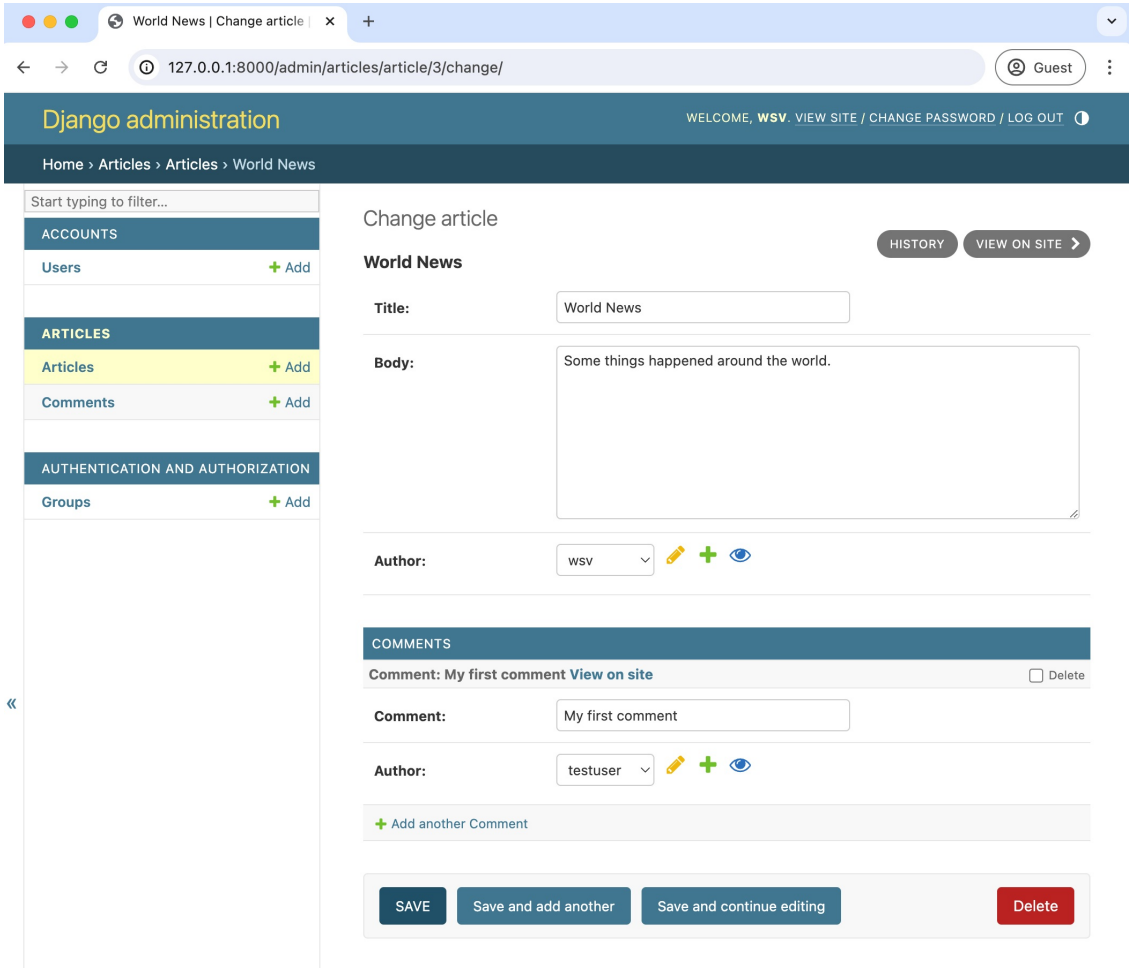
Admin Change Page

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
```



Admin No Extra Comments

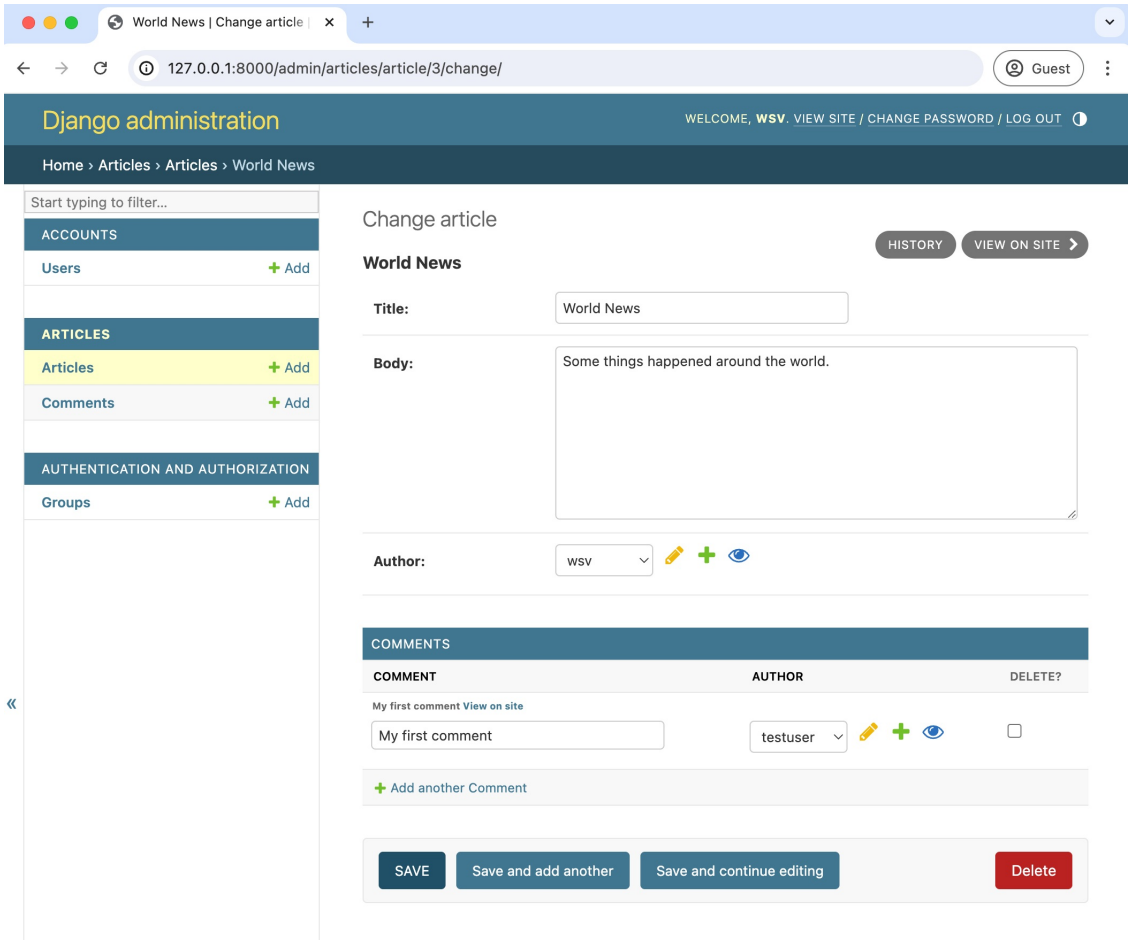
Personally, though, I prefer using `TabularInline` as it shows more information in less space: the comment, author, and more on one single line. To switch to it, we only need to change our `CommentInline` from `admin.StackedInline` to `admin.TabularInline`.

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.



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`.

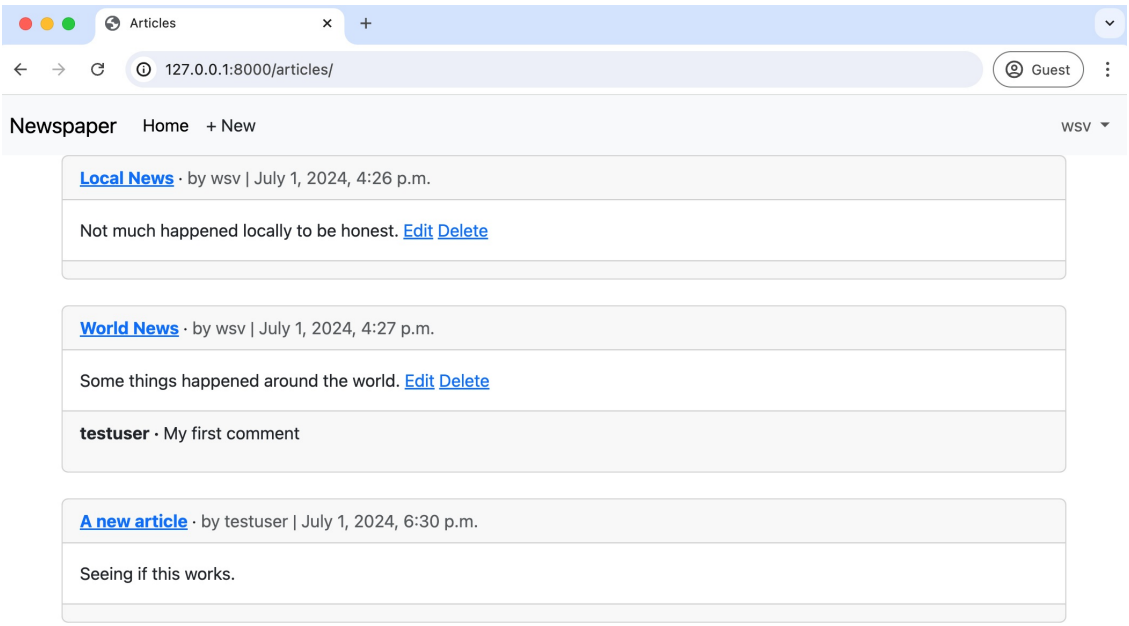
Let's start with `article_list.html`. If you look at the `articles/models.py` file again, it is clear that Comment has a foreign key relationship to the article. To display **all** comments related to a specific article, we will follow the relationship backward via a "query," which is a way to ask the 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 }} &middot;
      </span>
      {{ comment }}
    </p>
    {% endfor %}
  </div>
</div>
<br/>
{% endfor %}
{% endblock content %}
```

