django

PYTHON FOR WEB DEVELOPMENT

django

Complete python web developer course for beginners

Emenwa Global

Django | Python for Web Development

Contents

INTRODUCTION

Welcome to the future. Reading this Django book is the best decision you will ever make this year. Python programming is taking over the world. The world of the web is more than impressive. You are here because you are interested in website development.

Django is exciting if you are interested in building websites. Django is a framework for making websites. It saves you a lot of time and makes building websites easier and more fun. Django makes it easy to build and maintain high-quality web applications.

Web development is a creative journey, fun-filled and full of adventure, with enough stress to last a day! Django lets you focus on your web application's fun and critical parts while making the boring parts easier. In other words, it makes it easier to create common programming tasks and abstract common web development patterns. It also gives clear rules for how to solve problems. It does all these things and allows you to work outside the framework's scope when needed. My goal with this book is to help you learn and master Django. I want you to go as far as you possibly want to go in and understand the Django web framework.

This book's structure will help you remain engaged until the very end because I picked it from the beginning. We're going to start with absolute basics. And then we're going to work our way up, introducing new concepts along the way.

And I'm not going to be building a whole project. So instead, I'm going to just jump into individual concepts and put them into a practical use case in a sort of project. And then, eventually, you'll have an excellent understanding of how to build real web applications using Django.

Of course, I recommend that you learn from entire projects, and there are many online. However, this book is all about getting the absolute basics to even advanced level things, bit by bit.

If you're learning Django for the first time, it's a fascinating experience, at least it was for me because once I was able to build a web application with a database attached to it, I felt like a demi-god!

Now, I think that you might be just as excited. And perhaps you've already done that with other web applications. Or maybe this is the first time you're programming. Either way, the most frustrating part is at the beginning. Not so much because of the programming language, but often because of how to set up your system. And since it's so frustrating, over the years, we have refined the way to set up your system, depending on what you're operating on. And that's part of the frustration is like, you're going to see me working a lot in a Windows OS environment. And if you're on macOS, you may look at the screenshots I share in this book and feel like, hey, why isn't it the same?

I mean, using Python and Django are the same on both systems. Because Python is Python, and Django is Django. The commands to get there might differ slightly, but it's the same realistically.

I will walk you through all the processes in this tutorial, step by step, and you will find it very easy. I will explain how to set it up on Windows and macOS.

And then the last thing is when in doubt, consult the documentation. Django's documentation is very well written. And there's just so much that

you can learn that we won't necessarily cover because they give additional context or specifics to whatever use case you have for the technology. And then the last thing is Google is your friend. You can use Google to search for something that you're not familiar with. The best of us use Google a lot.

And often, that will bring up stack overflow.com. Stack Overflow has all of these questions from people for all sorts of programming languages, including Python, Django, JavaScript, and all kinds of things.

So, a brief intro into some concepts before we dive right in.

What Is a Web Framework?

Before we talk about Django, we need to talk about web frameworks, which are very important in modern web apps. Let's look at how a Python application is coded when you don't use a framework to learn about web frameworks. A Common Gateway Interface is the best way to do this (CGI). You just need to make a script that outputs HTML and save it to a web server with a.cgi extension. Writing from scratch is probably the best way to go for simple pages. There is no other code to read, and the code is easier to understand. It's also easier to set up.

Even though the approach is simple, it has a few problems. What would you do, for example, if you needed more than one part of your app to connect to a database? Using the above method, you would have to put the code for connecting to the database into each CGI script twice. This can be hard to do, making it more likely that someone will make a mistake. Putting this code in a shared function would be easier, though. When the same code is used in different environments, each with its own password and database, it will need to be set up for each virtual environment.

Also, if you haven't used Python much, you're more likely to make small mistakes that can cause the program to crash. The page's logic and HTML display elements should be kept separate, so the editor can change one without changing the other.

A web framework solves these problems by giving programmers a place to start building applications. This lets you focus on writing code that is easier

to understand and keep up to date. Django does this as well.

Meet Django

Django is a high-level Python web framework that facilitates quick development and clean, practical design. It was made by experienced developers and takes care of a lot of the trouble of web development, so you can focus on writing your app instead of starting from scratch. It's free, and anyone can use it. Django is a free and open-source Python-based web application framework for building the back end of websites and web apps.

It uses the architectural pattern called Model View Template (MVT). It divides the code into the Model, the View, and the Templates. The developers only have to write the code for what should be shown to the user. Django will take care of everything else.

The main goal of Django, which the Django Software Foundation runs, is to make it easier to create complex, database-driven websites.

Some of the biggest companies in the world, like Google, YouTube, NASA, and Instagram, use Django for web development. This is primarily because of its robust and practical design, with features like ORM, Django's template language, scalability, flexibility, the Django admin panel, and Python's ease of use.

What is Django Used For?

Django's ORM layer is powerful. It speeds up development by streamlining data and database management. Without ORM, web developers would have to form tabular displays and explain operations or queries, which can result in a large amount of SQL that is difficult to track.

Its models are all in one file, unlike other web frameworks. In larger projects, models.py may be slow.

Django supports multiple databases.

SQLite can be used for development and testing without additional software.

Production databases are PostgreSQL or MYSQL.

For a NoSQL database, use MongoDB with Django.

In this book, you will walk with me as we create really awesome web development projects, CRUD (wiki style) blogs, and so on. The best way to learn is by doing. So, follow along on your computer as you read the steps and keep up with me. You don’t have to cram every step. You will always have this book as a reference. Follow me.

Chapter 1 - Installing to Get Started

Alright, so in this one, we will create a new virtual environment and install Django. Django is essentially a Python code. That means Python must be installed before installing Django.

This chapter explains how to configure Windows or macOS for Django projects. Developers use the Command Line to install and configure Django projects.

This chapter shows you how to set up your Windows or macOS computer correctly so you can work on Django projects. We start by giving an overview of the Command Line, powerful text-only interface developers use to install and set up Django projects. Then we install the most recent version of Python, learn how to set up virtual environments that are only used for one thing, and install Django. As the last step, we'll look at how to use Git for version control and a text editor. By the end of this chapter, you will have set up a Django project from scratch.

Introducing the Command Line

The command line is that blank screen you see in hacker movies where they type matrices. It is how coders and software developers interact with the computer while most people use a mouse or finger. We use it to run programs, install software, and connect to cloud servers. Most developers find that the command line is a faster and more powerful way to move around and control a computer after a bit of practice.

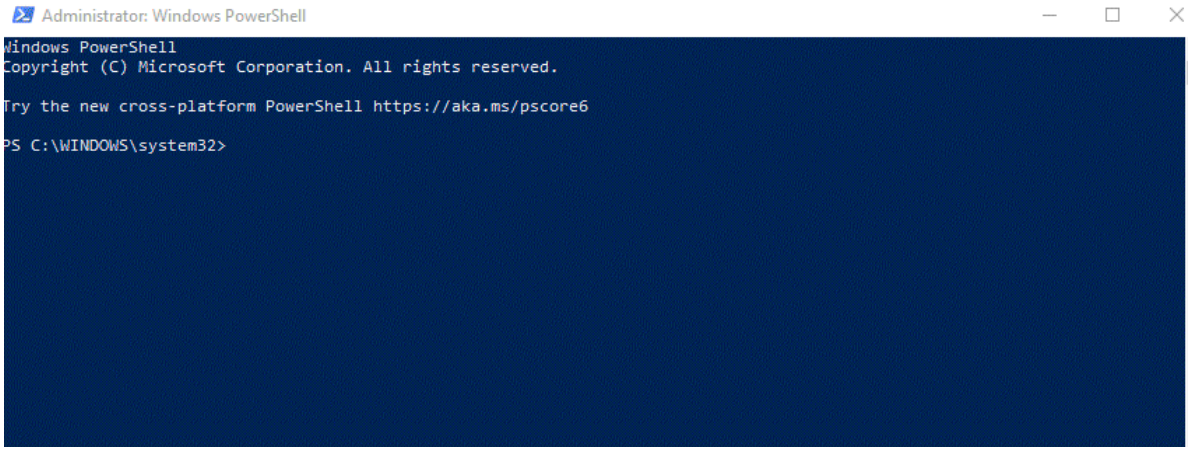
The command line is scary for people who have never used it because it only has a blank screen and a blinking cursor. After a command has run, you often don't get any feedback. You can wipe an entire computer with a single command without a warning if you're not careful. Because of this, the command line should only be used with care. Make sure not to just copy and paste commands you find online. If you don't fully understand a command, only use trusted sources.

In real life, the command line is also called the console, terminal, shell, prompt, or Command Line Interface (CLI). Technically, the terminal is the program that opens a new window to access the command line.

A **console** is a text-based application; a shell is a program that runs commands on the underlying operating system; a **prompt** is where you type commands and run.

Are there terms confusing? Haha. They all mean the same thing: the command line is where we run and execute text-only commands on our computer.

PowerShell is the name of both the built-in terminal and shell on Windows. To get to it, press the Windows button and type "PowerShell" to open the app. After the > prompt, it will open a new window with a dark blue background and a blinking cursor. On my computer, it looks like this.



Before the prompt is PS, which stands for PowerShell. Then comes the Windows operating system's initial C directory, followed by the Users directory and the current user, SYSTEM32, on my computer. Your username will be different, of course. Don't worry about what's to the left of the > prompt right now. It will be different on each computer and can be changed later. From now on, Windows will use the shorter prompt >.

The built-in terminal on macOS is called Terminal, as it should be. You can open it with Spotlight by pressing the Command key and the space bar at the same time, then typing "terminal." You can also open a new Finder window, go to the Applications directory, scroll down to the Utilities folder,

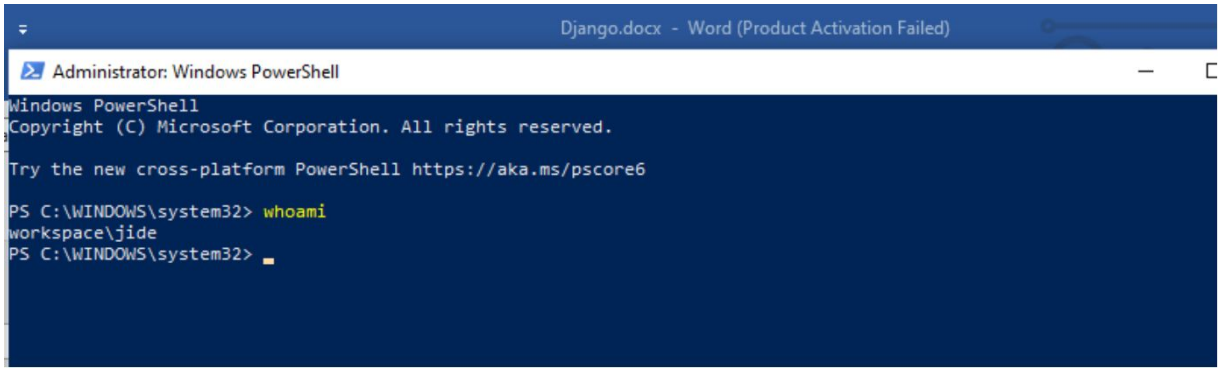
and double-click the Terminal application. After the "%" prompt, it opens a new screen with a white background and a blinking cursor. Don't worry about what comes after the percent sign. It's different for each computer and can be changed in the future.

If your macOS prompt is $ instead of %, then Bash is used as the shell. The default shell for macOS changed from Bash to zsh in 2019. Most of the commands in this book can be used with either Bash or zsh. If your computer still uses Bash, you should look online to learn how to switch to zsh through System Preferences.

Shell Commands

There are a lot of shell commands, but most developers use the same few over and over and look up more complicated ones when they need them.

Most of the time, the commands for macOS and Windows (PowerShell) are the same. On Windows, the whoami command shows the computer name and user name. On macOS, it only shows the user name. Type the command and press the return key as with any other shell command.



But sometimes, the shell commands on Windows and macOS are very different from each other. One good example is the primary "Hello, World!" command. " message to the terminal. On Windows, the command is called Write-Host, and on macOS, it is called echo.

Using the computer's filesystem is a task that is often done at the command line. The default shell should show the current location on Windows, but Get-Location can also be used to do this. Use pwd on Mac OS (print working directory).

You can save your Django code wherever you want, but for ease of use, we'll put ours in the desktop directory. Both systems can use the command cd followed by the location you want to go to.

```
cd OneDrive\Desktop
```

OR

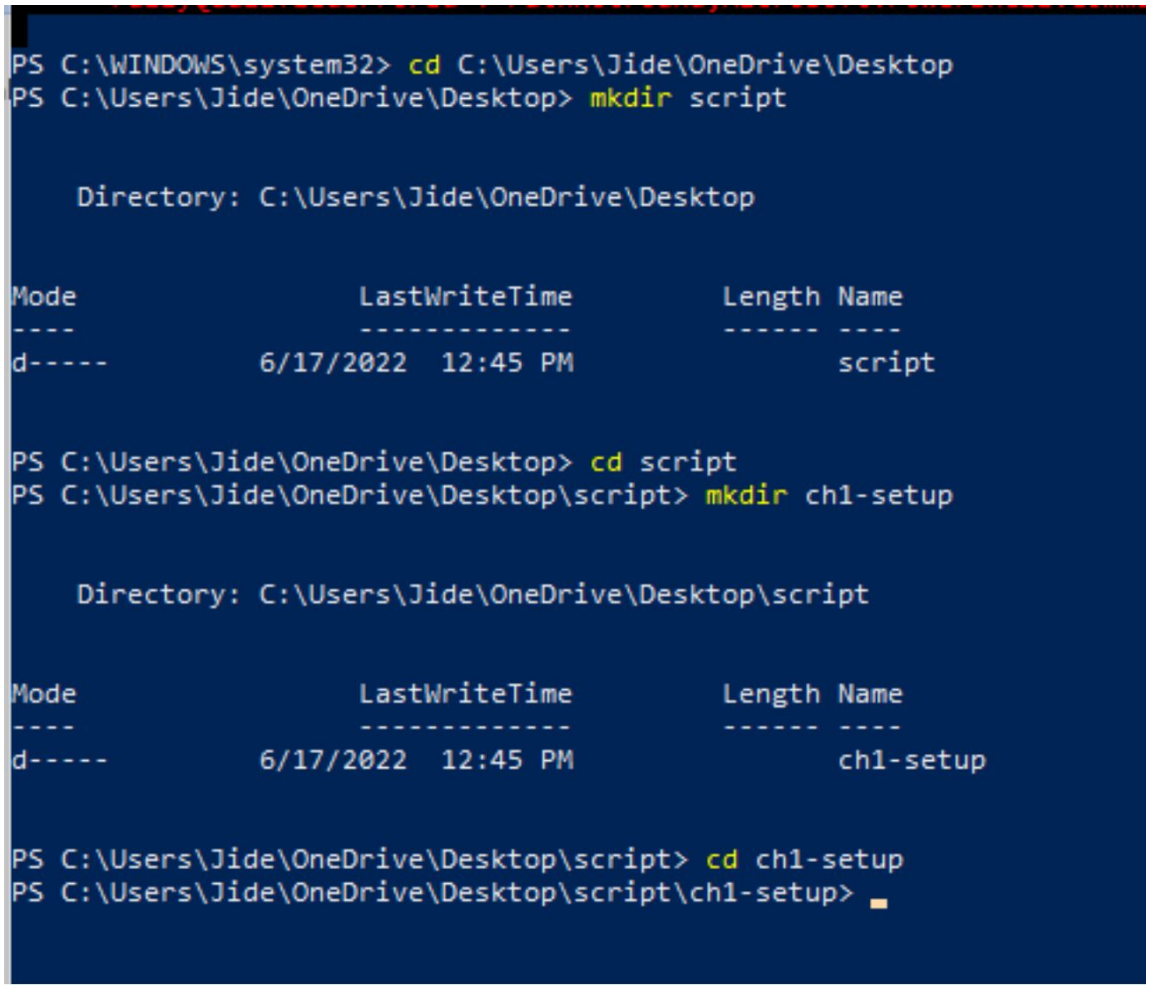
```
% cd desktop
```

On macOS

You can use the command mkdir to create a new folder. We want to create a folder called script on the Desktop. We will keep another folder inside it called ch1-setup. Now here is the command line to do all of these:

```
> mkdir code
> cd code
> mkdir ch1-setup
> cd ch1-setup
```

Press enter after each line, and you will get something like this:



I love to believe that you have installed Python on your computer. If you haven't, please head on to Python's official website and install Python. You will find the latest version of Python on the official website. After installing Python, you have to set up your system for Django.

To verify that you have Python installed on your Windows or Mac system, open your command prompt and type in the following code:

```
Python --version
```

Once you press Enter, the version of Python you have installed on your computer will show. If it doesn't, go ahead and install Python.

Once you have verified the installation of Python, you can now install Django.

Virtual Environments

Django's purpose is now clear to you. One of the most common issues with Django is that a project built in one version may not be compatible with one created in another. You may run into issues if you upgrade from a version of Django 1.5x to Django 1.6x.

Installing the latest versions of Python and Django is the right way to start a new project. Let’s say you created a project last year and used older versions of Python and Django. Now, this year you want to use Django 4.0. You may have to reinstall the version you used in creating that project at the time to open it.

Python and Django are installed globally on a computer by default, making it a pain to install and reinstall different versions whenever you want to switch between projects.

This problem can be easily solved if you use Django's version across all your projects. That is why creating a virtual environment with its own set of installation folders is essential.

You can easily create and manage separate settings for each Python project on the same computer using virtual environments. Otherwise, any changes you make to one website in Django will affect all the others.

There are many ways to set up virtual environments, but the easiest is to use the venv module, which comes with Python 3 as part of the standard library. To try it out, go to the ch1-setup directory that is already on your Desktop.

```
cd onedrive\desktop\code\ch1-setup
```

Use the following command line to create a virtual environment

```
python -m venv <name of env>
```

on Windows or

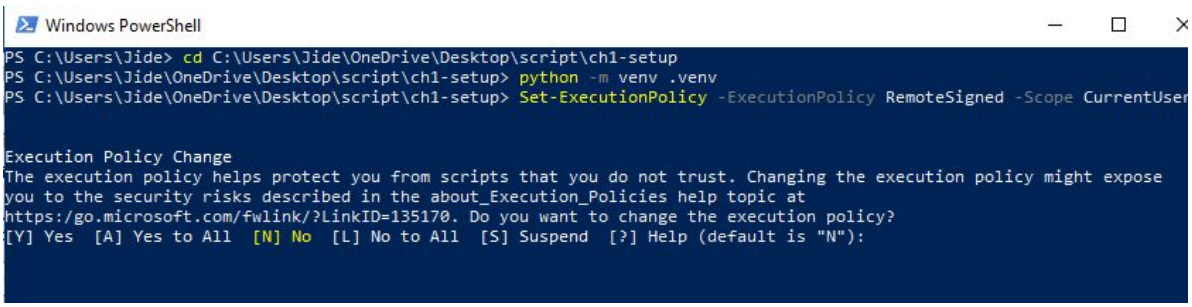
```
python3 -m venv <name of env>
```

on macOS

It is up to the developer to choose a good name for the environment, but .venv is a common choice.