If you refresh the articles page at `http://127.0.0.1:8000/articles/`, we can see our new comment on the page.



Articles Page with Comments

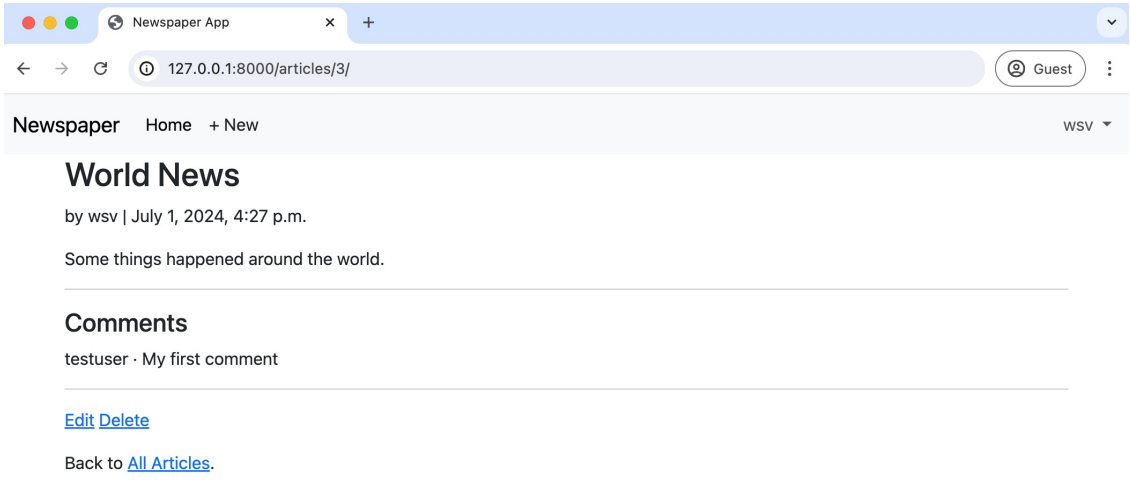
Let's also add comments to the detail page for each article. We'll use the same technique of following the relationship backward to access comments as a foreign key of the article model.

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.



Article Details Page with Comments

We won't win any design awards for this layout, but this is a book on Django, so our goal is to output the correct content.

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 that translates 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. Finally, create the class `CommentForm`, specifying both the underlying model and the 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 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 `CommmentForm()`, and returned the updated context.

Comment Template

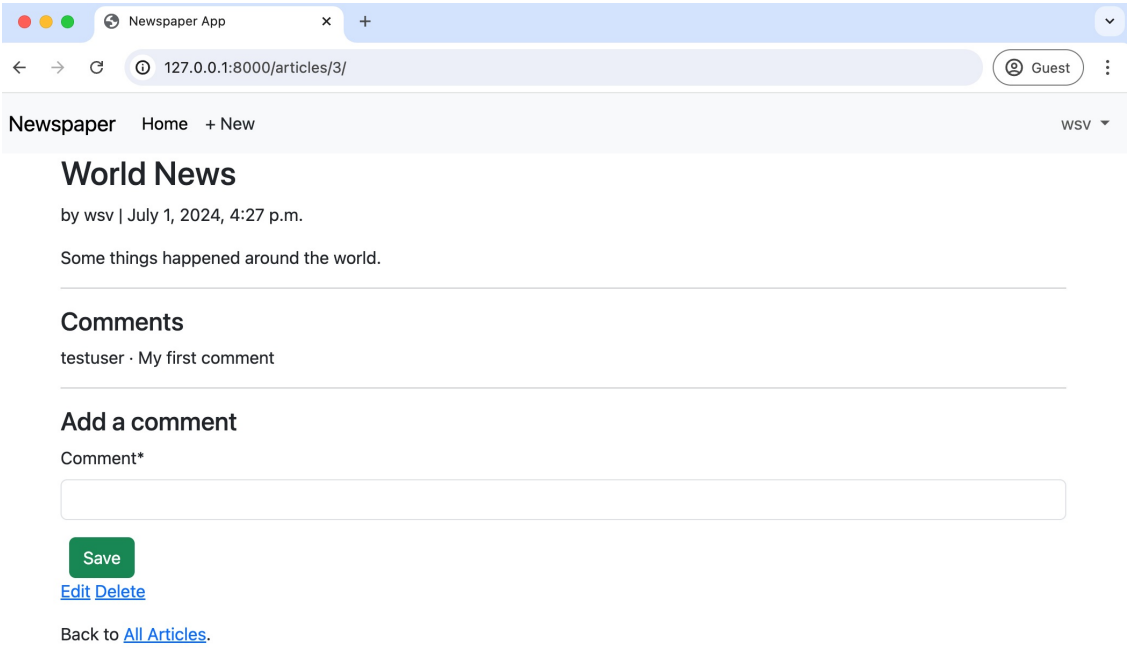
To display the form in our `article_detail.html` template file, we'll rely on the `form` variable and crispy forms. This pattern is the same as what we've done before in our other forms. At the top, load `crispy_form_tags`, create a standard-looking `post` form that uses a `csrf_token` for security, 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 ms-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.



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 the 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.

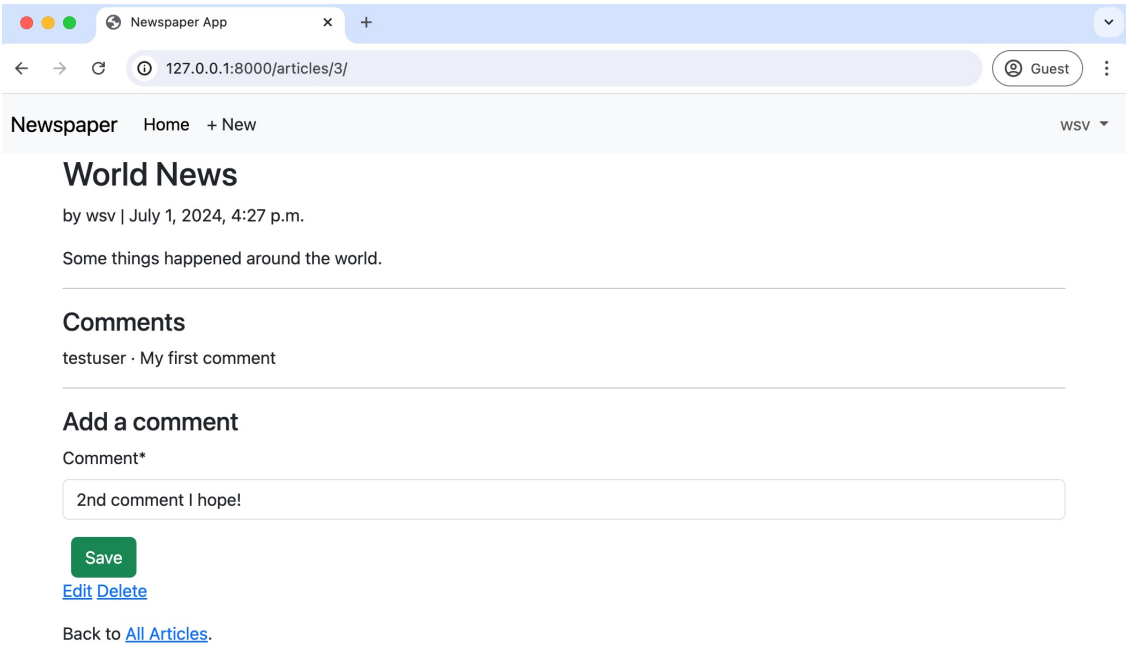
Code

```
# 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

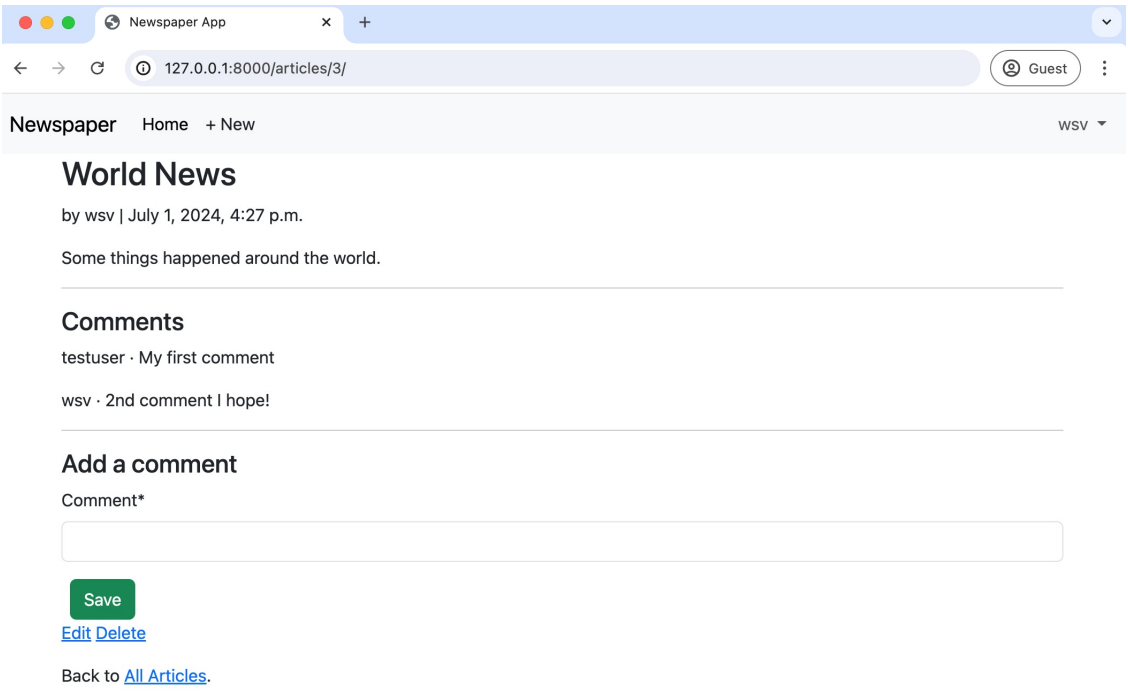
article it belongs to. Initially, we save the form but set `commit` to `False` because we associate the correct article with the form object in the next line. We also set the `author` field in our `Comment` model to the current user. In the following line, we save the form. Finally, we return it as part 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.

And we're done! Go ahead and load your articles page now, refresh the page, then try to submit a second comment.



Submit Comment in Form

It should automatically reload the page with the new comment displayed like this:



Comment Displayed

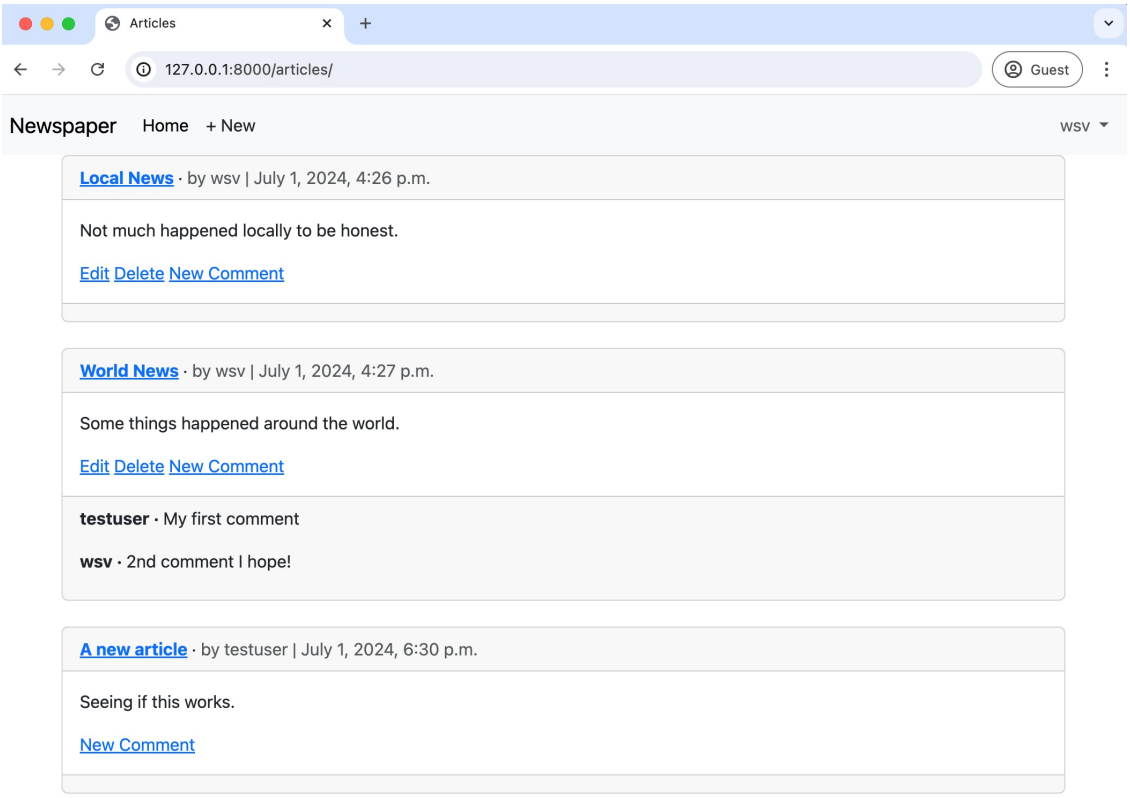
New Comment Link

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, articles, comments, and improved styling with

Bootstrap. We even dipped our toes into permissions and authorizations.

The remaining task is to deploy it online. In the next chapter, we'll see how to properly deploy a Django site using environment variables, PostgreSQL, and additional settings.

Chapter 16: 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 production. Django is designed to make web developers' lives easier, and therefore, it defaults to a local configuration when the `startproject` command is first run. We've seen this in the use of SQLite as the local file-based database, the built-in `runserver` command that launches a local web server in your web browser, and various default configurations in the `settings.py` file, including `DEBUG` set to `True` and an auto-generated `SECRET_KEY`.

In production, things are different. All the configurations optimized for ease of use in the development environment need to focus instead on security, performance, and scalability.

Deploying a Django website to production requires several steps—so many, in fact, that the Django docs even have a deployment checklist. It is a very helpful tool, but unfortunately, it is not sufficient. Additional factors include various hosting options, environment variables, database configurations, handling static files, and more.

In this chapter, we will create a deployment checklist and deploy the Newspaper project using Heroku. The techniques covered will apply to almost any Django website that needs to be readied for production, regardless of the hosting platform.

Hosting Options

If you ask five Django developers for the best hosting option, you'll likely receive five different answers. Everyone has a different preference based on their own experience and project needs.

Ultimately, though, we can divide hosting options into three main categories:

- **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.
- **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. It typically comes with a managed database, as well. The downside is that a developer cannot access the same degree of customization that a VPS or a dedicated server provides. At scale, PaaS can become quite expensive.

The choices here are all about tradeoffs. Many Django developers and small companies are happy using a PaaS to essentially abstract away many of the difficulties inherent in putting code into production. Popular PaaS options include Heroku, Fly, and Render, among many others. For a VPS, Digital Ocean has the cleanest interface for a solo developer or small team, but for enterprise applications, the choice is typically between AWS, Google, or Microsoft.

For our Newspaper website, we will use a PaaS, specifically Heroku, because it is mature, widely used, and has a relatively straightforward deployment process. However, with the exception of creating a Heroku-specific `Procfile` file at the end, the steps outlined in this chapter will work with any PaaS provider.

Web Servers and WSGI/ASGI Servers

The local server provided by Django and run via `runserver` handles multiple jobs that must be handled differently in production. First, it acts as a *web server*, 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, a Platform-as-a-Service knows we are deploying a website and automatically bundles a web server, generally Nginx, so we don't have to install or manage it ourselves.

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, `runserver` powers that request through URLs, views, models, the database, and templates

and then generates an HTTP response. In other words, `runserver` also acts as an *application server*, not just a web server.

Application servers are colloquially referred to as "WSGI servers" because they use WSGI to connect Python web apps to a server. In the early days of web development, web frameworks didn't implicitly work well with various web servers without a lot of customization. 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. Common examples of production WSGI servers include Gunicorn, uWSGI, and Daphne.

Traditionally, Python was a *synchronous* programming language: code executed sequentially, meaning each piece of code had to be completed 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 the result can then be reported back when complete.

Django has been gradually adding asynchronous support since version 3.0 in 2019. 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 to be 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`.

Full async support for the entire Django stack is still in the works but comes ever closer with each new major release. 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 most relevant for websites that need “real-time” functionality, such as live notifications, chat applications, real-time data updates, and interactive dashboards.

Deployment Checklist

It can be overwhelming to see a complete deployment checklist right at the beginning, but it is a helpful guide for this chapter. Here, then, is the deployment checklist we will cover:

- configure static files and install `WhiteNoise`
- add environment variables with `environs`
- create a `.env` file and update the `.gitignore` file
- update `DEBUG`, `ALLOWED_HOSTS`, `SECRET_KEY`, and `CSRF_TRUSTED_ORIGINS`
- update `DATABASES` to run PostgreSQL in production and install `psycopg`
- install `Gunicorn` as a production WSGI server
- create a `Procfile`
- update the `requirements.txt` file
- create a new Heroku project, push the code to it, and start a dyno web process

We could toggle many more production settings, but this list covers the most critical security and performance concerns.

Static Files

Static files are the images, JavaScript, and CSS used by a website. We worked with them in Chapter 6 on the *Blog* project, where we added custom CSS. For local usage, as long as `DEBUG` is set to `True` in `settings.py`, the files are served automatically by the `runserver` command.

Django automatically looks for static files within each app in a folder called “static,” but a common technique is to place all static files in a project-level folder called “static” instead. We’ll do that here. Quit the local server with `Control+c` and create a new `static` directory in the same folder as the `manage.py` file. Add new folders within it for `css`, `js`, and `img` on the command line.