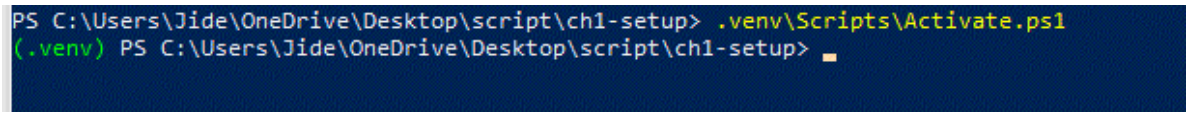
After that, if you are on Windows, type in the following:

Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser



After creating a virtual environment, we need to turn it on. For scripts to run on Windows, we must set an Execution Policy for safety reasons. The Python documentation says that scripts should only be run by the CurrentUser, and that is what that second line does. On macOS, scripts are not limited in the same way, so you can run source.venv/bin/activate immediately.

.venv\Scripts\Activate.ps1



For Mac users:

source.venv/bin/activate

As you can see in that screenshot, the environment name (.venv) is now added to the shell prompt. That shows that the virtual environment is

activated. Any Python packages that are installed or updated in this location will only work in the active virtual environment.

Now we can install Django.

Installing Django

Now, with the virtual environment active, we can install Django with this simple command line:

```
py -m pip install Django
```

That line will download and install the latest Django release.

Please consult the Django official website here if you have any issues installing Django.

Setup your Virtual Environment for Django on macOS/Linux

Now, I want you to have a new virtual environment and a fresh Django install, not only just to get the practice of it but also to make sure that we're all starting from the exact same spot. So if you open up your terminal window, or if you're on Windows, your PowerShell, or command prompt.

So if we type out Python -V in the Terminal, you will get the version of Python you have on your Mac or Linux computer. If you don’t have Python 3 installed, go to the official Python website to get it on your MacOs.

Now, we need the Virtual Environment. Introducing…

Installing Pipenv Globally

Now the first thing you need to get your installation of Django to work on all projects is to install a virtual environment. The best way to do this on Mac is by installing pipenv.

First, open Terminal and upgrade pip with the following command line:

```

python3 -m pip install pip --upgrade

```

This will upgrade whatever pip version is in your system. After this, you can install pipenv to use Django:

```

python3.8 -m pip install pipenv

```

This will essentially install the virtual environment. You can verify it by using the following line:

Now, you can install Django with a single line:

$ python -m pip install Django

Your First Blank Django Project

The way to create a blank website on Django is to first get the name of the site and then type in the following command:

django-admin startproject mysite .

Where mysite is the name of your project. You can use almost any name, but we will use mydjango in this book. This is what the command line loos like:

(.venv) PS C:\Users\Jde\OneDrive\Desktop\script\ch1-setup> django-admin startproject mydjango .

Now, let's ensure everything is working by running the runserver command to run Django's internal web server. This is good for developing locally, but when it's time to put our projects online, we'll switch to a more robust WSGI server like Gunicorn.

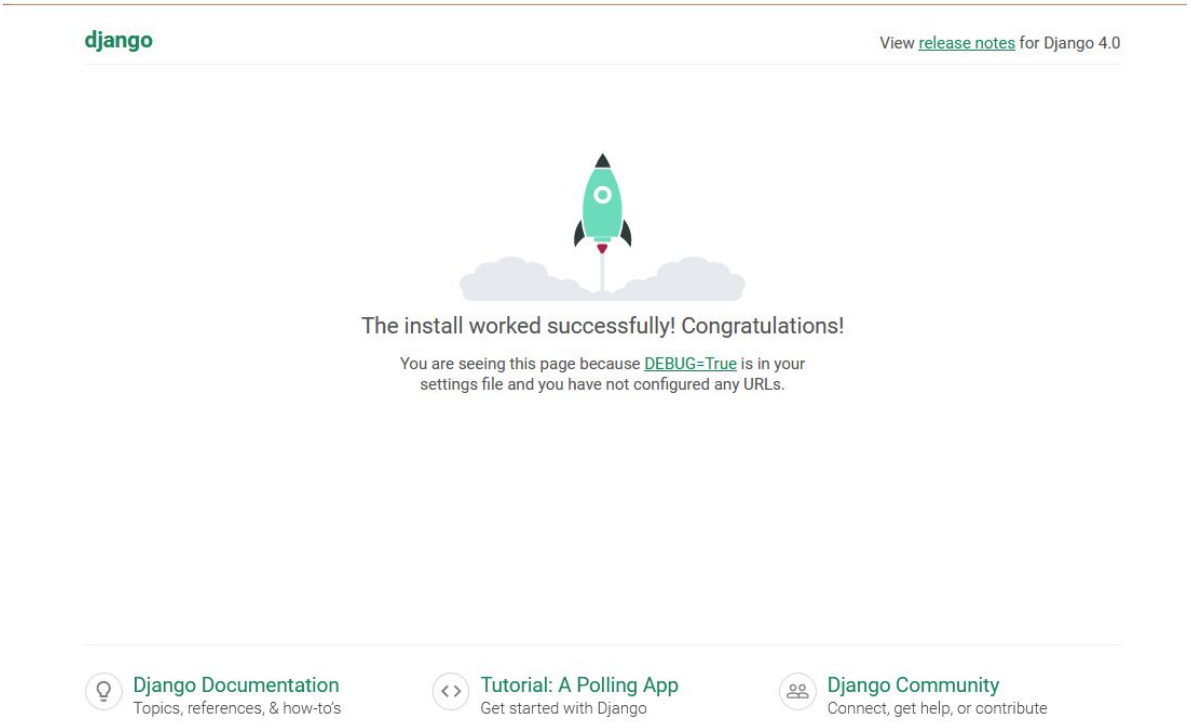
Type in the following command:

python manage.py runserver

```
(.venv) PS C:\Users\Jide\OneDrive\Desktop\script\ch1-setup> 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 17, 2022 - 15:23:39
Django version 4.0.5, using settings 'mydjango.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

If you do this, you have successfully created a website. Check now by opening your web browser and typing the following in the URL http://127.0.0.1:8000/.

You should see the following:



Well done! You have successfully created your first Django project on a local server. Stop the local server by typing the correct command. Then, leave the virtual environment by pressing "deactivate" and pressing Enter.

See you in the next chapter, where we will create a website with some words.

This book will give us a lot of practice with virtual environments, so don't worry if it seems complicated right now. Every new Django project follows the same basic steps: make and turn on a virtual environment, install Django, and run startproject.

It's important to remember that a command line tab can only have one virtual environment open at a time. In later chapters, we'll make a new virtual environment for each new project, so when you start a new project, make sure your current virtual environment is turned off or open a new tab.

Introducing Text Editors

You have met command lines. That is where we run commands for our programs, but expert developers write code in a text editor. There are many different text editors you can use. The computer doesn't care what text editor you use because the end result is just code, but a good text editor can give you helpful tips and catch typos for you.

There are many modern text editors, but Visual Studio Code is very popular because it is free, easy to install, and used by many people. If you don't already have a text editor, you can get VSCode from the website and install it.

Setting Up Django on VS Code

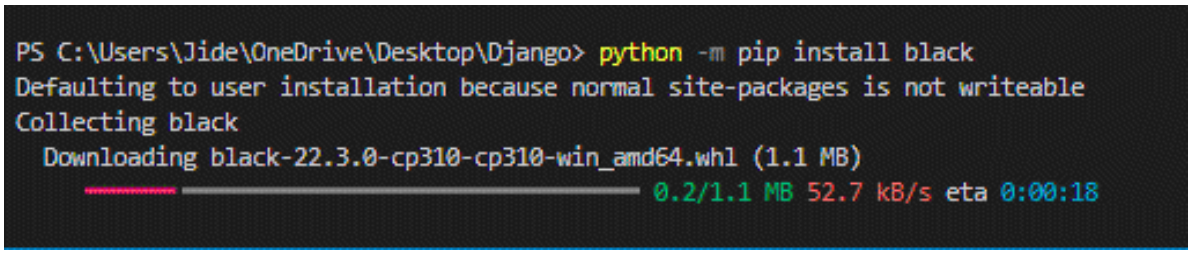
We will set up our Django project on VS Code or Visual Studio Code. If you don’t have that app, go to code.visualstudio.com and download the version for your machine. It is free and cross-platform. It also has a vast community of people or developers that build all sorts of great things for it.

Note that this is not the same as Visual Studio. Visual Studio is a different kind of text editor. There are other types of text editors like Sublime Text and Pycharm, but Visual Studio Code or VS Code is my favorite.

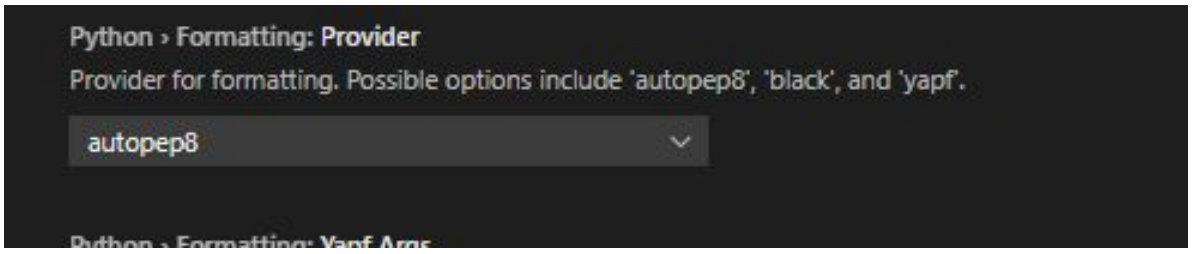
Open up VS code, and we will start our new project. So you're going to see a welcome screen. You will need to install a couple of extensions in VSCode.

Go to the Extensions tab. Search Python and install the first result with the highest number of downloads. After that, you need to install Black. To do this, go to Terminal, and click on New Terminal. From there, type in the following command:

```
python -m pip install black
```



Next, go to File > Preferences > Settings on Windows or Code > Preferences > Settings on macOS to open the VSCode settings. Look for "python formatting provider" and then choose "black" from the list. Then look for "format on save" and make sure "Editor: Format on Save" is turned on. Every time you save a .py file, Black will now format your code for you.



Go to the Explorer tab to confirm that Black and Python are working. Find Desktop, and open your ch1-setup folder. Create a new file and name it hello.py. On the new page, type in the following using single quotes:

```
print('Hello, World!')
```

Press CTRL + S to save and see if the single quotes change to double. If it changes, that is Black working.

Lastly, Git

The last step is to install Git, which is a version control system that modern software development can't do without. Git lets you work with other developers, keep track of all your work through "commits," and go back to any version of your code, even if you accidentally delete something important.

On Windows, go to https://git-scm.com/, which is the official site, and click on "Download." This should install the correct version for your computer. Save the file, then go to your Downloads folder and double-click on the file. This will start the installer for Git on Windows. Click "Next" through most of the early defaults, which are fine and can be changed later if necessary. There are two exceptions, though. Under "Choosing the default editor used by Git," choose VS Code instead of Vim. And in the section called "Changing the name of the initial branch in new repositories," select the option to use "main" as the default branch name instead of "master." If not, the suggested defaults are fine; you can always change them later if necessary.

To ensure that Git is installed on Windows, close all shell windows and open a new one. This will load the changes to our PATH variable. Then type the following

```
git –version
```

This will show the version you have installed.

For MacOs, you can install Git with XCode. First, open your Terminal. Type the following:

```
git –version
```

There should be a message that git is not found, and there will be a suggestion to install it. Or you could just type in xcode-select –install to install it directly.

Once installed, you need to set it up and register a new account, and you are good to go!

Chapter 2 - Create Your First Django Project

In this chapter, we'll build a Django website. Our website will have a simple homepage that says, "Welcome to my website." Let’s get started.

Setup

To start, fire up a new command prompt window or use VS Code's in-built terminal. The latter can be accessed by selecting "Terminal" from the menu bar and "New Terminal" from the drop-down menu.

Verify that you are not in a preexisting virtual environment by ensuring that the command prompt does not have any parentheses. To be sure, type "deactivate," and you'll be turned off. You can then use the following commands in the code directory on your Desktop to make a helloworld folder for our new website.

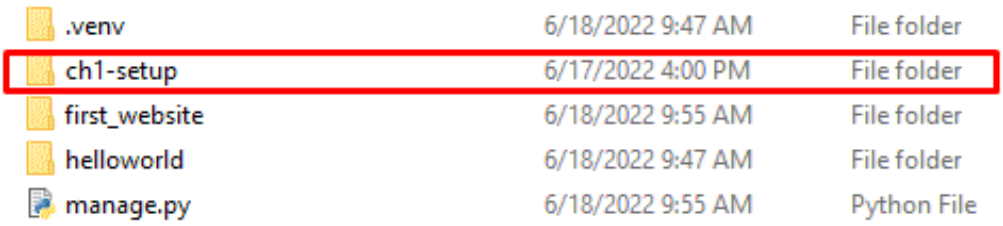
```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\Jide\OneDrive\Desktop> cd  ^C
PS C:\Users\Jide\OneDrive\Desktop> cd C:\Users\Jide\OneDrive\Desktop\script
PS C:\Users\Jide\OneDrive\Desktop\script> mkdir helloworld


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         6/18/2022   9:47 AM                helloworld


PS C:\Users\Jide\OneDrive\Desktop\script> python -m venv .venv
PS C:\Users\Jide\OneDrive\Desktop\script> .venv\Scripts\Activate.ps1
(.venv) PS C:\Users\Jide\OneDrive\Desktop\script> python -m pip install django~=4.0.0
Collecting django~=4.0.0
  Using cached Django-4.0.5-py3-none-any.whl (8.0 MB)
Collecting sqlparse>=0.2.2
  Using cached sqlparse-0.4.2-py3-none-any.whl (42 kB)
```

As you can see in the above screenshot, there is the first code to call in the folder we have created in the previous chapter called scripts, and we made

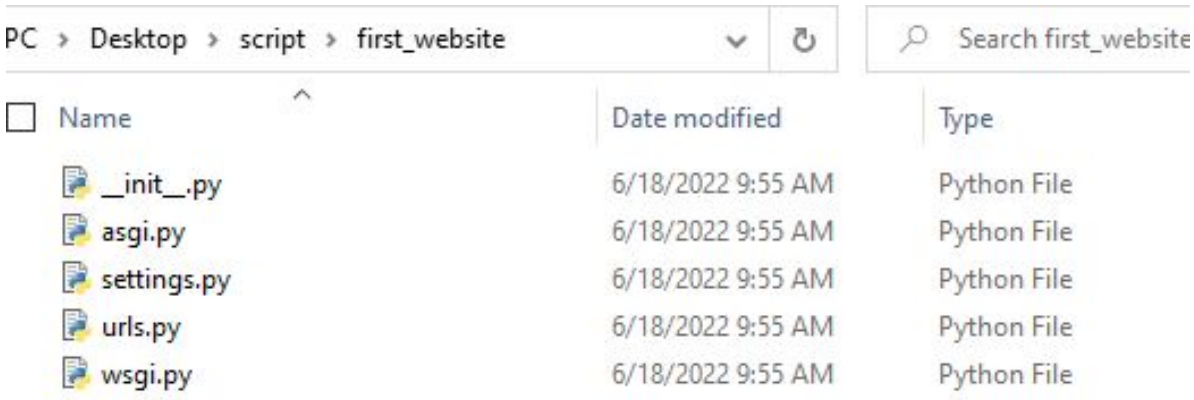
another folder within it called helloworld. Then we activated our virtual environment and installed the version of Django we wanted to use for this project.

From here, you should remember the Django `startproject` command. This command will create a new Django project. Let us call our new project first_website. Include the space + full stop (.) at the end of the command so that the program will be installed in the current folder.

The most important thing for a web developer is the project structure. You need an organized space and directories for all your projects and programs. Django will automatically set up a project structure for us in this script. If you want to see what it looks like, you can open the new folder on your Desktop. The Ch1-setup is the folder from chapter 1. We don't need that now.



However, you can see that the `.venv` folder was created with our virtual environment. Django added the first_website folder and python file. If you open the first_website folder, you will find 5 new files:



__init__.py shows that the folder's files are part of a Python package. We can't install files from another folder without this file, which we will do a lot in Django.

asgi.py offers the option of running an Asynchronous Server Gateway Interface.

settings.py manages the settings of our Django project.

urls.py tells Django what pages to make when a browser or URL asks for them.

wsgi.py stands for Web Server Gateway Interface. WSGI helps Django serve our web pages.

The manage.py file is not a core component of the Django project, but it is used to run Django commands like starting the local web server or making a new app.

Let's test our project using the light web server with Django for local development. The runserver command will be used. It can be found in the file manage.py. Type in this command:

```
python manage.py runserver
```

Once that runs, you can test your server by going to this with your web browser: http://127.0.0.1:8000/

```
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 migratio
ns for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
June 18, 2022 - 10:11:46
Django version 4.0.5, using settings 'first_website.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

You may see the error in the above screenshot too. Don’t fret. That is Django telling you that we haven't made any changes to our existing database (i.e., "migrated") yet. This warning is harmless because we won't use a database in this chapter.

But if you want to stop the annoying warning, you can get rid of it by pressing Control + c to stop the local server and then running the following command line:

python manage.py migrate.

```
(.venv) PS C:\Users\Jide\OneDrive\Desktop\script> 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
```

Django has migrated its pre-installed apps to a new SQLite database. The equivalent file in our folder is called db.sqlite3.

Warnings should now be gone if you rerun python manage.py runserver.

Let us learn a few concepts you need to know before building our first Django app together.

HTTP Request/Response Cycle

A network protocol is a set of rules for formatting and processing data. It's like a common language for computers that lets them talk to each other even if they are on opposite sides of the world and have very different hardware and software.

HTTP is a protocol that works with a client-server model of computing. When you go to a website, your computer, or "client," sends a "request," and a "server" sends back a "response." The client doesn't have to be a computer, though. It could be a cell phone or any other device that can connect to the internet. But the process is the same: a client sends an HTTP request to a URL, and the server sends an HTTP response back.

In the end, a web framework like Django takes HTTP requests to a given URL and sends back an HTTP response with the information needed to render a webpage. All done. Usually, this process involves finding the correct URL, connecting to a server, logic, styling with HTML, CSS, JavaScript, or static assets, and then sending the HTTP response.

This is what the abstract flow looks like:

HTTP Request -> URL -> Django combines database, logic, styling -> HTTP Response

Model-View-Controller (MVC) and Model-View-Template (MVT)

The Model-View-Controller (MVC) sequence has become a popular way to split up an application's data, logic, and display into separate parts over time. This makes it easier for a programmer to figure out what the code

means. The MVC pattern is used by many web frameworks, such as Ruby on Rails, Spring (Java), Laravel (PHP), ASP.NET (C#), and many others.

There are three main parts to the traditional MVC pattern:

Model: Takes care of data and the primary project logic

View: Gives the model's data in a specific format.

Controller: Takes input from the user and does application-specific logic.

Django's method, often called Model-View-Template, only loosely follows the traditional MVC method (MVT). Developers who have worked with web frameworks before might find this confusing at first. In reality, Django's approach is a 4-part pattern that also includes URL Configuration. A better way to describe it would be something like MVTU.

Here's how the Django MVT pattern works:

Model: Manages data and core business logic

View: Tells the user what data is sent to them, but not how it is shown.

Template: Shows the information in HTML, with CSS, JavaScript, and Static Assets as options.

URL Configuration: Regular-expression components set up for a View

This interaction is a crucial part of Django, but it can be hard to understand for new users, so let's draw a diagram of how an HTTP request and response cycle works. When a URL like https://djangoproject.com is typed in, the first thing that happens in our Django project is that a URL pattern (contained in urls.py) that matches it is found. The URL pattern is linked to a single view (in views.py) that combines the data from the model (in models.py) and the styling from a template (any file ending in .html). After that, the view gives the user an HTTP response.

The flow looks like below:

HTTP Request -> URL -> View -> Model and Template -> HTTP Response

Creating A Blank App

Django uses apps and projects to keep code clean and easy to read. Multiple apps can be part of a single Django project. Each app will have a set of functions to control. For example, to build an e-commerce site, you may use one app to log in users, another to handle payments, and another to list item details. That's three different apps that are all part of the same main project.

You must activate the virtual environment to add a new app to your project. Do you still remember how to do that?

Type in one of the following lines on your Windows or Mac:

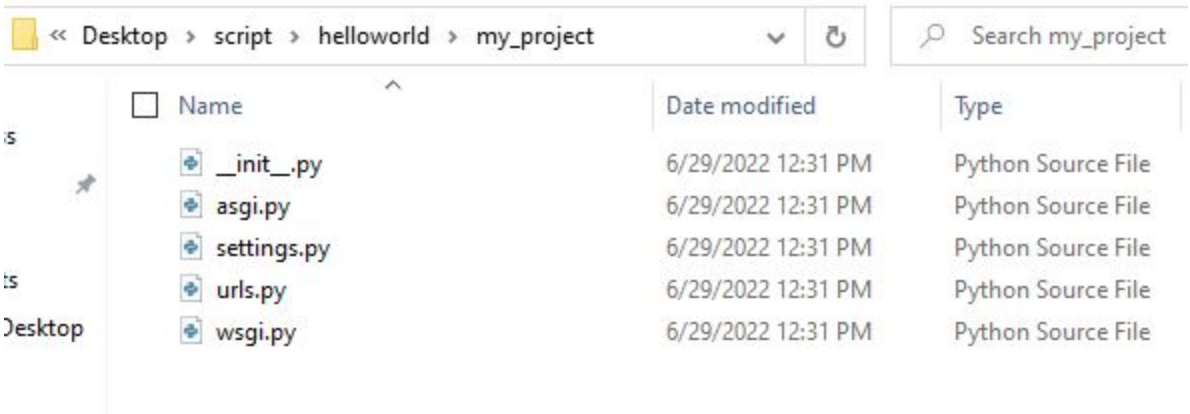
```
.venv\Scripts\Activate.ps1
```
OR

```
source .venv/bin/activate
```

We will create a new project (or folder) in our Scripts directory. Let us call it my_project. Remember to put the space and full stop (.) at the end of the command so that it is installed in the current folder we are working in.

```
django-admin startproject my_project .
```

Let's take a moment to look at the new folders that Django has set up for us by default. If you want to see what it looks like, you can open the new my_project folder on the Desktop. You may not see the.venv folder because it is hidden.



Let's try out our new project using the light web server with Django for local development. The runserver command will be used. It can be found in the file manage.py. Use the following line:

python manage.py runserver
OR

python3 manage.py runserver

Now visit http://127.0.0.1:8000/ on your web browser to test the server. Don’t worry about the migration error. You know it. Let’s fix it. Type in the following:

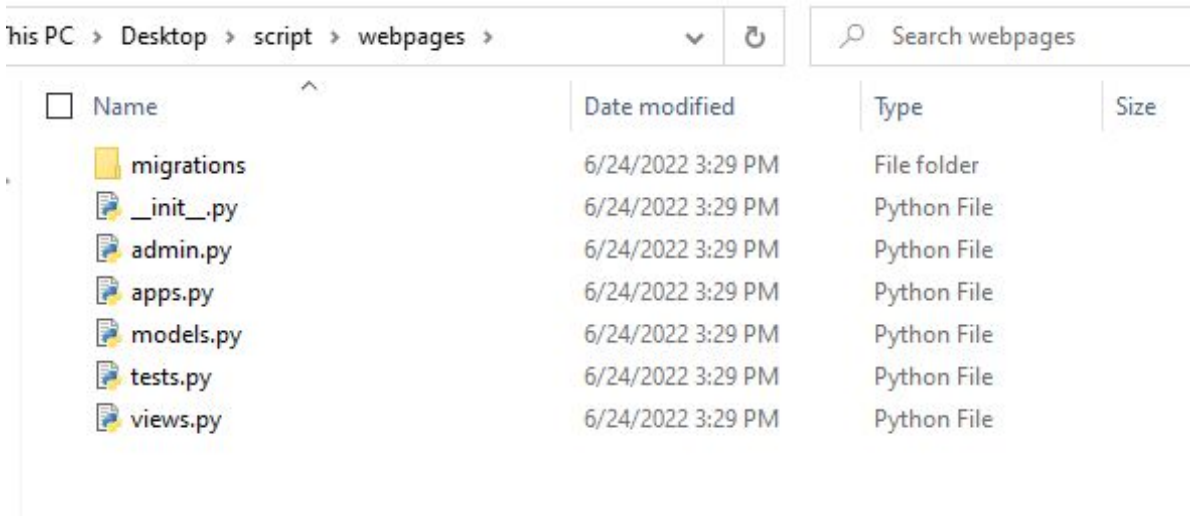
python manage.py migrate

Let us put our app up in there.

If you have a running server, you must deactivate it by pressing Ctrl + C. You then use the Django startapp command to create the new project and follow it by the name of your new app. I will call my app webpages.

python manage.py startapp webpages

If you look at the folder we have been using, you will find the new folder for webpages:



Let's go over what each new webpages app file does:

admin.py is a file that tells the Django Admin app how to work.

apps.py is a file that tells the app how to work and migrations/ keeps track of changes to our models.py file so that it stays in sync with the models in our database.

models.py is where our database models are written, and Django automatically turns them into database tables and tests.

tests.py is for testing views in an app.

views.py is where we handle the logic for our web app's requests and responses.

Notice that the MVT pattern's model, view, and URL are there from the start. Only a template is missing, which we'll add soon.

Even though our new app is part of the Django project, we still have to make Django "know" about it by adding it to the `my_project/settings.py` file. Open the file in your text editor and scroll down to where it says "INSTALLED APPS." There are already six Django apps there.

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

At the end, add webpages.apps.WebpagesConfig.

What is PagesConfig? The only thing you have to know at this point is that this is a function that we call from the apps.py file that Django created in the webpages folder.

Designing Pages

Web pages on the internet are linked to a database. To power a single dynamic web page in Django, you need four separate files that follow this MVT pattern:

models.py

views.py

templates.html (any HTML file will do)

urls.py

Since our project today does not need to connect to a database, we can simply hardcode all the data into a view and skip the MVT model. That is what we will do now. This means everything you do on your end can only be accessed from your computer.

So, the next thing to do is to make our first page (view). Open the views.py file in the webpages folder and edit the code like this:

```
from django.shortcuts import render
```

```
# Create your views here.
from django.http import HttpResponse

def homePageView(request):
    return HttpResponse("My New App!")
```

Basically, we're saying that whenever we call the function homePageView, Django should display the text "My New App!" In particular, we've imported the built-in HttpResponse method so that we can give the user a response object. We made a function called homePageView that takes the request object and sends back the string "My New App!" as a response.

Function-based views (FBVs) and class-based views are the two types of views in Django (CBVs). In this example, our code is a function-based view. It is clear and easy to implement. Django started out with only FBVs, but over time it added CBVs, which make it easier to reuse code, keep things DRY (Don't Repeat Yourself), and allow mixins to add more functionality. The extra abstraction in CBVs makes them very powerful and short, but it also makes them more complicated for people who are new to Django to read.

Django has a number of built-in generic class-based views (GCBVs) to handle common use cases like creating a new object, forms, list views, pagination, and so on. This is because web development tends to be repetitive. In later chapters of this book, we will use GCBVs a lot.

So, technically, there are three ways to write a view in Django: function-based views (FBVs), class-based views (CBVs), and generic class-based views (GCBVs). This customization is useful for more experienced developers, but it is hard to understand for new developers. Many Django developers, including the person who wrote this article, like to use GCBVs when they can and switch to CBVs or FBVs when they have to. By the end of this book, you'll have tried all three, so you can decide for yourself which one you like best.

Next, we need to configure the URLs. Notice that there is no urls.py in the webpages folder. We need to create it. Once you do that, write in the following code:

```
from django.urls import path
from .views import homePageView

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

On the first line, we import the path from Django to link our URL; on the second line, we import the views from the same folder. By calling the views.py file .views, we are telling Django to look for a views.py file in the current folder and import the homePageView function from there.

Our URL file is made up of three parts:

a Python regular expression for the empty string " ",

a reference to the view called "homePageView," and

an optional named URL pattern called "home."

In other words, if the user asks for the homepage, represented by the empty string "," Django should use the view called homePageView.

Just one last thing now. Now we need to update the urls.py file in our django my_project folder. It's common for a Django project to have more than one app in our webpages, each app needs its own URL path.

```
script > helloworld > my_project > urls.py > ...
"""my_project URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/4.0/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""
from django.contrib import admin
from django.urls import path

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

```

All you need to do is edit the code like this:

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

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

Now let us test our Home Page. Restart your server with the following code and reload that url in your browser:

python manage.py runserver

Now, let us move on.

Using Git

In the last chapter, we set up Git, which is a version control system. Let's put it to use. The first step is to add Git to our repository or start it up. Make sure you have Control+c pressed to stop the local server, and then run the command git init.

```
git init
```

When you run this, git will take control of the script. You can check and track changes by typing the command git status.

It is not advisable to allow our virtual environment, .venv, to be controlled by git. It shouldn't be in Git source control because it often contains secret information like API keys and the like. To hack this, use Django to create a new file called .gitignore that tells Git what to ignore.

```
.venv/
```

.venv will no longer be there if you run git status again. Git has ‘ignored’ it.

We also need to track the packages that are installed in our virtual environment. The best way to do that is to put this data in a requirements.txt file. Type the following command line:

```
pip freeze > requirements.txt
```

This will create the requirements.txt file and output the data we need. We need this because besides installing Django, there are many other packages that Django relies on to run. When you install one Python package, you often have to install a few others that it depends on as well. A requirements.txt file is very important so that it can help us see all the packages.

Now, we want to ensure that we will not have to manually add anything. We will automate it so that it inputs whatever we install moving on. Use this code:

```
(.venv) > git add -A

(.venv) > git commit -m "initial commit"
```

You can now exit the virtual environment by running “deactivate”. Congratulations! In this chapter, we've talked about a lot of essential ideas. We made our first Django app and learned how projects and apps are set up in Django. We learned about views, URLs, and the Django web server built into the program. Move on to Chapter 3, where we'll use templates and class-based views to create and deploy a more complex Django app.

Chapter 3 - Django App With Pages

In this chapter, we'll create, test, and deploy a website app with a homepage and a services page. We haven't learned about databases, so you don’t have to worry much. However, we'll cover that in the next chapter. We'll learn about class-based views and templates, which are the building blocks for the more complex web applications we'll make later in the book.

In the previous chapter, the process of creating our blank app involves some initial setup where we need to create some new .py app files for the server. We will do the same here.

Setup

You have learned how to set up Django to create an application in chapter 2. Use the knowledge to

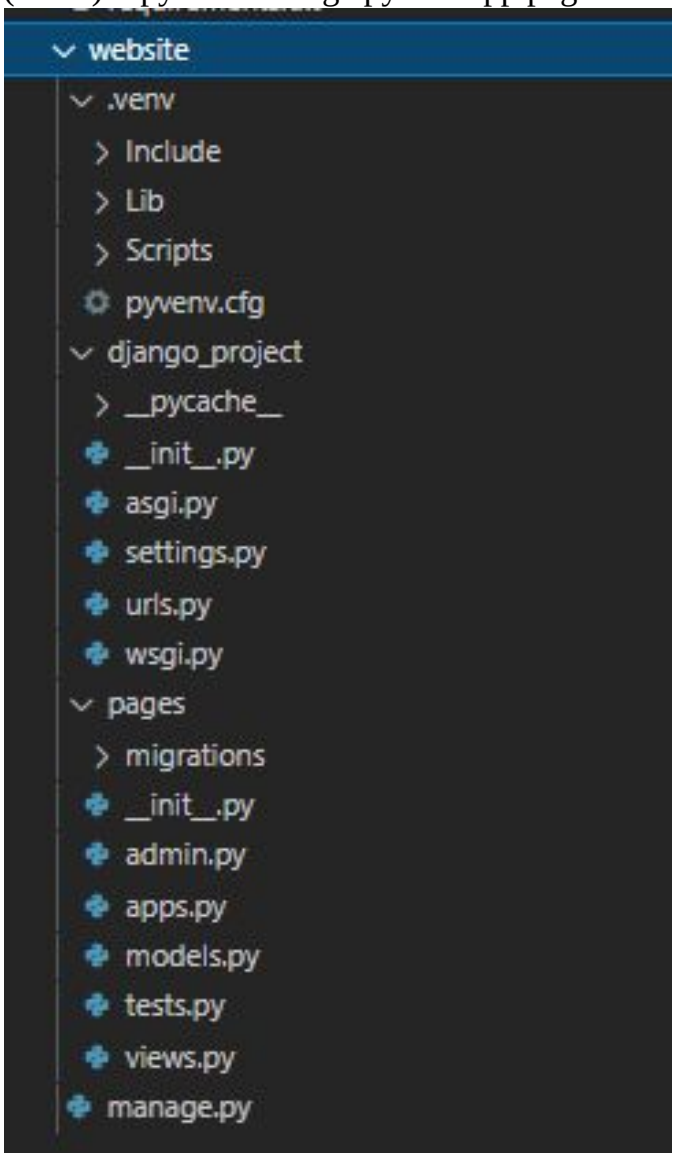
- make a new folder (project) called "website" for our code and go there.
- create a new virtual environment with the name .venv and turn it on.
- install Django.
- create a new Django project and call it django_project
- make a new app and call it Pages

Make sure, at the command line, that you are not working in a virtual environment that is already set up.

The steps outlined above are in easy steps, with each of the following lines a command you must run before the next:

```
> cd OneDrive\Desktop\script
> mkdir website
> cd website
> python -m venv .venv
> .venv\Scripts\Activate.ps1
```

(.venv) > python -m pip install django~=4.0.0
(.venv) > django-admin startproject django_project .
(.venv) > python manage.py startapp pages



Remember that we need to add the new project to the INSTALLED APPS setting in the settings.py file under the django_project folder. Now, open this file in your text editor and add the following line to the end:

"pages.apps.PagesConfig",

The migrate function moves the database and the runserver tool to start the local web server. Refer to chapter 2.

Adding Templates

A good web framework must make it easy to make HTML files. In Django, we use templates, which are separate HTML files that can be linked together and also have some basic logic built into them.

Remember that in the last chapter, the phrase "My First App" was hardcoded into a views.py file on our first site. That works technically, but if you want to build a big website, you will suffer a lot going that route. The best way is to link a view to a template because the information in each is kept separate.

In this chapter, we'll learn how to use templates to make our homepage and about page. In later chapters, you'll learn how to use templates to develop websites with hundreds, thousands, or even millions of pages that only need a small amount of code.

The first thing to learn is where to put templates in a Django project. By default, Django's template loader looks inside each app for templates that go with it. But the structure is a little confusing: each app needs a new templates directory, another directory with the same name as the app, and then the template file.

That implies that there will be a new folder in the pages folder called templates. Inside templates, we need another folder with the name of the app as pages, and then we will now save our template itself inside that folder as home.html.

Now, let us create a templates folder. Enter the pages folder in the code and type in the following:

```
mkdir templates
```

Next, we have to add the new template to the settings.py file inside the django project so that Django knows where our new templates directory is. Add the following to the TEMPLATES setting under "DIRS."

```
[BASE_DIR / "templates"],
```

So it looks like this:

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [BASE_DIR / "templates"],
        "APP_DIRS": True,
        "OPTIONS": {
```

Make a new file called home.html in the templates directory. You can do this in your text editor. In Visual Studio Code, click "File" and then "New File" in the top left corner of the screen. Make sure to give the file the correct name and save it in the right place.

For now, a simple headline will be in the home.html file.

```
<h1>Homepage
  Welcome to My Website
</h1>
```

That’s it. We are done creating our template. The next thing is for us to update the URL and view files.

Class and Views

You have seen how we deployed function-based views in the previous chapter. That was how Django was when it came. But doing that means developers will repeat the same patterns over and over again, writing a view that lists all objects in the model, and so on.

Classes are an essential part of Python, but we won't go into detail about them in this book. If you need an introduction or a refresher, I suggest reading the official Python documentation, which has an excellent tutorial on classes and how to use them.

We will use the built-in TemplateView to show our template in our view. Here is how to do that: Go to the pages folder and edit the views.py file

with this code:

```
from django.shortcuts import render

# Create your views here.
from django.views.generic import TemplateView
```

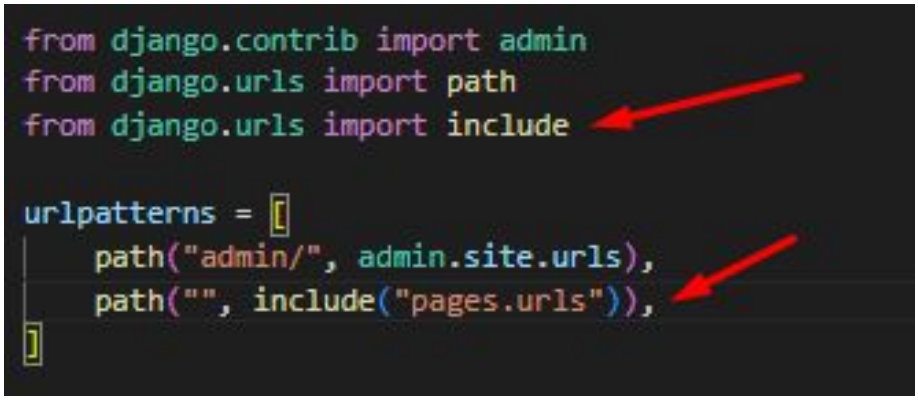
```
class HomePageView(TemplateView):
    template_name = "home.html"
```

Since HomePageView is now a Python class, we had to capitalize it. Unlike functions, classes should always start with a capital letter. The logic for showing our template is already built into the TemplateView. All we need to do is tell it the name of the template.