```
(.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. If no files are within a folder, Git ignores it by default. One solution–which we will adopt here–is to add a `.keep` file to the three subfolders with your text editor:

- `static/css/.keep`
- `static/js/.keep`
- `static/img/.keep`

For local usage, only two settings are required for static files: `STATIC_URL`, which is the base URL for serving static files, and `STATICFILES_DIRS`, which defines the additional locations the built-in `staticfiles` app will traverse looking for static files beyond an `app/static` folder.

Code

```
# django_project/settings.py
STATIC_URL = "static/"
STATICFILES_DIRS = [BASE_DIR / "static"]  # new
```

Our local Django server is not designed to host static files in production. A best practice is to bundle all the static files into a single directory and then have the production web server, not the Django server, serve them. Django has a management command, `collectstatic`, for just this purpose: it copies all static files into a single location for deployment. The one configuration required of us is setting `STATIC_ROOT` to define the location of compiled static files. By convention, we will create a new project-level directory called `staticfiles`.

Code

```
# django_project/settings.py
STATIC_URL = "static/"
STATICFILES_DIRS = [BASE_DIR / "static"]
STATIC_ROOT = BASE_DIR / "staticfiles"  # new
```

Now run the command `python manage.py collectstatic` to compile all static files into the `staticfiles` folder.

Shell

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

The only static files right now are contained within the built-in `admin` app, so a new `staticfiles` directory should appear with sections for the admin. When additional static files are added in the future, they will also be compiled in this directory.

Code

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

We need to use the `{% load static %}` template tag to display static files in the templates. Add it now to the top of the `base.html` file.

Code

```
<!-- templates/base.html -->
{% load static %}
<!DOCTYPE html>
...
```

If we were deploying with a dedicated server or VPS, it would be up to us to write code for the web server, likely Nginx or Apache, to serve static files. But since we are using the PaaS Heroku, we can leverage the popular WhiteNoise third-party package optimized to serve static files from Django. It allows additional compression and immutable file storage and sets appropriate HTTP caching headers. In short, it makes our deployment process much more straightforward.

Install the latest version of `WhiteNoise` using `pip`.

Shell

```
(.venv) $ python -m pip install whitenoise==6.7.0
```

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

- add `whitenoise` to the `INSTALLED_APPS` **above** the built-in `staticfiles` app
- under `MIDDLEWARE`, add a new `WhiteNoiseMiddleware` on the third line
- change `STORAGES` 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",
    # 3rd Party
    "crispy_forms",
    "crispy_bootstrap5",
    # Local
    "accounts",
    "pages",
    "articles",
]

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",
]

...
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
    },
}
```

STORAGES is a new setting in Django 4.2+ that defines how files are stored. It is implicitly set in `settings.py` but we are changing the `staticfiles` section to use `WhiteNoise` compression.

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
```

That’s it! We have configured our static files to be compiled in one place for production, added the `static` template tag to our `base.html` template, and

installed `WhiteNoise` to efficiently serve them.

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` **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.

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 architecture fit together.

Environment Variables

Real-world Django projects require at least two environments (local and production) but typically have more if multiple testing servers are involved. There are two ways to toggle between different environments in the same project: environment variables and multiple settings files. These days, the most popular approach is to use environment variables, which we will do here. An environment variable is a variable whose value is set outside the current program

and can be loaded in at runtime. We can store these variables securely and load them into our Django project as needed.

There are multiple ways to work with environment variables, but for this project, we will use environs, a popular third-party package that comes with additional Django-specific features. Use `pip` to add `environs` and include the double quotes, `""`, to install the helpful Django extension.

Shell

```
(.venv) $ python -m pip install "environs[django]"==11.0.0
```

Then, add three new lines to the top of the `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
```

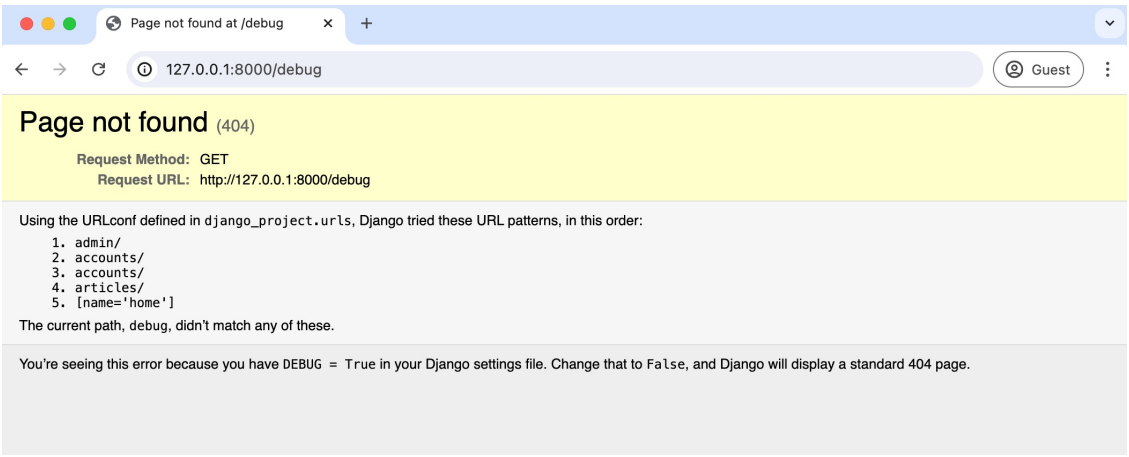
Next, create a new file, `.env`, in the root project directory, containing our environment variables. We already know that any file or directory starting with a period will be treated as a hidden file and not displayed by default during a directory listing. The file still exists, though, and needs to be added to the `.gitignore` file to avoid being added to our Git source control.

.gitignore