Our URLs

Last, we need to change our URLs. You may remember from Chapter 2 that we have to make changes in two places. First, we change the django project/urls.py file so that it points to our pages app. Then, we match views to URL routes within pages.

Let's start with the urls.py file in the django project folder.



Do you remember this code? On the second line, we add include to point the current URL to the Pages app.

Now, go ahead and create a new file in the pages folder and name it urls.py, and put the following code in it. This pattern is almost the same as what we did in Chapter 2, with one big difference: when using Class-Based Views, you always add as view() to the end of the view name.

```
from django.urls import path
from .views import HomePageView

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

And that is it! You can run the code now by typing the command:

python manage.py runserver

Then go to your browser.

We did it!

About Page

The process is the same. The only difference is in the content. We'll create a new template file, a new view, and a new url route. How will you do this? Start by creating a new template file called about.html within the templates folder and put a short HTML header in it.

```
<h1>About Me</h1>
```

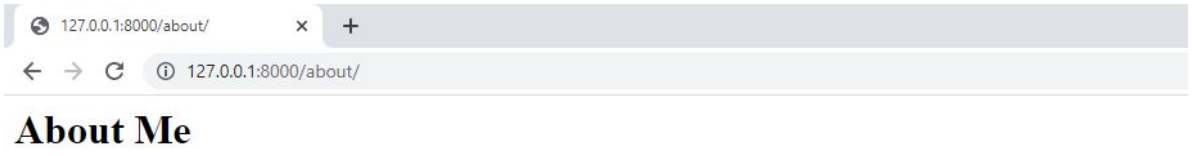
Now, like you did for the homepage, go to the views.py file in pages and create a view for this new page template you just built. Add the following code after the Home page view that is already there:

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

Lastly, you need to go to the urls and import the about page view name so that you can connect it to a URL. Use the code below:

path("about/", AboutPageView.as_view(), name="about"),

Go back to your browser and try the url http://127.0.0.1:8000/about



Extending Templates

The best thing about templates is how you can extend them. Most websites have headers or footers that you see on every page. How can you do that?

First, we make a file called base.html within the templates folder, and we will put in a header with links to the two pages we have. You can call this file anything, but many developers use base.html.

Django has a simple templating language that we can use to add links and simple logic to our templates. The official documentation shows the full list

of template tags that come with the program. Template tags are written like this: %something%, where "something" is the template tag itself. You can make your own template tags, but we won't cover that here.

We can use the built-in url template tag, which takes the URL pattern name as an argument, to add URL links to our project. Create the base.html file and add the following code:

```
<header>
   <a href="{% url 'home' %}">Home</a> |
   <a href="{% url 'about' %}">About</a>
</header>

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

Now let us go and edit the home.html and about.html files to show the new base.html code. The extends method in the Django templating language can be used for this.

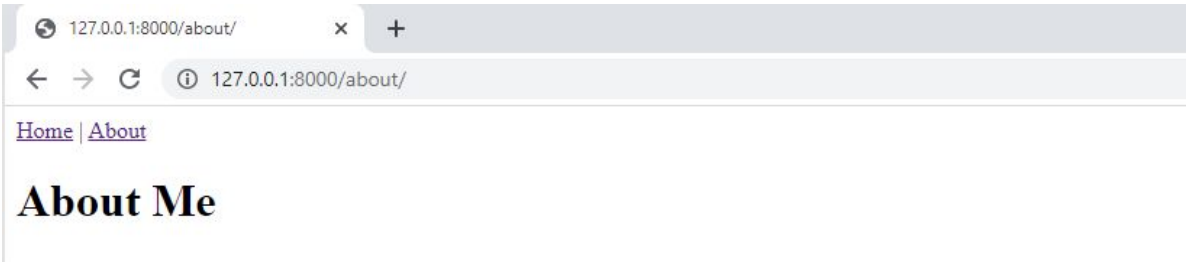
Open the home.html and change the code that was there to this:

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

{% block content %}
<h1>Welcome to my website!</h1>
{% endblock content %}
```

Open the about.html and change the code that was there to this:

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

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

Reload your server in the browser, and you will see the header showing on both pages like so:



Yay! We have created a two-page website. Let us talk about one practice that differentiates good programmers from great ones.

Testing

When a codebase changes, it's crucial to add automated tests and run them. Tests take a little time to write, but they pay off in the long run.

Unit testing and integration testing are the two main types of testing. Unit tests look at a single piece of functionality, while integration tests look at how several pieces work together. Unit tests only test a small amount of code, so they run faster and are easier to keep up to date. Integration tests take longer and are harder to keep up with because the problem comes from when they fail. Most developers spend most of their time writing unit tests and only a few integration tests.

The Python standard library has a built-in testing framework called unittest. It uses TestCase instances and a long list of assert methods to check for and report failures.

On top of Python's unittest, Django's testing framework adds several

Base class for TestCase. These include a test client for making fake Web browser requests, many Django-specific additional assertions, and four test case classes: SimpleTestCase, TestCase, TransactionTestCase, and LiveServerTestCase.

In general, you use SimpleTestCase when you don't need a database, while you use TestCase when you do want to test the database. LiveServerTestCase starts a live server thread that can be used for testing with browser-based tools like Selenium. TransactionTestCase is useful if you need to test database transactions directly.

One quick note before we move on: you may have noticed that the names of methods in unittest and django.test are written in camelCase instead of the more Pythonic snake case pattern. Because unittest is based on the jUnit testing framework from Java, which does use camelCase, camelCase naming came with unittest when it was added to Python.

If you look in our pages app, you'll see that Django has already given us a file called tests.py that we can use. Since our project hasn't got to do with a database, we'll import SimpleTestCase at the top of the file. For our first test, we'll make sure that both of our website's URLs, the homepage and the "about" page, return the standard HTTP status code of 200, which means that the request was successful.

```
from django.test import TestCase

# Create your tests here.
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 test, you must first stop the server with Ctrl + C and then type in the command python manage.py test to run the tests.

```
Found 2 test(s).
System check identified no issues (0 silenced).
..
----------------------------------------------------------------------
Ran 2 tests in 0.089s

OK
```

If you see an error like "AssertionError: 301 does not equal 200," you probably forgot to add the last slash to "/about" above. The web browser knows to automatically add a slash if it's not there, but that causes a 301 redirect, not a 200 success response.

How about we test the name of the urls of our pages? In our urls.py file in pages, we added "home" to the path for the homepage and "about" to the path for the about page. We can run a test on both pages with a useful Django function called reverse. Now, open the test.py file, and edit it. First, import reverse at the top of the code and add a new unit test for each below it. This is the latest updated code in the test.py file:

```
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)
```

```
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)
```

Now, rerun the test.

So far, we have tested where our URLs are and what they are called, but not our templates. Let's ensure that the right templates, home.html, and about.html, are used on each page and that they show the expected content we wrote inside the templates.

Let us use assertTemplateUsed and assertContains. Update the test.py code to become this:

```
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):
        response = self.client.get(reverse("home"))
        self.assertTemplateUsed(response, "home.html")

    def test_template_content(self):
        response = self.client.get(reverse("home"))
        self.assertContains(response, "<h1>Welcome to my website!</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):
        response = self.client.get(reverse("about"))
        self.assertTemplateUsed(response, "about.html")

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

If an experienced programmer looks at our test code, they may scoff at us because it repeats a lot. For example, we had to set an answer for each of the eight tests.

In Django coding, there is a rule called Don't Repeat Yourself (DRY). This rule makes code clean. However, unit tests work best when they are self-contained and very verbose. As a test suite grows, it might be better for performance to combine multiple assertions into fewer tests. However, that is an advanced and often subjective topic that is beyond the scope of this book.

In the future, especially when we start working with databases, we'll do a lot more testing. For now, it's essential to see how easy and important it is to add tests to our Django project whenever we add new features.

Now, let us use Git to track the changes. You can upload your code to GitHub if you have a repository. You can create one for your Django

projects. Also, remember to create a .gitignore file in your project folder and put .venv/ so that we will keep our virtual environment out of the checks. Then run the git add -A and the git commit -m "initial commit".

Now go to GitHub. If you don’t already have a GitHub account, it’s time you created one. You must now create a new repository and call it "pages," and make sure the "Private" radio button is selected. Then click the button that says "Create repository."

Scroll to the bottom of the next page until you see "...or push an existing repository from the command line." Copy the two commands there and paste them into your terminal.

It should look like the example below, but instead of MacVicquayns, your GitHub username should be there.

```
git remote add origin https://github.com/MacVicquayns/pages.git
git push -u origin main
```

Website Production

To deploy our new web project to the internet so that everyone can access it, we need to put our code on an external server and database. What we have done is local code. That only lives on our computer. We need production code that will be on a server outside of our computer that everyone can access.

The settings.py in our django_project folder is used to set up a new project for local development. Because it's easy to use, we have to change a number of settings when it's time to put the project into production.

Django comes with its own basic server, which can be used locally but not in a production setting. You can choose between Gunicorn and uWSGI. Gunicorn is the easiest to set up and works well enough for our projects, so we will use that.

We will use Heroku as our hosting service because it is free for small projects, is used by a lot of people, and is easy to set up.

Heroku

Search for Heroku on your search engine and open the official website. Create a free account with the registration form and wait for an email with a link to confirm your account. A link in the verification email takes you to the page where you can set up your password. Once you've set everything up, the site will take you to the dashboard.

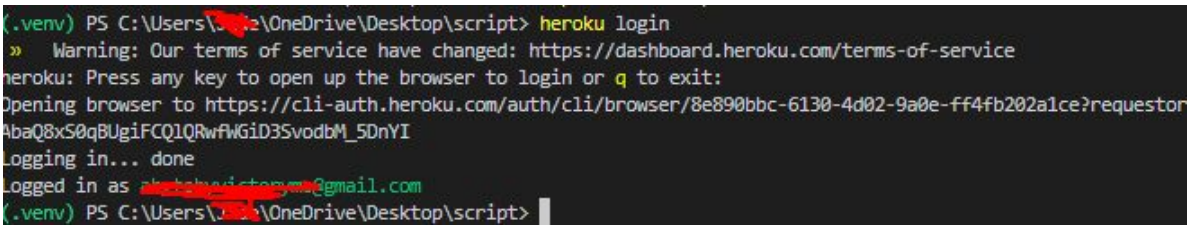
Now that you have signed up, you need to install Heroku's Command Line Interface (CLI) so that we can deploy from the command line. We currently work on our Pages project in a virtual environment, but we want Heroku to be available everywhere on our machine and not only in the virtual environment. So you can open a new command line terminal for this.

On Windows, go to the Heroku CLI page to learn how to install the 32-bit or 64-bit version correctly. For macOS, you can use Homebrew to install it. Homebrew is already on your Mac computer. Type this code in a new terminal tab, not in a virtual environment.

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

Once the installation is done, you can close the new command line tab and go back to the first tab with the pages virtual environment open.

Type "heroku login" and follow the instructions to use the email address and password you just set up for Heroku to log in.



Now, we are ready to deploy the app online.

Let’s Deploy

The first thing to do is to set up Gunicorn, which is a web server for our project that is ready for production. Remember that we've been using

Django's own lightweight server for local testing, but it's not good enough for a live website. Let us use Pip to install Gunicorn.

Type in the following code:

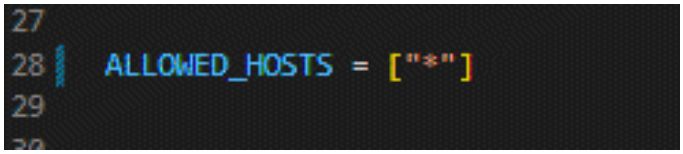
```
python -m pip install gunicorn==20.1.0
```

Step two is to make a file called "requirements.txt" that lists all the Python dependencies that our project needs. That is, all of the Python packages we have installed in our virtual environment right now. This is important in case a team member, or we ever want to start over with the repository. It also lets Heroku know that the project is written in Python, which makes the deployment steps easier.

To make this file, we will tell the pip freeze command to send its output to a new file called requirements.txt. Use the code below:

```
python -m pip freeze > requirements.txt
```

The third step is to look in the django project and add something to the settings.py file. Go to the ALLOWED HOSTS setting, which tells us which host/domain names our Django site can serve. This is a way to keep HTTP Host header attacks from happening. For now, we'll use the asterisk * as a wildcard so that all domains will work. We'll learn later in the book how to explicitly list the domains that should be allowed, which is a much safer way to do things.



Step four is to make a new Procfile in the same folder as manage.py (the base folder). Go to the folder where manage.py is, create a new file, and name it Procfile. The Procfile is unique to Heroku and tells you how to run the app in their bundle. In this case, inside the Profile, we're telling the web function to use the gunicorn server, the WSGI configuration file at

django_project.wsgi, and the --log-file flag to show us any logging messages. Type the following line inside the Profile.

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

The last step is to tell Heroku which version of Python to use. This will let you quickly know what version to use in the future. Since we are using Python 3.10, we need to make a runtime.txt file that is just for it. Using your text editor, create this new runtime.txt file in your text editor in the same folder as the Procfile and manage.py files.

Run python --version to find out what version of Python is being used and copy it and paste it into the new runtime.txt file. Make sure everything is in small letters.

Check the changes with git status, add the new files, and then commit the changes:

```
git status
git add -A
git commit -m "New updates for Heroku deployment"
```

The last step is to use Heroku to put the code into action. If you've ever set up a server on your own, you'll be surprised at how much easier it is to use a platform-as-a-service like Heroku.

Here's how we'll do things:

Heroku: make a new app

Disable the static file collection (we'll discuss this later).

The code was sent to Heroku.

start the Heroku server so the app can be used by people

visit the app's URL, which Heroku gives you.

The first step, making a new Heroku app, can be done from the command line with the heroku create command. Heroku will give our app a random name, like ⬢ intense-inlet-86193 in my case. You will have a different name.

The heroku create command also makes a remote for our app called "heroku." Type git remote -v to see this.

With this new remote, we can push and pull code from Heroku as long as the word "heroku" is in the command.

At this point, we only need one more set of Heroku setup, and that is to tell Heroku to start ignoring static files like CSS and JavaScript. Django will optimize these for us by default, which can cause problems. We'll talk about this in later chapters, so for now, just run the command below:

```
heroku config:set DISABLE_COLLECTSTATIC=1
```

Now, use the following line to push the code to Heroku:

```
git push heroku main
```

We're done! The final step is to make sure our app is up and running. If you type the command heroku open, your web browser will open a new tab with the URL of your app:

You don't have to log out of your Heroku app or leave it. It will keep running on its own at this free level, but you'll need to type "deactivate" to leave the local virtual environment and move on to the next chapter.

Congratulations on getting your second Django project up and running. This time, we used templates and class-based views, explored URLs in more depth, added basic tests, and used Heroku. Don't worry if the deployment process seems too much for you. Deployment is complex, even

with a tool like Heroku. The good news is that most projects have the same steps, so you can use a deployment checklist each time you start a new project.

In the next chapter, we'll start our first database-backed project, a Message Board website, and see where Django shines. We'll use templates and class-based views to build and deploy a more complex Django app.

Chapter 4 - Create Your First Database-Driven App And Use The Django Admin

In this chapter, we'll build a basic Message Board application where users can post and read short messages. This will be the first time we use a database. We'll look at Django's powerful built-in admin interface, which lets us change our data in a way that is easy to understand. After adding tests, we'll push our code to GitHub and put the app on Heroku.

Thanks to the powerful Object-Relational Mapper (ORM) in Django, there is built-in support for MySQL, PostgreSQL, Oracle, MariaDB, and SQLite as database backends. As developers, we can write the same Python code in a models.py file, and it will automatically be turned into the correct SQL for each database. Only the DATABASES section of our settings.py file that is inside the django project folder needs to be changed. This really is a great feature!

Django uses SQLite by default for local development because it is a file-based database and, therefore, much easier to use than the other database options, which require a separate server to run in addition to Django.

Initial Setup

At this point in the book, we've already set up a few Django projects, so we can quickly go through the basic commands to start a new one. Here's what we need to do:

Create a folder called message-board to store our code.

Make a new project called django_project and install Django in a virtual environment.

Make a new Django app called posts

update the settings file at django project/settings.py.

Type the following commands into a new command line console. Remember that you must run each line before typing the next:

> cd C:\Users\OneDrive\Desktop\script

> mkdir message-app

> cd message-app

> python -m venv .venv

> .venv\Scripts\Activate.ps1

(.venv) > python -m pip install django~=4.0.0

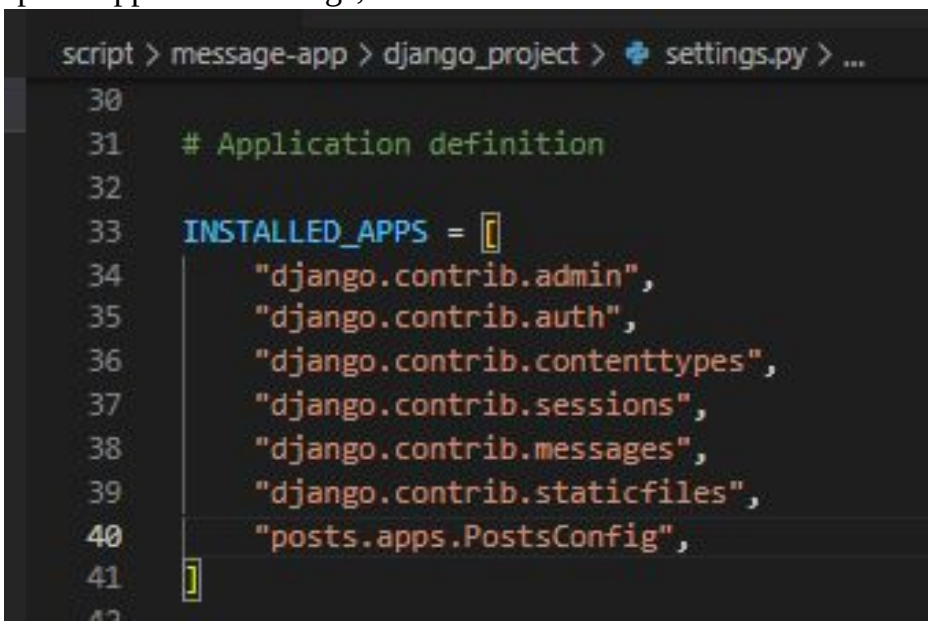
(.venv) > django-admin startproject django_project .

(.venv) > python manage.py startapp posts

Now, let’s add the new app, posts to the INSTALLED_APPS section of our settings.py file in the django_project folder. Do you remember the way to do that?

Add the following line to the section:

"posts.apps.PostsConfig",



Then, use the migrate command to get started with a database already configured for use with Django.

python manage.py migrate

You should see db.sqlite3 among the new files now representing the SQLite database.

When you first run either migrate or runserver, a db.sqlite3 file is generated, but the migrate command will update the database to reflect the current state of any database models that are part of the project and are included in INSTALLED APPS. That is to say, every time you change a model, you'll need to execute the migrate command to ensure the database is in sync with your changes. More to come on this.

Use the runserver to launch our local server and check whether it’s working.

```
python manage.py runserver
```

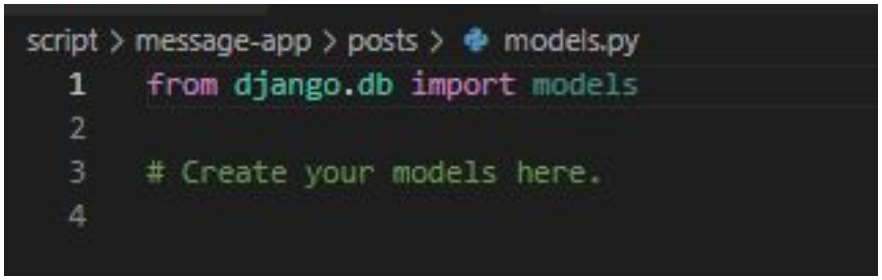
Now go to the local URL on your browser: http://127.0.0.1:8000/

If you don’t see the Django welcome page, there is something wrong with your script.

Let’s Create a Database Model

The first course of action is to build a database structure that can be used to save and display user-submitted content. This model can be easily converted into a database table with the help of Django's ORM. While many different database models may be required for a complex Django application, this simple message board program simply requires a single one.

Open the models.py file in the posts folder to view the Django-supplied default code.



In the first line there, as you can see, Django imports a module called models to allow us to create new database models that can "model" our data. We need a model to save the text of a message board post, and we can achieve so by adding the following lines:

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

Keep in mind that we just made a new database model called Post, which has a field text. The type of information stored in this TextField(). Model fields in Django may store a wide variety of data, including text, dates, numbers, emails, and more.

Activate the models

Our new model is complete; the next step is to put it into action. In the future, updating Django will involve a two-step process anytime a model is created or modified:

To begin, we use the makemigrations command to generate a migrations file. By using migration files, we can keep track of modifications made to the database models over time and debug issues as they arise.

Second, we use the migrate command, which runs the commands in our migrations file, to construct the database.

Ensure that the local server is stopped. You can stop it by typing Control + c on the command line. After that, run python manage.py makemigrations posts and python manage.py migrate.

Please keep in mind that the last name is optional after makemigrations. A migrations file will be generated for all accessible modifications in the Django project if you simply execute python manage.py makemigrations. That makes sense for a small project with a single app, like ours, but not for

the vast majority of Django projects, which typically involve multiple apps. So, if you updated the model across different apps, the resulting migrations file would reflect all of those revisions. Clearly, this is not the best scenario. The smaller and more concise a migrations file is, the simpler it is to debug and undo any mistakes. To this end, it is recommended that the name of an application be specified whenever the makemigrations command is run.

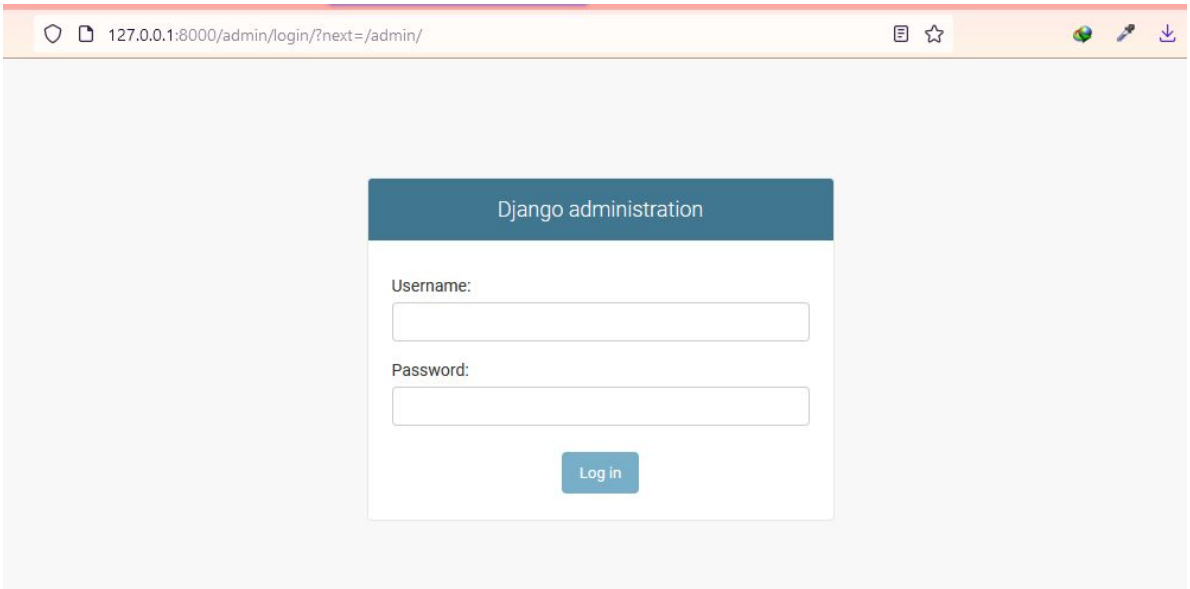
Django Admin

Django's strong admin interface, which allows users to visually interact with data, is a major selling point for the framework. This is partly due to the fact that Django's origins lie in its employment as a content management system for newspapers (Content Management System). The goal was to provide a place for writers to draft and revise articles outside the "code" environment. The in-built admin app has matured into a powerful, ready-made resource for handling any and all parts of a Django project.

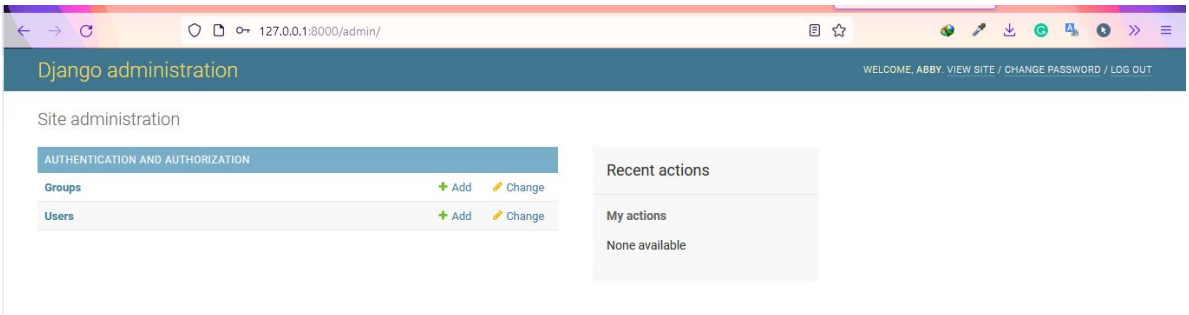
It is necessary to generate a superuser before accessing the Django admin. Type python manage.py createsuperuser into the command prompt and enter a username, email address, and password when prompted.

```
python manage.py createsuperuser

Username (leave blank to use 'jide'): Abby

Email address: abytobyvictoryme@gmail.com

Password:

Password (again):

Superuser created successfully.
```

The command line console will hide your password as you write it for security purposes. Run python manage.py runserver to restart the Django server, then navigate to http://127.0.0.1:8000/admin/ in a web browser. A screen prompting you to enter your admin login should appear.



Enter the new login details you just registered. The next screen you see is the Django administration dashboard:



You can adjust the LANGUAGE in the settings.py file. It is set to American English, en-us, by default. You can access the admin, forms, and other default messages in a language other than English.

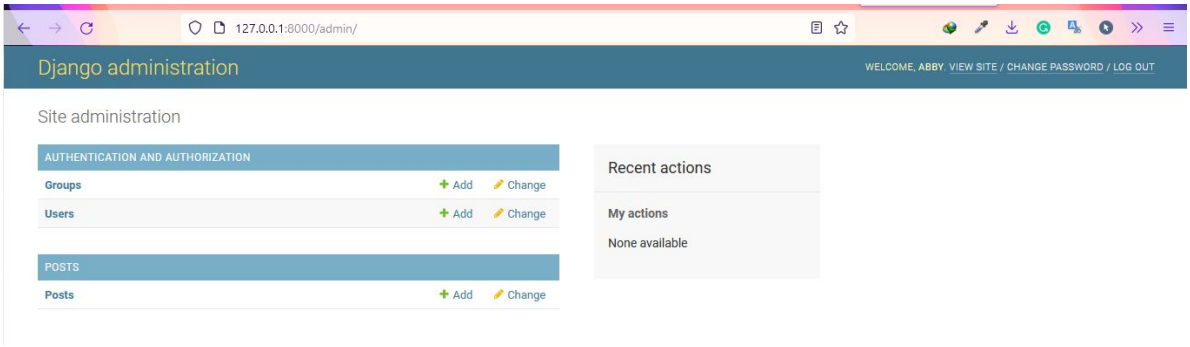
Our post app does not appear on any primary administration pages. Before this will show on the website, the admin.py file for an app needs to be updated in the same way that the INSTALLED_APPS settings needs to be modified in order for the app to be shown in the admin.

For the Post model to be visible, open admin.py in the posts folder in your preferred text editor and insert the following lines of code.

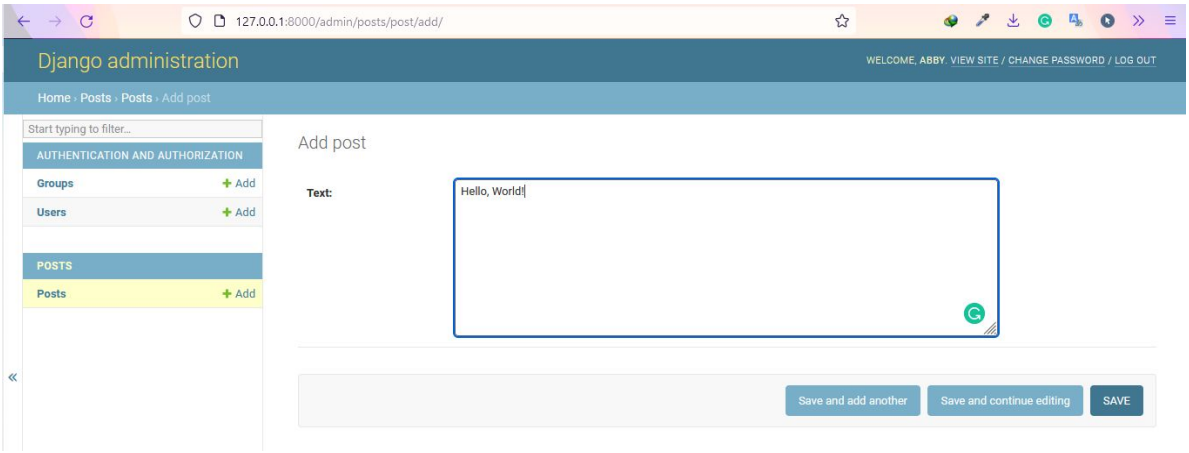
```
from .models import Post
```

```
admin.site.register(Post)
```

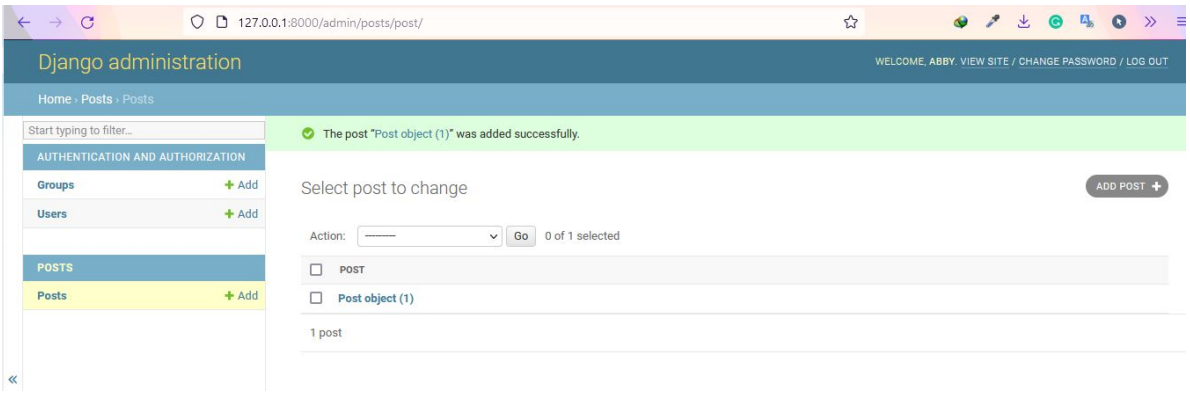
Now, run the server again and go to the page.



Let's add our first post to the message board to our database. Click the "+ Add" button next to "Posts" and type your own text in the "Text" field.



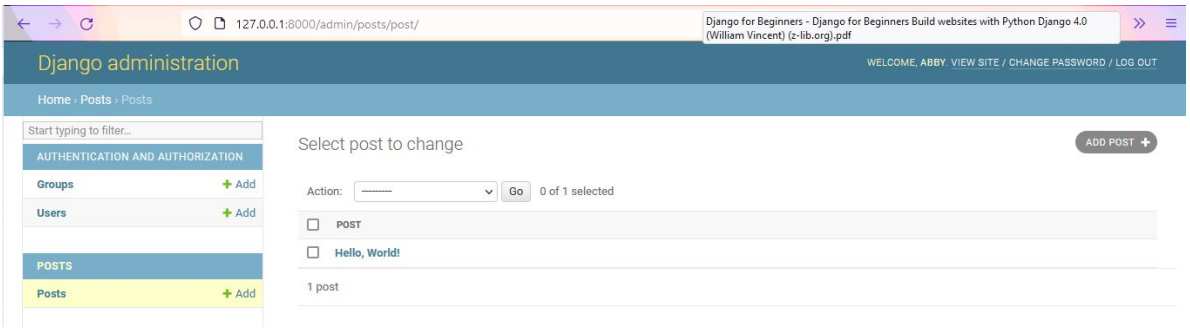
Then, click "Save." This will take you back to the main Post page. Yet, if you take a closer look, you'll notice that our new entry is titled "Post object (1)."



You can change that. Go to the posts folder and open the models.py file. From there, add a new function with the following code:

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

We told the code to give the post a title based on the first 50 characters of the post on the page. If you save this and refresh your admin page, you will see the change:



All models should have str() methods to make them easier to read.

Views/Templates/URLs

We need to connect our views, templates, and URLs so that the data in our database can be displayed on the front page. You should recognize this structure.

First, let's take in the view. Earlier in the book, we displayed a template file on our homepage using the built-in generic TemplateView. To that end, we

will detail our database model's components. Thankfully, this is very simple in web development, and Django provides the generic class-based ListView for this purpose.

Copy and paste the following Python code into the posts/views.py file:

```
from django.shortcuts import render

# Create your views here.

from django.views.generic import ListView
from .models import Post
```

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

The ListView and Post models are imported on the first and second lines. HomePageView is a subclass of ListView with the appropriate model and template declared.

With the completion of our view, we can go on to the next steps of developing our template and setting up our URLs. So, let's get started with the basic structure. First, use Django to make a folder named templates.

```
mkdir templates
```

Then we need to tell Django to use this new templates directory by editing the DIRS column in the Templates section in our settings.py file in the django project folder.

```
"DIRS": [BASE_DIR / "templates"],
```

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [BASE_DIR / "templates"],
        "APP_DIRS": True,
        "OPTIONS": {
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
                "django.contrib.messages.context_processors.messages",
            ]
```

Create a new file in the templates folder, home.html, using your preferred text editor. The template tag has a built-in looping capability, and ListView provides us with a context variable named <model>_list, which is the name of our model. We'll make a new variable called post and then use post.text to get at the field we want to show. This is the script for the home.html file:

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

Now, lastly, we set up our URLs. Go to the urls.py file inside the django_project folder. Go to the point where we added our posts app and put include in the second line like so:

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

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

Next, go to the posts folder and create the urls.py there too. Update that with the following code:

```
from django.urls import path
from .views import HomePageView

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

Use python manage.py runserver to restart the server and navigate to the local url in your browser. Check the home page of our new app.

We're almost finished, but before we call it a day, let's make a few more forum posts in the Django backend and make sure they show up appropriately on the front page.

Let’s Add New Posts

Please return to the Admin and make two more posts in order to update our forum. It will then display the prepared posts automatically on the homepage when you return to it. Awesome!

Assuming no errors have been encountered, we may now set up the directory and make a .gitignore file. Make a new .gitignore file in your text editor and add the following line:

.venv/

Then, after using git status once more to verify that the .venv directory is being ignored, you can use git add -A to add the desired files and directories and a first commit message.

```
(.venv) PS C:\Users\Jide\OneDrive\Desktop\script\message-app> git init
Initialized empty Git repository in C:/Users/Jide/OneDrive/Desktop/script/message-app/.git/
(.venv) PS C:\Users\Jide\OneDrive\Desktop\script\message-app> git add -A
(.venv) PS C:\Users\Jide\OneDrive\Desktop\script\message-app> git commit -m "initial commit"
[main (root-commit) dc3ede6] initial commit
 30 files changed, 271 insertions(+)
```

Tests

Previously, we used SimpleTestCase because we were testing fixed pages. Since our project now incorporates a database, we must use TestCase to generate a replica of the production database for testing purposes. We may create a new test database, populate it with sample data, and run tests against it instead of our live database, which is both safer and more efficient.

To generate test data, we will invoke the hook setUpTestData(). This feature, introduced in Django 1.8, makes it possible to produce test data only once per test case rather than once each test, making it much faster than using the setUp() hook from Python's unittest. However, setUp() is still commonly used in Django projects. Any such tests should be migrated to setUpTestData, as this is a proven method of increasing the overall speed of a test suite.

Let's get our data in order and then double-check that it was saved correctly in the database, as there is only one field in our Post model: text. To make sure Django runs them, all test methods should begin with test*. The code will look like this:

```
from django.test import TestCase

# Create your tests here.

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!")
```

TestCase and Post are imported first. PostTests extends TestCase and uses setUpTestData to create initial data. In this case, cls.post stores a single item that may be referred to as self.post in the following tests. Our first test, test model content, uses assertEqual to verify text field content.

Go to the command line and run this:

python manage.py test

(.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 0.002s

OK

Destroying test database for alias 'default'...

The test shows no errors! Still, the output ran only one test when we have two functions. Note that we set the test to only check functions that start with the name test*!

Now, let’s check our URLs, views, and templates as we did in chapter 3. We will also check

- URL for / and a 200 HTTP status code.
- URL for “home”.
- The home page shows “home.html” content correctly

Since only one webpage is involved in this project, all of these tests may be incorporated into the already PostTests class. In the header, select "import reverse," then add the tests as seen below.