```
.venv/
__pycache__/
db.sqlite3
.env  # new!
```

DEBUG and ALLOWED_HOSTS

The first setting we will configure with environment variables is `DEBUG`. By default, `DEBUG` is set to `True`, which is helpful for local development but a major security issue if deployed into production. For example, if you 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:



Debug Page

This page lists all the URLs tried and apps loaded, a treasure map for any hacker attempting to break into your site. You'll even see that at the bottom of the error page, it says that Django will display a standard 404 page if `DEBUG=False`. That's what we want! The first step is to change `DEBUG` to `False` in the `settings.py` file.

Code

```
# django_project/settings.py
DEBUG = False
```

Refresh the web page `http://127.0.0.1:8000/debug`, and you'll see an error: the site doesn't load at all. On the command line, Django has provided us with the explanation via CommandError, which is raised for serious problems.

Shell

```
(.venv) $ python manage.py runserver
...
CommandError: You must set settings.ALLOWED_HOSTS if DEBUG is False.
```

In this case, Django is telling us that we can't set `DEBUG` to `False` if we have not set `ALLOWED_HOSTS`. So what is ALLOWED_HOSTS? It is a list of strings representing host/domain names that our Django site can serve. By default, `ALLOWED_HOSTS` is set to accept all hosts, which is not secure! We must update it to accept local ports (`localhost` and `127.0.0.1`) and `.herokuapp.com` for our Heroku deployment. We can add all three routes to our config.

Code

```
# django_project/settings.py
ALLOWED_HOSTS = [".herokuapp.com", "localhost", "127.0.0.1"]  # new
```

Now that we've set `ALLOWED_HOSTS` try the `runserver` command again.

Not Found Page

This is the generic Django 404 page that we want displayed in production. It does not give away any information to a potential hacker.

Manually setting configurations for development and production environments is not ideal. For one thing, it is a major pain and easy to make a mistake. For another, it is insecure if we put production information that should be secret into `settings.py` and perform a Git commit by mistake.

This is where environment variables come to the rescue. To add any environment variable to our project, we first add it to the `.env` file and then update `django_project/settings.py`.

Within the `.env` file, create a new environment variable called `DEBUG` and set its value to `True`.

Code

```
# .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", default=False)
```

The syntax of `env.bool` says to load an environment variable from the `.env` file that is a Boolean, meaning true or false, with the name "DEBUG." If an environment variable can't be found, then the `default` value, set here to `False`, will be used. It is 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.

SECRET_KEY, and CSRF_TRUSTED_ORIGINS

`SECRET_KEY`, a random 50-character string generated each time `startproject` is run. This string provides cryptographic protection throughout our Django project. In the settings file, you'll see the current value that begins with `django-insecure`. Here is the `django_project/settings.py` value of the `SECRET_KEY` in my project. Yours will be different.

Code

```
# django_project/settings.py
SECRET_KEY = "django-insecure-3$k(g9eheqqbzr@#&tt)r6%ab-g1=!j@2c^y7*sl6+ltzys05!"
```

And here it is, without the double quotes, in the `.env file`

.env

```
DEBUG=True
SECRET_KEY=django-insecure-3$k(g9eheqqbzr@#&tt)r6%ab-g1=!j@2c^y7*sl6+ltzys05! # new
```

Update `django_project/settings.py` so that `SECRET_KEY` points to this new environment variable. It is a string so the syntax is `env.str`.

Code

```
# django_project/settings.py
SECRET_KEY = env.str("SECRET_KEY")
```

The `SECRET_KEY` is out of the settings file and safe now, right? Actually no! Because we made previous Git commits that included the value, it is stored in our Git history no matter what we do. The solution is to create a new `SECRET_KEY` and add it to the `.env` file. 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
```

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` loaded from the `.env` file but not tracked by Git since `.env` is in the `.gitignore` file.

Our Newspaper project requires that we log into the admin on the production website to create, read, update, or delete posts. That means CSRF_TRUSTED_ORIGINS must be correctly configured since it is a list of trusted origins for unsafe HTTP requests like `POST`. Add it to the bottom of the `settings.py` file and set it to match a production URL on Heroku, `https://*.herokuapp.com`. We will update both `ALLOWED_HOSTS` and `CSRF_TRUSTED_ORIGINS` to match our production URL at the end of the chapter.

Code

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

DATABASES

We want to use SQLite locally but PostgreSQL in production. Currently, our settings file for `DATABASES` lists only SQLite. The `ENGINE` specifies what type of database to use, and the `NAME` points to its location.

Code

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

Most PaaS 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. Conveniently, `dj-database-url` is already installed since it is one of the helper packages added by `environs[django]`.

This means 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.

Code

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

We will also set a `DATABASE_URL` environment variable in the `.env` file for local development.

.env

```
DEBUG=True
SECRET_KEY=imDnfLXy-8Y-YozfJmP2Rw_81YA_qx1XKl5FeY0mXyY
DATABASE_URL=sqlite:///db.sqlite3
```

Let's review what's happening here because it is can be confusing initially. For local development, our project will try to find a `.env` file with environment variables. It will use them if it finds them, which is why they are the local defaults.

In production, we have included the `.env` file in our `.gitignore` file so Heroku won't know about them unless we set the environment variables manually. Heroku will *automatically* create a `DATABASE_URL` environment variable with the configuration for our production database, so that will be used.

We also need to install Psycopg, a database adapter that lets Python apps like ours talk to PostgreSQL databases. You can install it with `pip` on Windows, but if you are on macOS, you must install PostgreSQL first via Homebrew.

Shell

```
# Windows
(.venv) $ python -m pip install "psycopg[binary]"==3.2.1

# macOS
(.venv) $ brew install postgresql
(.venv) $ python3 -m pip install "psycopg[binary]"==3.2.1
```

We are using the binary version because it is the quickest way to start working with Psycopg.

Gunicorn and Procfile

Since Django's default development server, `runserver`, is explicitly not designed for production, we must select a production-ready WSGI server to use. Gunicorn is one of the most popular and easiest-to-configure options. It can handle multiple requests simultaneously while being scalable, stable, reliable, and compatible with production web servers.

Install Gunicorn with `pip`. Since we are using a PaaS, no additional configuration steps are required.

Shell

```
(.venv) $ python -m pip install gunicorn==22.0.0
```

Heroku relies on a proprietary `Procfile` file that provides instructions on running applications in their stack. In your text editor, create a new file called `Procfile` in the base directory. We only need a single line of configuration for our project, telling Heroku to use Gunicorn as the WSGI server, the location of the WSGI config file at `django_project.wsgi`, and finally, the flag `--log-file -` makes any logging messages visible to us.

Procfile

```
web: gunicorn django_project.wsgi --log-file -
```

requirements.txt

We're almost at the end of implementing the deployment checklist. The last step before deploying to Heroku is to update the `requirements.txt` file. After all, for deployment we have installed the following new packages: `whitenoise`, `environs`, `psycopg`, and `gunicorn`.

Use the command `pip freeze` and the `>` operator to output our virtual environment information into a `requirements.txt` file.

Shell

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

The `requirements.txt` file will appear in the root directory containing all our installed packages and their dependencies. My list as of the writing of this book looks as follows:

Code

```
# requirements.txt
asgiref==3.8.1
black==24.4.2
click==8.1.7
crispy-bootstrap5==2024.2
dj-database-url==2.2.0
dj-email-url==1.0.6
Django==5.0.6
django-cache-url==3.4.5
django-crispy-forms==2.2
environs==11.0.0
gunicorn==22.0.0
marshmallow==3.21.3
mypy-extensions==1.0.0
packaging==24.1
pathspec==0.12.1
platformdirs==4.2.2
psycopg==3.2.1
psycopg-binary==3.2.1
python-dotenv==1.0.1
sqlparse==0.5.0
typing_extensions==4.12.2
whitenoise==6.7.0
```

We can use `git status` to check our changes, add the new files, and commit them. We can also push to GitHub for an online backup of our code changes.

Shell

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

Heroku Setup

Before 2022, Heroku had a generous free tier, but unfortunately, that is not the case anymore. It costs a company real money to spin up virtual servers on your behalf, and as a result, few hosting companies offer a free tier anymore.

Heroku pricing involves multiple tiers of features and bills per hour with a maximum monthly limit. The deployment setup we will implement here costs $12/month if left on all the time, but if you are cost-conscious and deploying for purely educational purposes, there is no reason to leave your website “live” all

the time. You can deploy the site, share it, and then take it down after a few days and the total cost should only be $1-$2.