```
from django.test import TestCase
from django.urls import reverse

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

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

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

With this, run the test again:

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 0.131s

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

```
from django.test import TestCase
from django.urls import reverse
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):
        response = self.client.get(reverse("home"))
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, "home.html")
        self.assertContains(response, "This is a test!")
```

We want our test suite to cover as much of the code as feasible while still being straightforward to reason about (both the error messages and the testing code itself). This revision is much simpler to read and comprehend, in my opinion.

Now that we've finished making changes to the code for testing, we can commit them to git.

(.venv) > git add -A

(.venv) > git commit -m "added tests"

[main 89ba70d] added tests

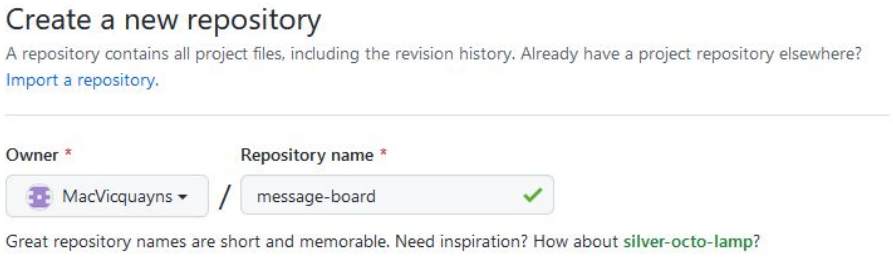
2 files changed, 20 insertions(+), 1 deletion(-)

create mode 100644 posts/__pycache__/tests.cpython-310.pyc

Storing to GitHub

We should use GitHub to host our source code. Message-board is the name of the repository you will be creating, and if you haven't already done so, log into GitHub and sign up for an account. For more discreet communication, choose the "Private" option.

The option to "or push an existing repository from the command line" is at the bottom of the following page. If you replace my username with your own GitHub username, the two commands there should look like the next and may be copied and pasted into your terminal:



git remote add origin https://github.com/MacVicquayns/message-board.git

git branch -M main

git push -u origin main

Setup Heroku

By now, you should have a Heroku account. The following is our deployment checklist:

- install Gunicorn
- setup requirements.txt
- edit the ALLOWED_HOSTS in settings.py
- create Procfile
- create runtime.txt

Use Pip to install Gunicorn.

python -m pip install gunicorn==20.1.0

In the past, we would simply set ALLOWED HOSTS to * to accept all hosts, but this proved to be a flawed and potentially harmful shortcut. Our level of specificity may and should be increased. Django can be used on either localhost:8000 or 127.0.0.1:8000. Having used Heroku before, we know that all Heroku sites will have the.herokuapp.com extension. All three hosts may be included in the ALLOWED HOSTS setting. Open your settings.py in the django_project folder and update the ALLOWED_HOSTS list with the following:

".herokuapp.com", "localhost", "127.0.0.1"

Now, create your Procfile and put this code in it:

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

Lastly, create a runtime.txt file in the base folder like Procfile. And populate with this line:

python 3.10.1

Now, commit the new changes to git.

```
> git status
> git add -A
> git commit -m "New updates for Heroku deployment"
> git push -u origin main
```

Deploy to Heroku

First, log in to your Heroku account with the heroku login command. Then use the heroku create to create a new server.

Type in the following to tell Heroku to ignore static pages. This is skipped when you are creating a blog app.

```
heroku config:set DISABLE_COLLECTSTATIC=1
```

After that, we push the code to Heroku.

```
git push heroku main
```

Then we scale it.

```
heroku ps:scale web=1
```

From the command line, type heroku open to open the new project's URL in a new browser window. Closing the present virtual environment is as simple as typing "deactivate" at the prompt.

That’s it! We have built a complete forum message board app. Well done. In the next section, we will create a blog app.

Chapter 5 – Blog App

This chapter will focus on developing a Blog application where users may add, modify, and remove posts.

Each blog post will have its own detail page in addition to being shown on the homepage. Also covered will be the basics of styling using CSS and how Django handles static files.

Initial Set Up

The first six steps we take in our development course have not changed. Set up the new Django project in the following steps:

- create a new base folder and call it blog
- start a new virtual and install Django
- start a new Django project and call it django_project
- start a new app and call it blog
- migrate the code to set up the database
- edit the settings.py file with the correct details.

Let’s get started.

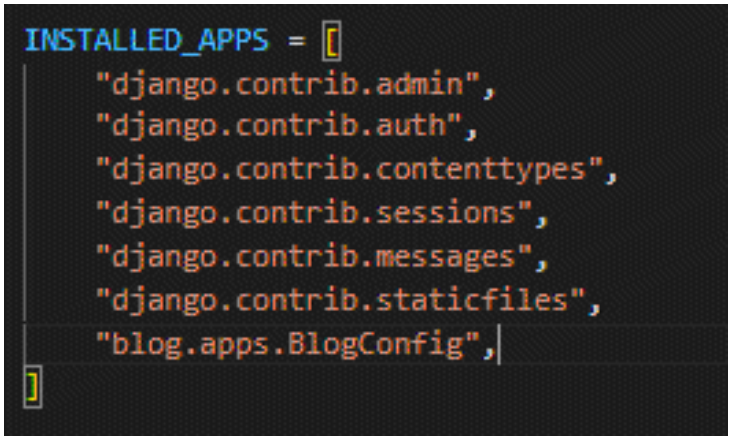
This is the sequence for Windows:

```
> cd onedrive\desktop\code
> mkdir blog
> cd blog
> python -m venv .venv
> .venv\Scripts\Activate.ps1
(.venv) > python -m pip install django~=4.0.0
(.venv) > django-admin startproject django_project .
(.venv) > python manage.py startapp blog
(.venv) > python manage.py migrate
```

This is for MacOs:

```
% cd ~/desktop/code
% mkdir blog
% cd blog
% python3 -m venv .venv
% source .venv/bin/activate
(.venv) % python3 -m pip install django~=4.0.0
(.venv) % django-admin startproject django_project .
(.venv) % python3 manage.py startapp blog
(.venv) % python3 manage.py migrate
```

Now go to the settings.py file and update the INSTALLED_APPS section:



Now, run the server and check the local url.

Initial setup complete! Well done!

Database Models

What are the standard features of a blog platform? Say each post contains a heading, author name, and article. The following code can be pasted into the models.py file in the blog folder to create a database model:

```
from django.db import models

# Create your models here.

from django.urls import reverse
```

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

    body = models.TextField()

    def __str__(self):
        return self.title

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

Once the new database model is complete, a migration record can be made, and an update may be made to the database.

Press Control+c to terminate the server.

You can finish this two-stage procedure by following the instructions below.

python manage.py makemigrations blog

python manage.py migrate

With these lines, we have created our database.

Admin Access

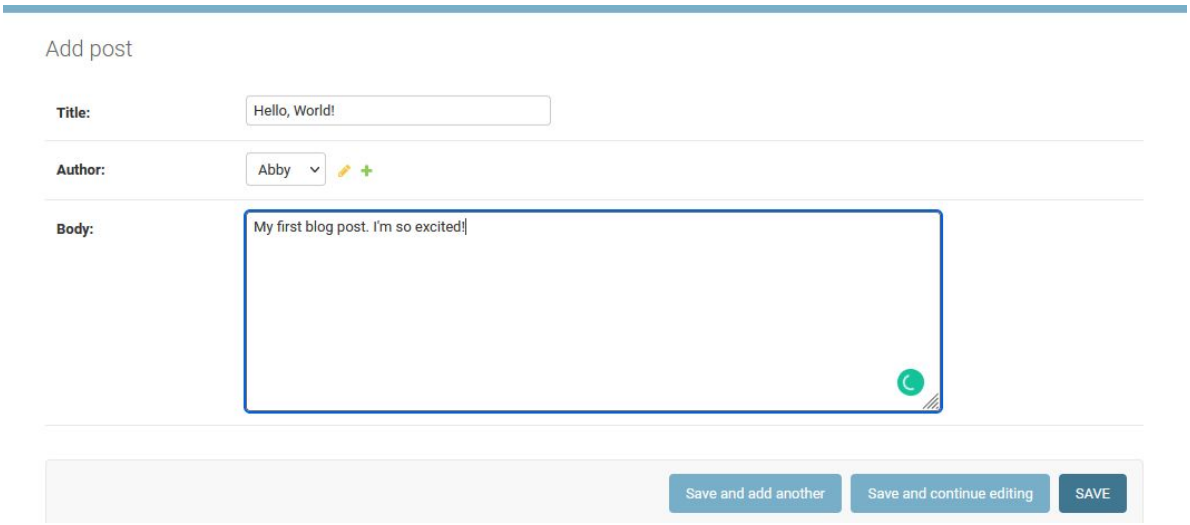
How will we access our data? We need to create Django's backend admin. Type the following command and then follow the prompts to create a superuser account with a unique email address and password. For security reasons, your password will not display as you type it.

python manage.py createsuperuser

Now, we update the admin.py file.

```
from django.contrib import admin

# Register your models here.

from .models import Post

admin.site.register(Post)
```

Let's add on a couple more blog posts. To add a new post, select the + Add button that appears next to Posts. All model fields are mandatory by default. Therefore be careful to give each post an "author" tag.



In order to display the data on our web application, we must now develop the views, URLs, and templates required to interact with the database.

URLs

To achieve this, we will first configure our urls.py file in the django_project folder, as we have done in previous chapters, and then our app-level blog folder's urls.py file.

Make a new file in the blog app named urls.py and paste the following into it using your text editor.

```
from django.urls import path
from .views import BlogListView

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

We imported the views we will do later. We give it a name, home so that we can use it in our views later on, and the empty string ("") instructs Python to match all values.

Although giving each URL a name isn't required, it's a good idea to help keep track of them as your list of URLs expands.

We also need to edit the urls.py file in the django_project folder so that it will send all blog app requests there.

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

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

Views

We will be using class-based views. Just a few lines of code in our views file, and we'll be able to see the results of our Post model in a ListView.

```
from django.shortcuts import render

# Create your views here.
from django.views.generic import ListView
from .models import Post
```

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

Templates

Now that we have finished with our URLs and views, we can go on to the next piece of the jigsaw, which is the templates. Previously in Chapter 4, we

learned that we can keep our code tidy by adopting from other templates. Therefore, we'll begin with a base.html file and an inherited home.html file.

Next, we'll add templates for making and revising blog articles, and those may all derive from base.html.

We should begin by making a folder to store our new template files. So, stop your server and type in the code:

mkdir templates

Make two new template files in the templates folder. Call them base.html and home.html.

The next step is to edit the settings.py file to direct Django to the appropriate folder to find our templates.

Add this line to the TEMPLATES section:

"DIRS": [BASE_DIR / "templates"],

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [BASE_DIR / "templates"],
        "APP_DIRS": True,
        "OPTIONS": {
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
                "django.contrib.messages.context_processors.messages",
            ],
        },
```

In the base.html file, put the following:

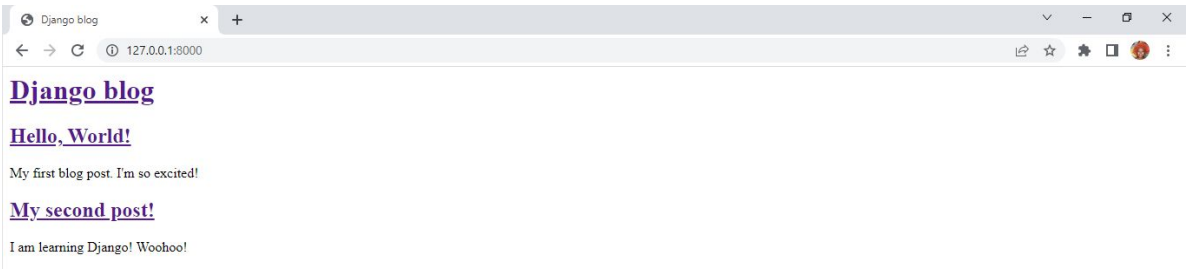
```
<!-- templates/base.html -->
<html>

<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>
```

Put this in the home.html:

```
<!-- templates/home.html -->
{% extends "base.html" %}
{% block content %}
{% for post in post_list %}
<div class="post-entry">
    <h2><a href="">{{ post.title }}</a></h2>
    <p>{{ post.body }}</p>
</div>
{% endfor %}
{% endblock content %}
```

If you run python manage.py runserver again and then reload the homepage, we will notice that the Django server is up and running.



Now, that is our first website. But it looks ugly! Let’s fix that.

Add some Style!

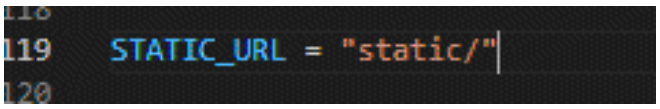
We need to add some CSS to our project to enhance the styling. A fundamental component of any contemporary web application is CSS, JavaScript, and pictures, which are referred to as "static files" in the Django ecosystem. Although Django offers enormous flexibility in terms of how these files are used, this can be very confusing for beginners.

Django will, by default, search each app for a subfolder called static. Or a folder with the name static in the blog folder. If you remember, this is also how the templates folder was created.

Stop the local server, then use the following line to create a static folder in the manage.py file's location.

mkdir static

We must instruct Django to look in this new folder when loading static files. There is already one line of configuration in the settings.py file, which you can find at the bottom:



Now, below that, we add the following line:

STATICFILES_DIRS = [BASE_DIR / "static"]

We instruct Django to look for static files in the newly formed static subfolder.

Use this line to create a CSS subfolder:

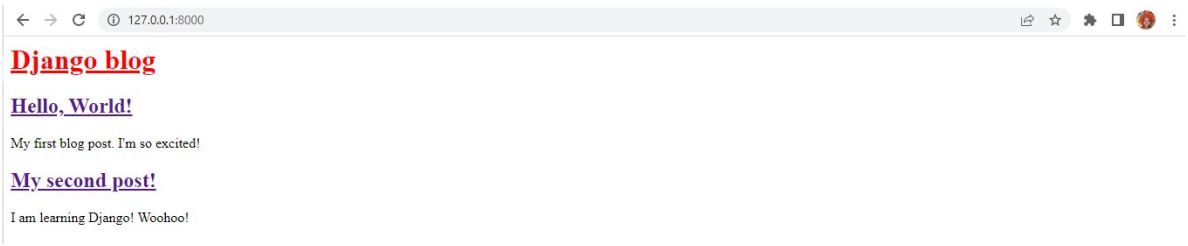
mkdir static/css

Use your text editor to create a new file within this folder called base.css inside the new CSS subfolder. Then fill it with this code to create a page title and color it red!

Almost there! Add % load static % to the top of base.html to include the static files in the templates. We only need to include it once because all of our other templates inherit from base.html. Insert a new line after closing the <head> tag to include a direct link to the base.css file we just created.

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

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

  <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>
```

Start up the server again and check the URL.



We can also customize other things like font size, type, etc., by tweaking the css file.

Individual Blog Pages

Individual blog posts can now have their stated features implemented. A new view, URL, and template will have to be developed.

One must first take in the view. To make things easier, we can utilize the DetailView, which is built on a generic class. Add DetailView to the import at the top of the script and generate a new view named BlogDetailView.

```
from django.shortcuts import render

# Create your views here.
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"
```

Let's say we want to create a new URL path for our view. Use the code seen below in the urls.py in the blog folder:

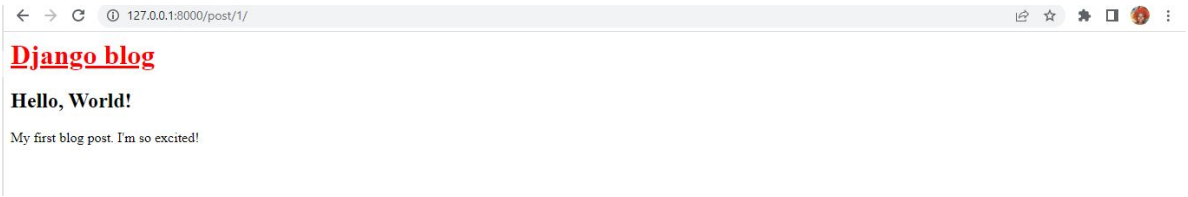
```
from django.urls import path
from .views import BlogListView, BlogDetailView

urlpatterns = [
    path("post/<int:pk>/", BlogDetailView.as_view(), name="post_detail"),
    path("", BlogListView.as_view(), name="home"),
]
```

For consistency, we've decided to prefix all blog post URLs with post/. The next thing to consider is the post entry's primary key, which we'll express as an integer, int:pk>. I know what you're thinking: "What is the main factor?" Our database models already have an auto-incrementing primary key86 because Django included it by default. As a result, while we only stated the fields title, author, and body on our Post model, Django automatically added an additional field named id, which serves as our primary key. Either an id or a pk will work to get in.

For our first "Hello, World!" message, we'll use a pk of 1. It's 2 for the second. The URL structure of our initial post, which will lead us to its particular entry page, will look like this: post/1/.

If you recall, the get absolute url method on our Post model accepts a pk argument in this case since the URL specifies it. Primarily, new users often struggle to grasp the relationship between primary keys and the get absolute url method. If you are still confused, it may help to read the previous two paragraphs again. You'll get used to it after some repetition.

After running python manage.py runserver, our first blog post will have its own URL of http://127.0.0.1:8000/post/1/.



To view the second entry, please visit http://127.0.0.1:8000/post/2/.

The link on the homepage should be updated so that we can easily navigate to certain blog entries. Replace the current empty link with a href=" % url 'post detail' post.pk % "> in home.html.

```
{% extends "base.html" %}
{% block content %}
{% for post in post_list %}
<div class="post-entry">
   <h2><a href="{% url 'post_detail' post.pk %}">{{ post.title }}</a></h2>
   <p>{{ post.body }}</p>
</div>
{% endfor %}
{% endblock content %}
```

Check and click the post from the home page.

Testing

New features have been added to our Blog project that we hadn't seen or tried before this section. We now have a user, various views (a list view of all blog posts and a detail view for each article), and a Post model with numerous fields. There is a lot to try out!

To start, we can prepare our test data and validate the Post model. So, this is how it may look in a nutshell:

```
from django.test import TestCase

# Create your tests here.
from django.contrib.auth import get_user_model
from django.urls import reverse
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):
        response = self.client.get("/")
        self.assertEqual(response.status_code, 200)

    def test_url_exists_at_correct_location_detailview(self):
        response = self.client.get("/post/1/")
        self.assertEqual(response.status_code, 200)

    def test_post_listview(self):
        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 test whether the requested URL exists in the correct folder for both views. Then, we ensure the home.html template is loaded, that the named URL is being utilized, that the right content is being returned, and that a successful 200 status code is being returned by creating the test post listview. To get a detail view of our test post, we must include the pk in response to the test post - detailview method. We keep using the same template but expand our tests to cover more edge cases. Since we haven't written two articles, we don't want a response at /post/100000/, for example. We also prefer to avoid an HTTP status code of 404. Incorrect examples of tests that should fail should be sprinkled in from time to time to ensure that your tests aren't all passing by accident.

Run the new tests to make sure everything is working as it should.

Git

Now, let us do our first Git commit. First, initialize our folder, create the .gitignore and review all the content we've added by checking the git status.

(.venv) > git status

(.venv) > git add -A

(.venv) > git commit -m "initial commit"

We have successfully created a working blog application from scratch. Django's admin panel allows us to quickly generate, modify, and remove content. For the first time, we were able to create a detailed view of each blog post separately by employing DetailView.

Chapter 6 – Django Web Forms

In this chapter, we'll continue developing the Blog application we started in Chapter 5 by adding the necessary forms for users to add, modify, or remove entries from their blogs. To accept user input raises security problems, making HTML forms one of the more complex and error-prone components of online development. All submitted forms must be rendered correctly, validated, and stored in the database.

Django's powerful in-built Forms abstract away much of the difficulties, making it unnecessary to write this code from scratch. Displaying, making changes to, or removing a form are some of the many commonplace actions that Django's built-in generic editing views are catered to.

CreateView

The first step is to provide a link to a website where new blog entries may be entered into our primary template. It will look like this: .

Your revised script should now look like this:

```
{% 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>
      <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>
      {% block content %}
      {% endblock content %}
    </div>
</body>

</html>
```

With this code, we have added the feature to post new content. But now, we need to add a new URL for the post_new feature. We need to import BlogCreateView in the urls.py file and add a URL path for post/new/.

```
from django.urls import path
from .views import BlogListView, BlogDetailView, BlogCreateView

urlpatterns = [
    path("post/new/", BlogCreateView.as_view(), name="post_new"),
    path("post/<int:pk>/", BlogDetailView.as_view(), name="post_detail"),
    path("", BlogListView.as_view(), name="home"),
]
```

We have seen this URL, views, and template pattern before. To build our view, we'll import the general class CreateView at the top and then subclass it to make a new view called BlogCreateView.

Now in the views.py file, update the code to be the following:

```
from django.views.generic import ListView, DetailView
from django.views.generic.edit import CreateView
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"]
```

The BlogCreateView class is where we define the Post database model, name our template post new.html, and establish the visibility of the title, author, and body fields in the underlying Post database table.

The final action is to make a template in the text editor and name it post_new.html. Then, add the following code to your file:

```
{% extends "base.html" %}
{% block content %}
<h1>New post</h1>
<form action="" method="post">{% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Save">
</form>
{% endblock content %}
```

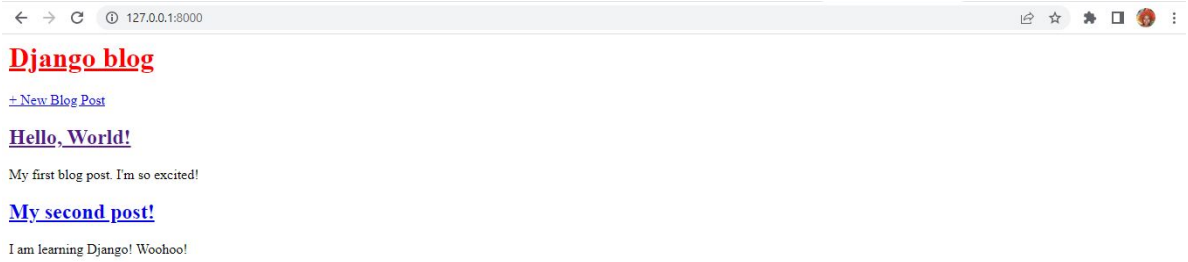
Let’s break it down:

- In the first line, we must inherit features from our base template.

- We are using an HTML form, so the <form> tags with the POST method are essential because we are sending. If it was to receive, like a search box, for example, instead of POST, we would use GET.
- Add a {% csrf_token %} from Django provides to protect our form from bots.
- We use {{ form.as_p }} to render the specified fields within paragraph <p> tags.

Lastly, set the value "Save" for a submit type input.

Launch the server with python manage.py runserver and navigate to the homepage to check at http://127.0.0.1:8000/.



Click the "+ New Blog Post" option to add a new blog post. If you click it, you'll be taken to a new page at http://127.0.0.1:8000/post/new/.

Try your hand at writing a new blog entry and publishing it by selecting "Save" from the file menu.

When it's done, it'll take you to a post-specific detail page at http://127.0.0.1:8000/post/3/.

Let Anyone Edit The Blog

Developing an edit form for blog entries should follow a similar pattern. To generate the necessary template, url, and view, we'll again leverage a built-in Django class-based generic view, UpdateView.

To begin, on each blog page there should be a link to post detail.html where the post can be edited. The following is the update:

```
{% extends "base.html" %}
{% block content %}
<div class="post-entry">
   <h2>{{ post.title }}</h2>
   <p>{{ post.body }}</p>
</div>
<a href="{% url 'post_edit' post.pk %}">+ Edit Blog Post</a>
{% endblock content %}
```

We use <a href>... and {% url ... %} tag to add the link. Within the tags, we specified the name of the new url, which we will call post_edit, and we also passed the needed argument, which is the primary key of the post.pk.

Now, let us create a template file for the new edit page. Call it post_edit.html and add the following code:

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

For the view. Open the views.py file and import UpdateView on the second-from-the-top line and then subclass it in the new view BlogUpdateView. Here is the updated code:

```
from django.views.generic import ListView, DetailView
from django.views.generic.edit import CreateView, UpdateView
from .models import Post
```

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

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

```
class BlogCreateView(CreateView):
    model = Post
    template_name = "post_new.html"
    fields = ["title", "author", "body"]
```

```
class BlogUpdateView(UpdateView):
    model = Post
    template_name = "post_edit.html"
    fields = ["title", "body"]
```

The final action is to modify the file urls.py in the way described below. We recommend placing the BlogUpdateView and the new route at the very top of the old urlpatterns.

```
from django.urls import path
from .views import (
    BlogListView,
    BlogDetailView,
    BlogCreateView,
    BlogUpdateView,
```

```
)

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

Now, if you click on a blog post, the Edit button will show like this:

If you click “+ Edit Blog Post,” it will redirect you to /post/3/edit/. You can edit anything.

When we modify and click the "Update" button, we're taken to the post's detail page, where we can see the update. This is due to our get_absolute_url configuration. If you go to the homepage now, you'll see the updated information alongside the rest of the posts.

Let Users Delete Posts

As with the post-update form, the post-deletion form is created similarly.

To build the required view, url, and template, we'll employ another generic class-based view, DeleteView.

To get started, go to post_detail.html to include a delete button on the page. Use the following code:

```
{% extends "base.html" %}
{% block content %}
<div class="post-entry">
  <h2>{{ post.title }}</h2>
  <p>{{ post.body }}</p>
</div>
<p><a href="{% url 'post_edit' post.pk %}">+ Edit Blog Post</a></p>
<p><a href="{% url 'post_delete' post.pk %}">+ Delete Blog Post</a></p>
{% endblock content %}
```

Make a new template for our post delete page. A file with the following contents will be created named 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 %}
```

In this case, the title of our blog post is being shown via the post.title variable. Since object.title is also a feature of DetailView, we could use that instead.

Create a new view that extends DeleteView, then update the views.py file to import DeleteView and reverse lazy at the beginning.

```
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 Post
```

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

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

```
class BlogCreateView(CreateView):
```

```
    model = Post
    template_name = "post_new.html"
    fields = ["title", "author", "body"]
```

```
class BlogUpdateView(UpdateView):
    model = Post
    template_name = "post_edit.html"
    fields = ["title", "body"]
```

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

DeleteView takes three parameters: a Post model, a post delete.html template, and a success url property. Exactly what effect does this have? After deleting a blog article, we want to send the user to the homepage.

In addition to CreateView, UpdateView also has redirects, but we did not need to supply a success url because of this. Because if get absolute url() is present on the model object, Django will utilize it by default. In addition, this attribute is only shown to those that take the time to study and memorize the documentation, namely the sections on model forms and success url.

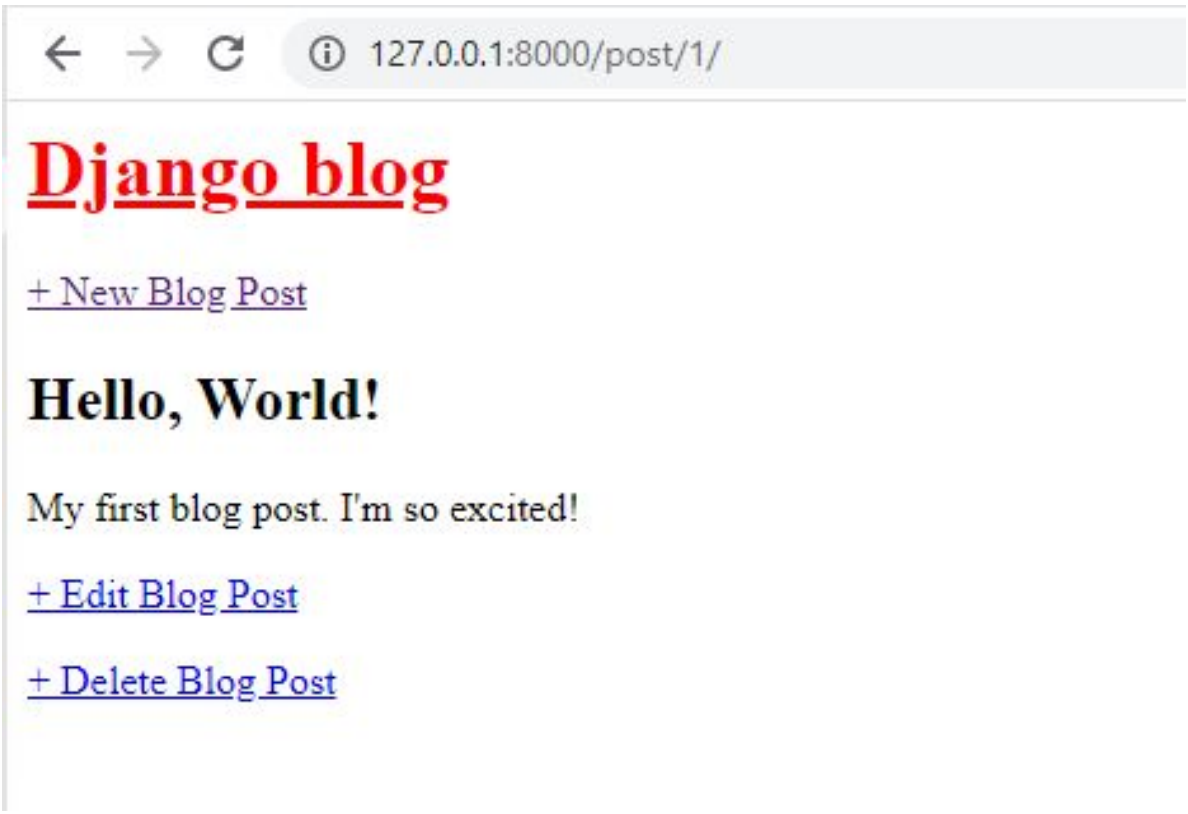
Or the likelihood of an error occurring and subsequent backtracking to resolve this Django-specific behavior is increased.

In this case, we use reverse_lazy rather than just reverse to delay the URL redirect's execution until after our view has completed removing the blog article.

Final step: Make a URL by importing our view BlogDeleteView and appending a new pattern:

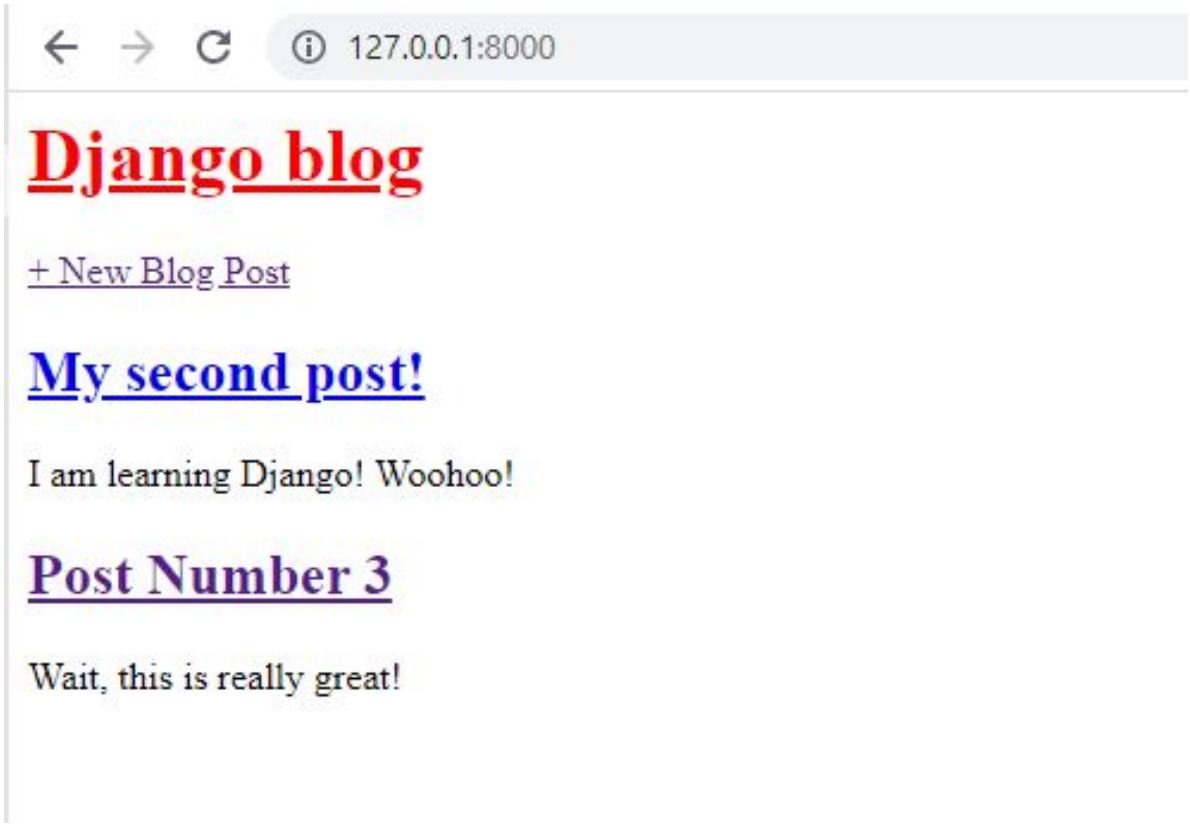
```
from django.urls import path
from .views import (
    BlogListView,
    BlogDetailView,
    BlogCreateView,
    BlogUpdateView,
    BlogDeleteView,
)

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

Once you've restarted the server with the python manage.py runserver command, you can refresh any post page to reveal our "Delete Blog Post" option.



The new page will show if you click it, asking you to confirm.

Click confirm, and the post is gone!



Testing Program

We have added so many features. Let us test everything to see that they will continue to work as expected. We have new views for creating, updating, and deleting posts. We will use three new tests:

- def test_post_createview
- def test_post_updateview
- def test_post_deleteview

The updated script in your tests.py file will be as follows.

```
from django.test import TestCase

# Create your tests here.
from django.contrib.auth import get_user_model
from django.urls import reverse
from .models import Post
```

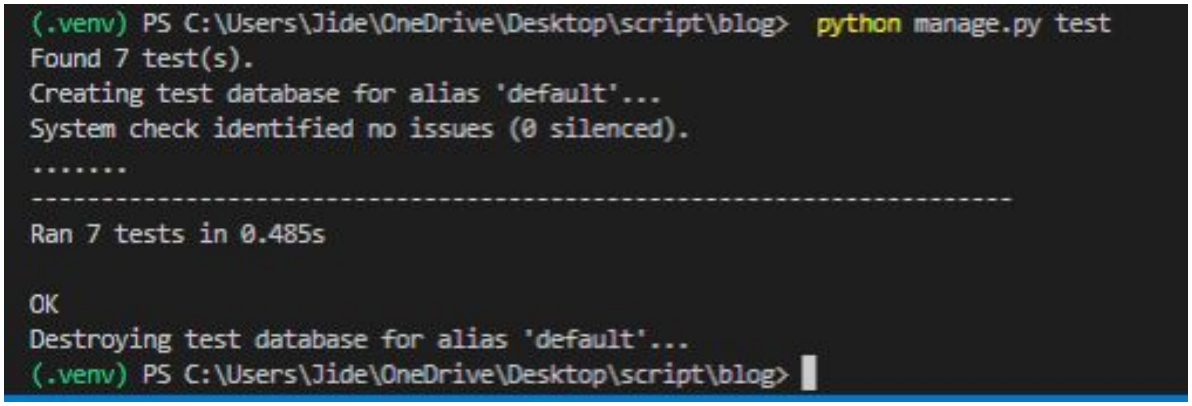
```
class BlogTests(TestCase):
    @classmethod
    def setUpTestData(cls):
        cls.user = get_user_model().objects.create_user(
            username="testuser", email="test@email.com", password="secret"
        )
        cls.post = Post.objects.create(
            title="A good title",
            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):
        response = self.client.get("/")
        self.assertEqual(response.status_code, 200)

    def test_url_exists_at_correct_location_detailview(self):
        response = self.client.get("/post/1/")
        self.assertEqual(response.status_code, 200)

    def test_post_listview(self):
        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):
```

```
        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")

    def test_post_createview(self):
        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):
        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):
    response = self.client.post(reverse("post_delete", args="1"))
    self.assertEqual(response.status_code, 302)
```

For test_post_createview, we make a fresh response and make sure it corresponds to the last() object on our model, checking that the page has a 302 redirect status code. The test_post_updateview function checks to determine if the initial post made in setUpTestData may be updated. Test _post_deleteview, the last newly added test, verifies that a 302 redirect is issued when a post is deleted.

Even while we have some coverage for our new features, we know there is room for improvement in terms of the number of tests we've run. Press Control+c to terminate the local web server, then proceed with the testing. Every single one of them ought to be okay.



We've developed a Blog app with minimal code that supports adding, editing, and removing blog entries. CreateRead-Update-Delete (or CRUD for short) describes these fundamental actions. While there may be other ways to accomplish this same goal (such as using function-based views or custom class-based views), we've shown how little code is required in Django to do this.

Chapter 7- User Accounts

We have a functional blog app with forms, but we lack a crucial component of most web apps: user authentication.

Proper user authentication is notoriously difficult to accomplish, and several security gotchas are along the way. Django already has a robust authentication system98 built in, which we can modify to meet our needs.

Django's default settings include the auth app, which provides us with a User object that consists of the following fields: username, password, email, first name, and last name.

We'll use this User object to log in, log out, and sign up on our blog.

User Login Access

Django's LoginView offers us a ready-made login screen. There are only a few things left to do, like updating our settings.py file and adding a URL pattern for the auth system and a log in template.

The django project/urls.py file must be modified first. The accounts/ URL is where you may access the login and logout pages. This modification involves adding a single line to the text on the second-to-last line.