Sign up for a Heroku account on their website. Fill in the registration form and await an email with a link to confirm your account. This will take you to the password setup page. Once configured, you will be directed to the dashboard section of the site. Heroku now requires enrolling in multi-factor authentication (MFA), which can be done with SalesForce or a tool like Google Authenticator. Heroku now requires adding a credit card for account verification and payment.

Once you are signed up, it is time to install Heroku's *Command Line Interface (CLI)* so we can deploy from the command line. Currently, we are operating within a local virtual environment for the *Newspaper* project. We want to install Heroku globally so it is available for all projects. An easy way to do this is by opening a new command line tab–``Control+t `on Windows or `Command+t` on a Mac–that is not currently operating in a virtual environment. Anything installed here will be global.

On Windows, see the Heroku CLI page to install the 32-bit or 64-bit version correctly. On a Mac, the package manager Homebrew is used for installation. If not already on your machine, install Homebrew by copying and pasting the long command on the Homebrew website into your command line and hitting Return. It will look something like this:

Shell

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/\
  install/HEAD/install.sh)"
```

Next, install the Heroku CLI by copying and pasting the following into your command line and hitting `Return`.

Shell

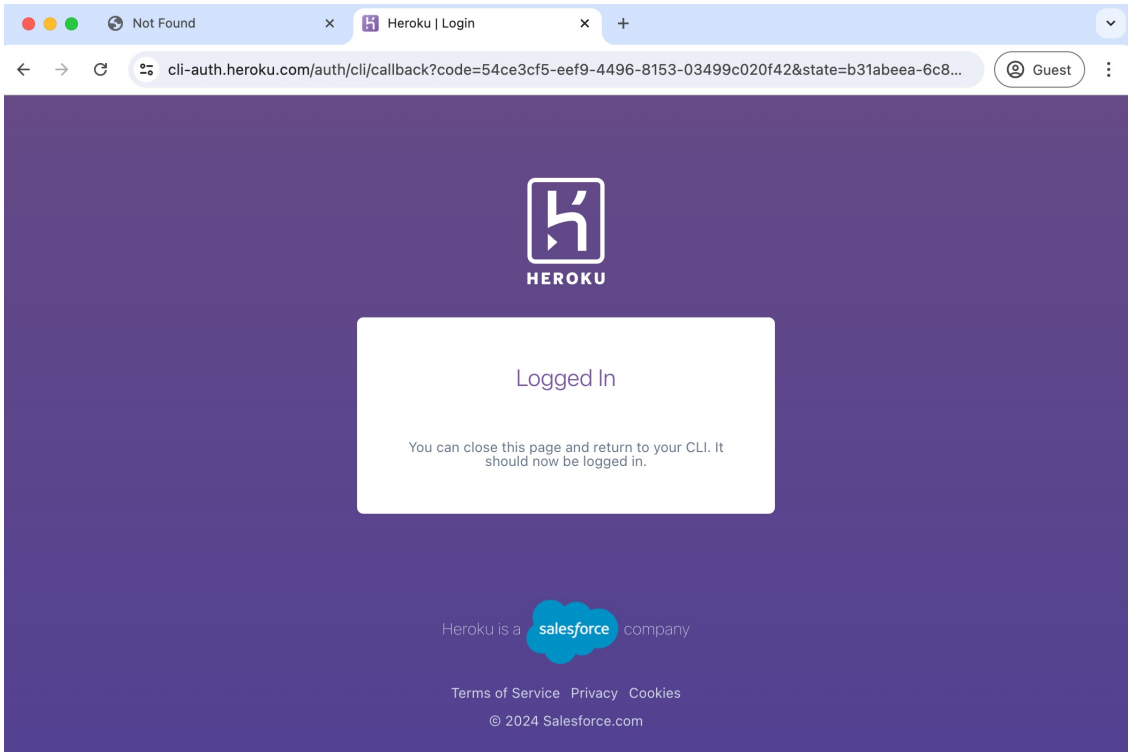
```
$ brew tap heroku/brew && brew install heroku
```

Once installation is complete, you can close the new command line tab and return to the initial tab with the `newspaper` virtual environment active.

Type the command `heroku login` and press `enter`. Then press any key to open up a browser window, where you can log in with your email, password, and two-factor authentication you just set.

Shell

```
(.venv) > heroku login
heroku: Press any key to open up the browser to login or q to exit:
Opening browser to https://cli-auth.heroku.com/auth/cli/browser/....
```



Heroku Login

Once you are logged in we are ready to go!

Deploy with Heroku

There are two ways to interact with Heroku: via its CLI (Command Line Interface) or the website. The CLI is much faster, which we will use here, but the website's more visual nature is helpful if you are new to Heroku.

The first step is to create a new Heroku app. from the command line with `heroku create`. Heroku will create a random name for our app, in my case `fathomless-hamlet-26076`. Your name will be different.

Shell

```
(.venv) $ heroku create
Creating app... done, ⬢ afternoon-wave-82807
```

```
https://afternoon-wave-82807-b672795cd97e.herokuapp.com/ |
https://git.heroku.com/afternoon-wave-82807.git
```

The `heroku create` command also creates a dedicated Git remote named `heroku` for our app. To see this, type `git remote -v`.

Shell

```
(.venv) $ git remote -v
heroku  https://git.heroku.com/afternoon-wave-82807.git (fetch)
heroku  https://git.heroku.com/afternoon-wave-82807.git (push)
```

The next step is creating a PostgreSQL database on Heroku. There are various Postgres tiers available for different use cases. The five plan tiers are Essential, Standard, Premium, Private, and Shield. The more you pay the less downtime is tolerated. For our use case, the lowest tier, Essential, is more than adequate. Run the following command to create a new Essential Postgres database for our project.

Shell

```
(.venv) $ heroku addons:create heroku-postgresql:essential-0
Creating heroku-postgresql:essential-0 on ⬢ afternoon-wave-82807...
  ~$0.007/hour (max $5/month)
Database should be available soon
postgresql-sinuous-77120 is being created in the background. The app will restart
  when complete...
Use heroku addons:info postgresql-sinuous-77120 to check creation progress
Use heroku addons:docs heroku-postgresql to view documentation
```

The database might require a moment to provision, in which case you can wait a few minutes and then run the command to “check creation progress.” Make sure the database name matches your project.

Shell

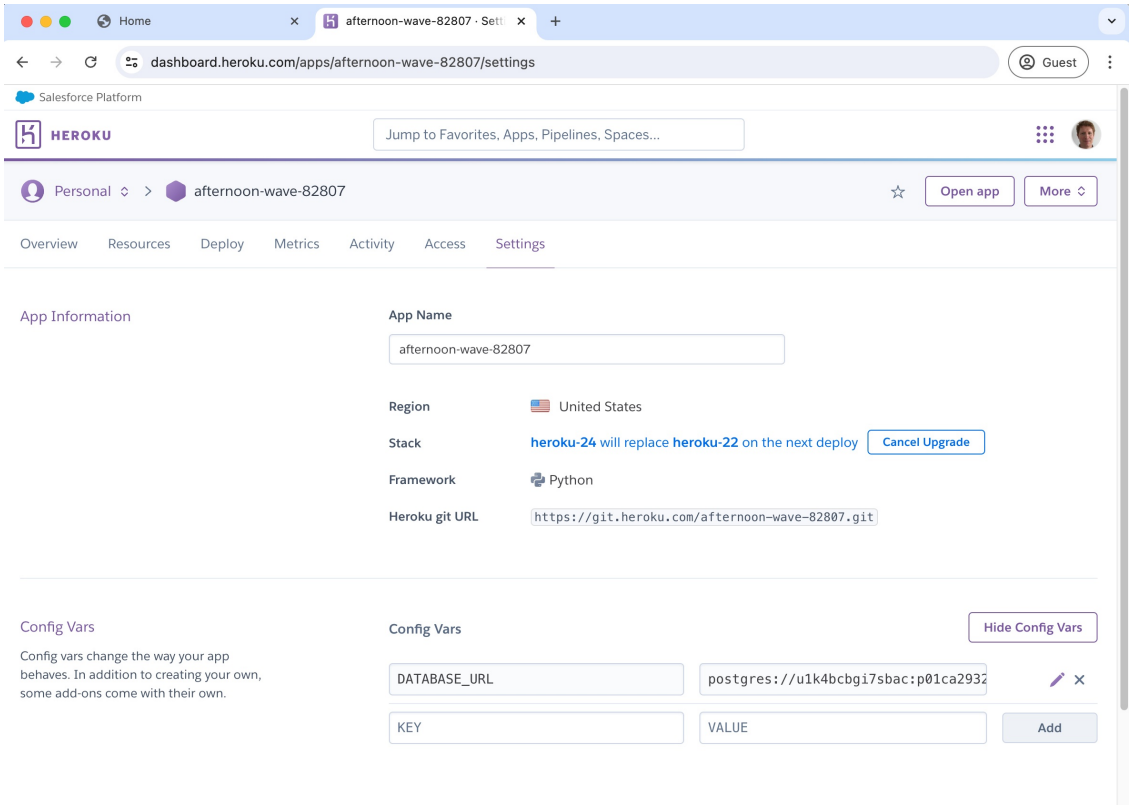
```
(.venv) $ heroku addons:info postgresql-sinuous-77120
=== postgresql-sinuous-77120
Attachments:  afternoon-wave-82807::DATABASE
Installed at: Tue Jul 02 2024 10:57:17 GMT-0400 (Eastern Daylight Time)
Max Price:    $5/month
Owning app:   afternoon-wave-82807
Plan:         heroku-postgresql:essential-0
Price:        ~$0.007/hour
State:        created
```

If you run `heroku config`, it will show all configuration variables set on Heroku. At the moment, that is just a `DATABASE_URL` with the information to connect to the production Postgres database.

Shell

```
(.venv) $ heroku config
=== afternoon-wave-82807 Config Vars

DATABASE_URL: postgres://u1k...us-east-1.rds.amazonaws.com:5432/d11ac0v0inabta
```

Select your project on the Heroku website dashboard and click “Settings” in the navigation bar. Under “Config Vars” you can see that the `DATABASE_URL` has been set.



Heroku Dashboard Configs

There are two other items in our local `.env` file, `DEBUG` and `SECRET_KEY`. We need to manually set both on Heroku, either in the web interface or the command line. First up is `DEBUG` which should be `False`.

Shell

```
(.venv) $ heroku config:set DEBUG=False
Setting DEBUG and restarting ⬢ afternoon-wave-82807... done, v6
DEBUG: False
```

Next is the `SECRET_KEY`. Make sure to wrap it in quotations, `""`, if you do so via the command line.

Shell

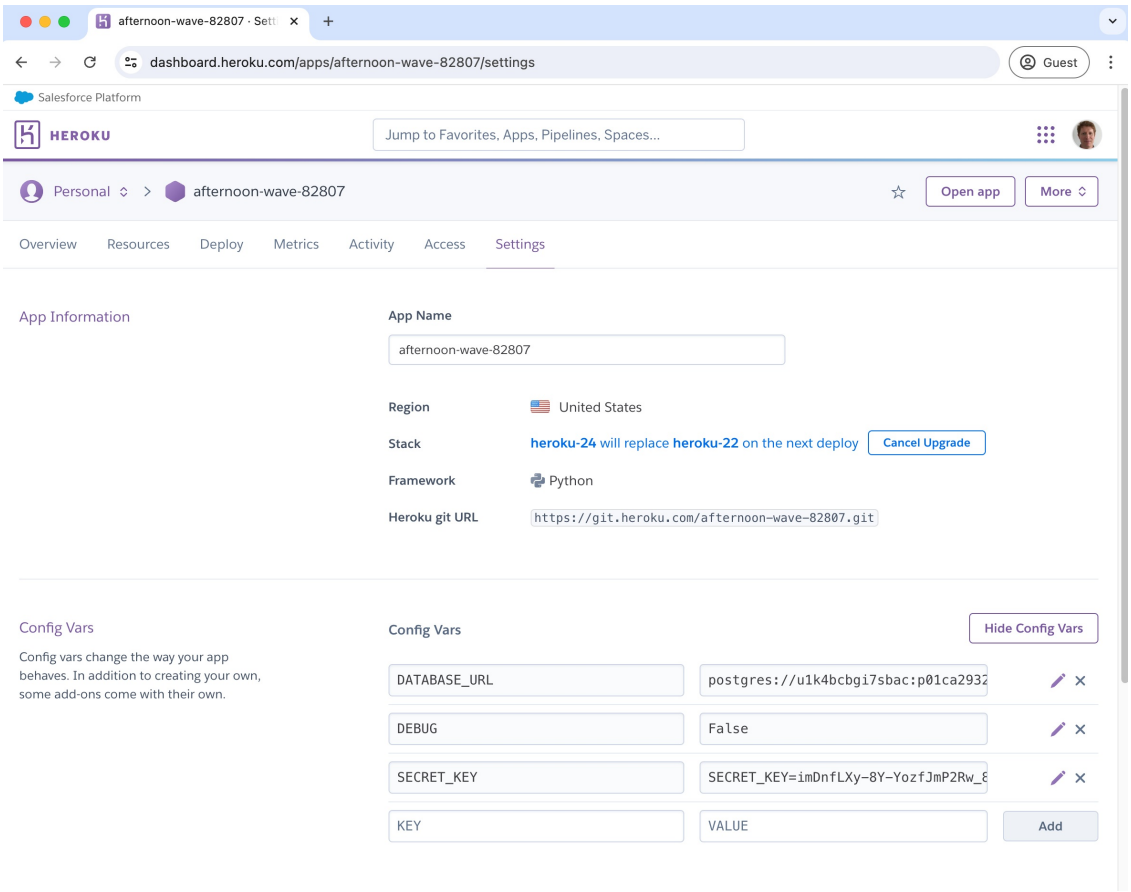
```
(.venv) $ heroku config:set SECRET_KEY="SECRET_KEY=imDnfLXy-8Y-YozfJmP2Rw_81YA_qx1X
  Kl5FeY0mXyY"
Setting SECRET_KEY and restarting ⬢ afternoon-wave-82807... done, v7
SECRET_KEY: SECRET_KEY=imDnfLXy-8Y-YozfJmP2Rw_81YA_qx1XKl5FeY0mXyY
```

It’s a good idea to double-check that the production environment variables are properly set. From the command line that means using the `heroku config` command.

Shell

```
(.venv) $ heroku config
=== afternoon-wave-82807 Config Vars

DATABASE_URL: postgres://u1k...us-east-1.rds.amazonaws.com:5432/d11ac0v0inabta
DEBUG:        False
SECRET_KEY:   SECRET_KEY=imDnfLXy-8Y-YozfJmP2Rw_81YA_qx1XKl5FeY0mXyY
```

You can also look at the web dashboard.



Heroku Dashboard Updated Configs

Now it is time to push our code up to Heroku with the command, `git push heroku main`. If we had just typed `git push origin main` the code would have been pushed to GitHub, not Heroku. Adding `heroku` to the command sends the code to Heroku.

Shell

```
(.venv) $ git push heroku main
Enumerating objects: 339, done.
Counting objects: 100% (339/339), done.
Delta compression using up to 10 threads
Compressing objects: 100% (333/333), done.
Writing objects: 100% (339/339), 798.15 KiB | 14.25 MiB/s, done.
Total 339 (delta 39), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (39/39), done.
remote: Updated 569 paths from 2ebafa9
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Building on the Heroku-22 stack
...
remote:        https://afternoon-wave-82807-b672795cd97e.herokuapp.com/
    deployed to Heroku
```

```
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/afternoon-wave-82807.git
 * [new branch]      main -> main
```

This command generates a lot of output from Heroku and might take a while the first time. We are pushing the code to Heroku, and it is rebuilding a production version of our Django project on its servers. You'll see that it installs each item from our `requirements.txt` file, among other actions.

The last step is starting a Dyno, Heroku's term for our app's lightweight container. We need at least one to be running to make our website live. If we start to see a spike in traffic, we could add more dynos to our project, and Heroku will handle all the infrastructure. For a small project like this, I recommend the Basic Dyno, which is $0.01 per hour with a maximum of $7 per month.

We will spin up one dyno using the Heroku CLI, but dynos can also be managed via the web interface. The general syntax for the CLI is to start with `heroku`, `ps` is a command that prefixes many commands affecting dynos, `ps:scale` is used to increase the number of dynos running a process. Therefore, the command below tells Heroku to run one dyno for our website.

Shell

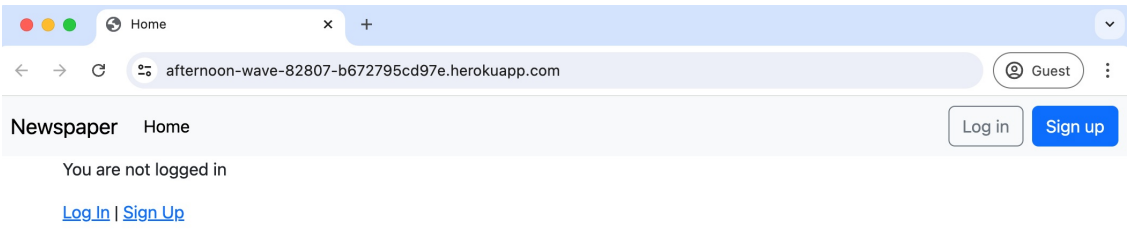
```
(.venv) $ heroku ps:scale web=1
Scaling dynos... done, now running web at 1:Basic
```

The total cost for our project–if we let it run all the time–is $12 per month: $5/month for the Postgres database and $7/month for the Dyno. Heroku bills per hour so you can always deploy the website and take it down after a few days, which should cost only cents.

We're done! The last step is to confirm our app is live and online. If you use the command `heroku open`, your web browser will open a new tab with the URL of your app:

Shell

```
(.venv) $ heroku open
```



Production Homepage

The Newspaper website is live, but you'll quickly see some problems if you try it out. For one thing, there are no articles or comments! That's because we still need to configure the production PosgreSQL database running on Heroku. Let's do that now. To run Django commands on Heroku instead of locally, we use the prefix `heroku run`. So to `migrate` our database with initial settings, we run the following command:

Shell

```
(.venv) $ heroku run python manage.py migrate
Running python manage.py migrate on ⬢ afternoon-wave-82807... up, run.2790 (Basic)
Operations to perform:
  Apply all migrations: accounts, admin, articles, 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 articles.0001_initial... OK
  Applying articles.0002_comment... OK
  Applying sessions.0001_initial... OK
```

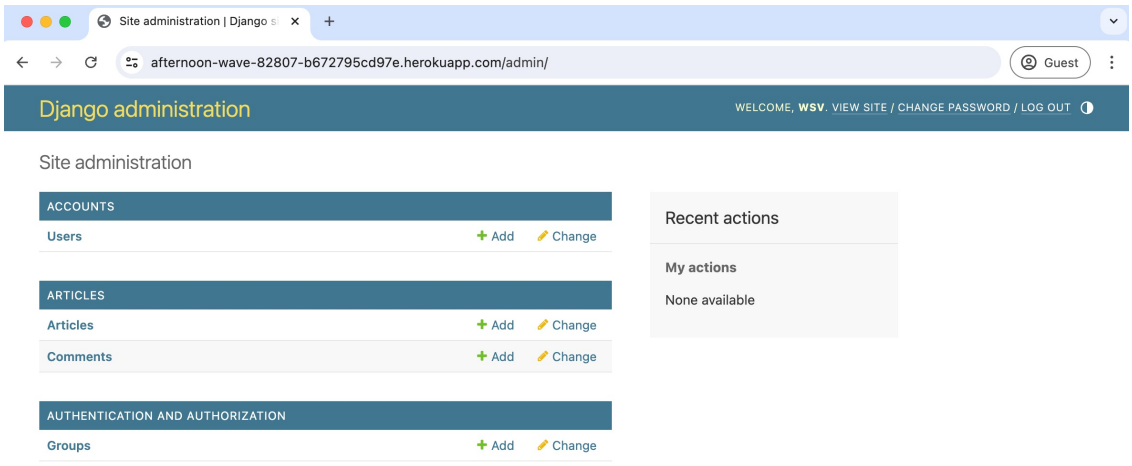
Now, create a superuser account to access the admin.

Shell

```
(.venv) $ heroku run python manage.py createsuperuser
Running python manage.py createsuperuser on ⬢ afternoon-wave-82807... up, run.7422
```

```
  (Basic)
Username: wsv
Email address: will@learndjango.com
Password:
Password (again):
Superuser created successfully.
```

Navigate to the admin section of your deployed website, log in with your superuser credentials, and add some articles and comments.



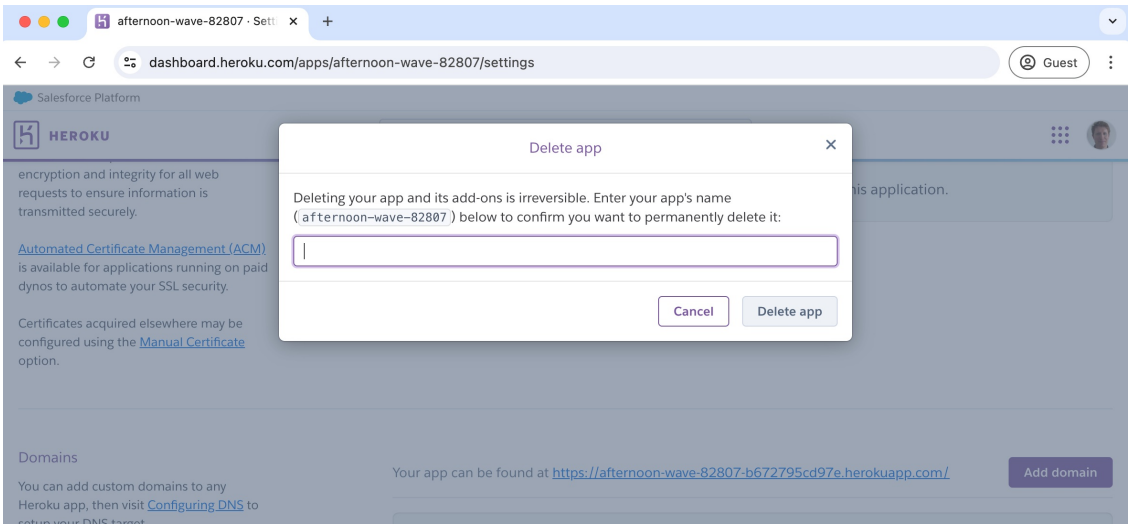
Production Admin Dashboard

They will then be displayed on the live website. You can also create user accounts and confirm that the user authentication flow works correctly by resetting your password.

For future updates to the production website, the pattern is as follows:

- make local code changes and save them with Git
- use `git push origin main` to deploy them to GitHub
- use `git push heroku main` to push the code to Heroku

If you want to remove a hosted website, log into your Heroku dashboard and click on the app name. Click on the Settings link in the navigation bar at the top, then scroll down to the bottom of the page under Delete App and click the “Delete App…” button. You will be asked to type in your full app name one more time to confirm that you want to permanently delete it.



Heroku Delete App

Another tip is that you can type `Ctrl + d` to exit the Heroku CLI at any time.

Additional Security Steps

There is an almost infinite list of additional security procedures to secure a production website. Our production checklist covers the basics, but there are more if you want to take the additional steps.

First, update `ALLOWED_HOSTS` and `CSRF_TRUSTED_ORIGINS` to use the exact production URL for your project.

Next, you would run Django's management command, which runs several automated checks around deployment. To run that command you would prefix `heroku run` so it would be `heroku run python manage.py check --deploy`. You now know how to reference the Django docs and update your local and production environment variables to make them pass.

Conclusion

We just covered a lot of new material, so you will likely feel overwhelmed. 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; use the deployment checklist!

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 makes it much smoother than it otherwise would be. You know enough to confidently deploy any Django project online with a PaaS.

Chapter 17: Conclusion

Congratulations on finishing *Django for Beginners*! Starting from scratch, we've built six 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 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? An Instagram 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.

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.

To stay current with the latest Django news, the Django News Newsletter is a free weekly newsletter with all the latest news, events, articles, tutorials, and projects. If you prefer the audio format, Django Chat is a biweekly podcast I co-host with Django Fellow Carlton Gibson, where we interview leading developers and provide deep dives on various Django topics.

If you want an all-in-one source of free tutorials and additional courses that cover APIs, Docker, testing, and other topics in more depth please check out LearnDjango.com, a comprehensive learning website I run focused exclusively on Django.

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 mindlessly 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 over 4,000 available third-party packages. A more curated option, the awesome-django repo, which I run with the current maintainer of Django Packages, is worth a look. And if you need help starting a new project quickly, I've long maintained a free starter project, DjangoX, that comes with the latest version of Django, built-in user authentication, and more to jumpstart any new projects.

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. Your review will have an enormous impact on book sales and help me continue to teach Django full-time.

Finally, 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@learndjango.com`.

Thank you for reading the book. Good luck on your journey with Django!