```
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("", include("blog.urls")),
]
```

By default, Django looks for a log in form in a templates directory called registration called login.html. Therefore, we must make a new folder named "registration" and place the necessary file within it. To end our local server,

use Control+c at the command prompt. The next step is to make the new folder.

```
mkdir templates/registration
```

Create a new template file in the new registration folder called login.html. This is the code for the login.html file:

```
{% extends "base.html" %}
{% block content %}
<h2>Log In</h2>
<form method="post">{% csrf_token %}
  {{ form.as_p }}
  <button type="submit">Log In</button>
</form>
{% endblock content %}
```

After a successful login, we must tell the system where to send the user. With the LOGIN REDIRECT URL setting, we can do this. Just add the following at the end of the settings.py file in django_project:

```
LOGIN_REDIRECT_URL = "home"
```

Now the user is redirected to our homepage, 'home'. And at this moment, our work is complete. Once you've restarted the Django server with python manage.py runserver, you should be able to see our login page at http://127.0.0.1:8000/accounts/login/.

After entering our superuser credentials, we were sent back to the main page.

Remember that we didn't have to manually develop a database model or implement any view logic because Django's authentication system already did that for us.

Calling the User’s Name on The HomePage

It would be a good idea to make a change to our base.html template that would show a message to all visitors, whether they are signed in or not. The is_authenticated attribute can be used for this purpose.

It will do for now to simply make this code easy to find. We can give it a better look later on when we have more time. Modify the base.html file by inserting new code behind the </header> tag.

This is the updated base.html file:

```
{% load static %}
```

```
<html>

<head>
  <title>Django blog</title>
  <link href="https://fonts.googleapis.com/css?family=Source+Sans+Pro:400"
rel="stylesheet">
  <link href="{% static 'css/base.css' %}" rel="stylesheet" s>
</head>

<body>
  <div>
    <header>
      <div class="nav-left">
        <h1><a href="{% url 'home' %}">Django blog</a></h1>
      </div>
      <div class="nav-right">
        <a href="{% url 'post_new' %}">+ New Blog Post</a>
      </div>
    </header>
    {% 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 %}
    {% block content %}
    {% endblock content %}
  </div>
</body>

</html>
```

This code will say a user's name and display Hello if they are logged in. Otherwise, it will be a link to our new login page.

User Log Out Access

We included logout template page logic, but how do we log out? We can do it manually in the Admin panel, but there's a better approach. Let's add a log out link that goes to the home page. With Django auth, this is easy.

Just below our user greeting, add a % url 'logout' % link in our base.html file.

This is the updated script:

```
{% load static %}
<html>

<head>
  <title>Django blog</title>
  <link href="https://fonts.googleapis.com/css?family=Source+Sans+Pro:400"
rel="stylesheet">
  <link href="{% static 'css/base.css' %}" rel="stylesheet" s>
</head>

<body>
  <div>
    <header>
      <div class="nav-left">
```

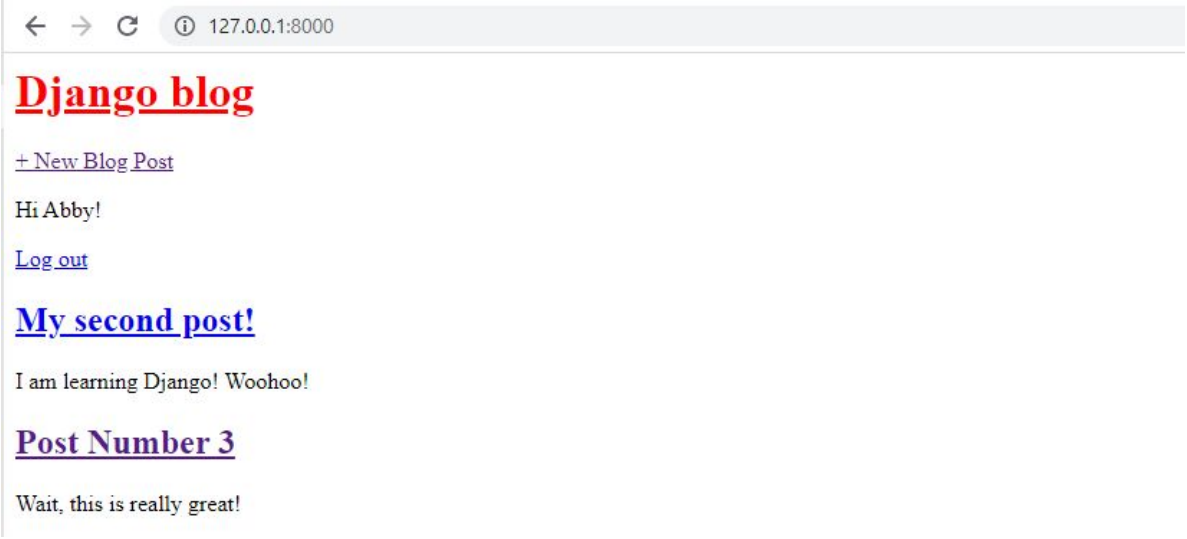
```
            <h1><a href="{% url 'home' %}">Django blog</a></h1>
        </div>
        <div class="nav-right">
            <a href="{% url 'post_new' %}">+ New Blog Post</a>
        </div>
    </header>
    {% if user.is_authenticated %}
    <p>Hi {{ user.username }}!</p>
    <p><a href="{% url 'logout' %}">Log out</a></p>
    {% else %}
    <p>You are not logged in.</p>
    <a href="{% url 'login' %}">Log In</a>
    {% endif %}
    {% block content %}
    {% endblock content %}
  </div>
</body>

</html>
```

Django auth app provides the essential view. We must indicate where to send logged-out users.

Update django project/settings.py with LOGOUT REDIRECT URL. We can add it next to our login redirect, so the file should look like this:

```
LOGOUT_REDIRECT_URL = "home"
```

You'll see a "log out" link if you refresh the homepage.



Go ahead. Click it and see where it leads.

Allow Users to Sign Up

To register new users, we need to create our own view. However, Django supplies us with a form class called UserCreationForm to make this process easier. By default, it has three fields: username, password1, and password2.

Code and URL structure can be organized in numerous ways for user authentication. Stop the local server by pressing Ctrl + C, and make a new app called "accounts" for our sign-up page.

```
python manage.py startapp accounts
```

Add the app to django_project under INSTALLED APPS in the settings.py file.

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "blog.apps.BlogConfig",
    "accounts.apps.AccountsConfig",
]
```

Add a new URL path to this app in urls.py of the django project folder below the built-in auth app.

```
path("accounts/", include("accounts.urls")),
```

Django reads this script top-to-bottom. Thus url order matters. When we request /accounts/signup, Django first looks in auth, then accounts.

Create a urls.py file in the new accounts folder using your text editor. Fill it with the following code:

```
from django.urls import path
from .views import SignUpView

urlpatterns = [
    path("signup/", SignUpView.as_view(), name="signup"),
]
```

Now let’s create the view. The view implements UserCreationForm and CreateView. Go to accounts/views.py and fill in with the following code:

We subclass CreateView in SignUpView. We use signup.html's built-in UserCreationForm and uncreated template. After successful registration, reverse lazy redirects the user to the login page.

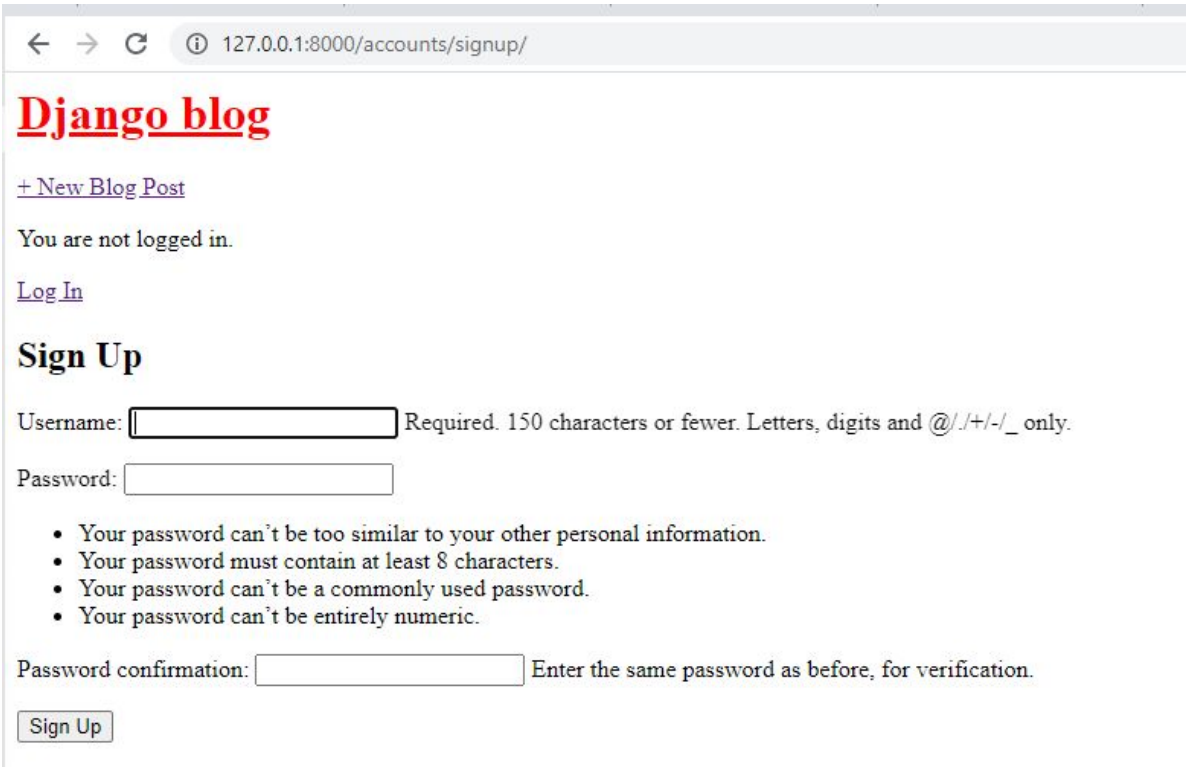
Why is reverse lazy used here rather than reverse? All generic class-based views don't load URLs when the file is imported. Therefore we use reverse's lazy form to load them afterward.

Create signup.html in the templates/registration/ folder. Add the following code.

```
{% 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 familiar. We extend our base template at the top, add our logic between <form></form> tags, use the csrf token for security, and provide a submit button.

Finished! To test it, run python manage.py runserver and visit http://127.0.0.1:8000/accounts/signup.

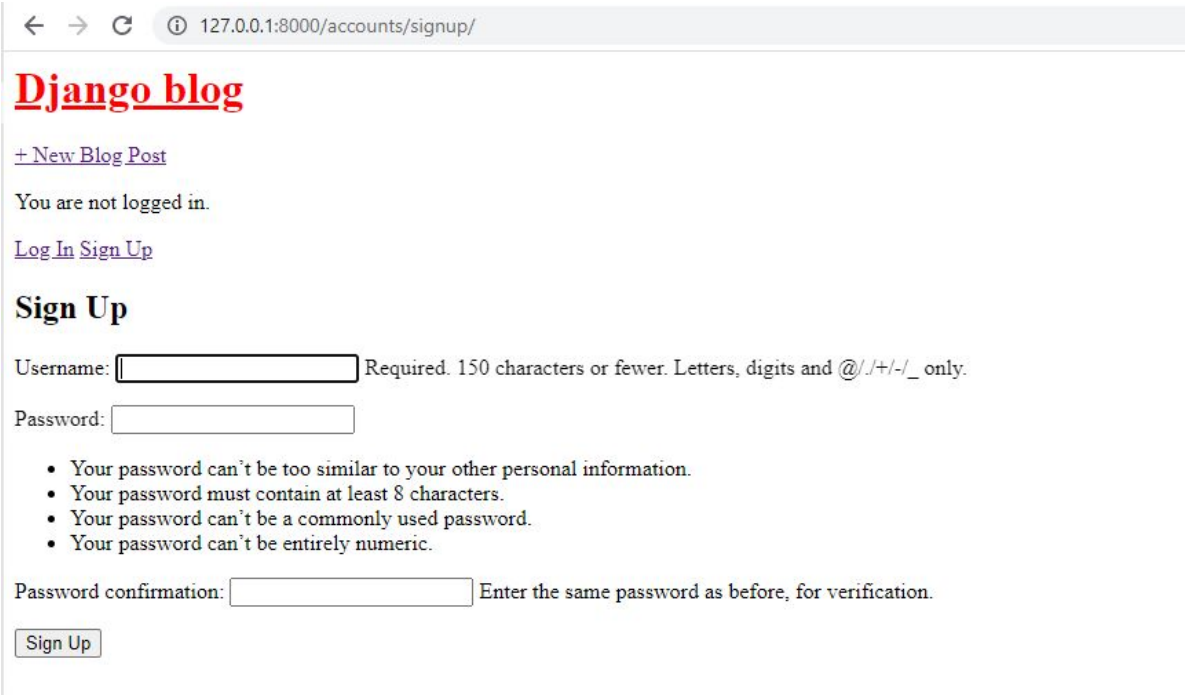


Link to Sign Up

Add a signup link on the logged-out homepage. Our users can't know the exact URL. We may add the URL to our template. In accounts/urls.py, we gave it the name signup, so that's all we need to add to base.html with the url template tag, exactly like our other links.

Add "Sign Up" underneath "Log In"

```
<a href="{% url 'signup' %}">Sign Up</a>
```



Looks much better!

GitHub

We haven't made a git commit in a while. Do that, then push our code to GitHub. First, check git status for new changes.

git status

git add -A

git commit -m "forms and user accounts"

Create a new repo on GitHub. I'll call it blog. After creating a new GitHub repo, I can input the following commands. Replace macvicquayns with your GitHub username.

git remote add origin https://github.com/MacVicquayns/blog.git

git branch -M main

git push -u origin main

Static Files

Previously, we configured our static files by establishing a static folder, directing STATICFILES DIRS to it, and adding % load static % to our base.html template. We need a few extra steps because Django won't support static files in production.

First, use Django's collectstatic command to assemble all static files into a deployable folder. Second, set the STATIC_ROOT setting to the staticfiles folder. Third, set STATICFILES_STORAGE, collectstatic's file storage engine.

Here's what the revised django project/settings.py file looks like:

```
120   STATIC_URL = "/static/"
121   STATICFILES_DIRS = [BASE_DIR / "static"]
122   STATIC_ROOT = BASE_DIR / "staticfiles"
123   STATICFILES_STORAGE = "django.contrib.staticfiles.storage.StaticFilesStorage"
124
```

Now go to the command line and run python manage.py collectstatic:

```
(.venv) PS C:\Users\Jide\OneDrive\Desktop\script\blog> python manage.py collectstatic

129 static files copied to 'C:\Users\Jide\OneDrive\Desktop\script\blog\staticfiles'.
```

A new staticfiles folder containing an admin and a css folder has been added to your project folder. The admin is the static files from the default admin, and the css is our own. The collectstatic command must be executed before each new deployment in order to compile the files into the staticfiles folder that is then utilized in production. To avoid overlooking it, this process is commonly automated in larger projects, but that is outside the scope of our current work.

There are a number of methods for delivering these precompiled static files in production, but we'll be using the WhiteNoise package, which is currently the most popular option.

To begin, install the newest version with pip:

python -m pip install whitenoise==5.3.0

Then update django project/settings.py:

- Add whitenoise above staticfiles in INSTALLED APPS
- Add WhiteNoiseMiddleware to MIDDLEWARE.
- Swap WhiteNoise for STATICFILES STORAGE

The updated file should look like this:

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "whitenoise.runserver_nostatic",
    "blog.apps.BlogConfig",
    "accounts.apps.AccountsConfig",
]

MIDDLEWARE = [
    "django.middleware.security.SecurityMiddleware",
    "django.contrib.sessions.middleware.SessionMiddleware",
    "whitenoise.middleware.WhiteNoiseMiddleware",
    "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"
STATICFILES_STORAGE = "whitenoise.storage.CompressedManifestStaticFilesStorage"
```

After all, these, rerun python manage.py collectstatic.

There will be a small warning. This will overwrite existing files! You sure? Enter "yes" WhiteNoise regenerates the static files in the same folder.

Static files are difficult for newbies, so here's a quick recap of our Blog site's stages. In Chapter 5, we built a top-level static folder for local development and changed STATICFILES DIRS. In this chapter, we added STATIC ROOT and STATICFILES STORAGE parameters before running

collectstatic, which assembled all static files into a single staticfiles folder. Installed whitenoise, updated INSTALLED APPS, MIDDLEWARE, and STATICFILES STORAGE, then ran collectstatic.

Most developers, like myself, have difficulties remembering these procedures and rely on notes.

Time for Heroku

Here we are, at the third attempt at using Heroku to launch a website. Set up Gunicorn as your primary web server:

```
python -m pip install gunicorn==20.1.0
```

Create a requirements.txt file to store the current virtual environment's contents with this command.

```
python -m pip freeze > requirements.txt
```

In django project/settings.py, update ALLOWED HOSTS.

```
ALLOWED_HOSTS = [".herokuapp.com", "localhost", "127.0.0.1"]
```

Also, make sure you have a manage.py file and a Procfile and runtime.txt file in the root folder of our project.

Put this code in the Procfile:

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

Put your current version of Python in the runtime.txt file and save.

Now, check the git status, and push everything to your GitHub. Run these commands in this order:

```
git status
```

```
git add -A
```

```
git commit -m "Heroku config"
```

```
git push -u origin main
```

Deploy to Heroku

Log in to your Heroku account from the command line.

```
heroku login
```

Heroku will then create a new container where our application will reside once the create command has been executed. If you don't specify a name and just run heroku create, Heroku will come up with one for you at random; however, you are free to choose your own name, provided it is unique on Heroku. You can't use the name d12-blog because I've already used it. You must use a different alphabetic and numeric sequence.

```
heroku create d12-blog
```

The prior apps did not have static file configurations. Thus we used heroku config:set DISABLE COLLECTSTATIC=1 to prevent Heroku from running the Django collectstatic command automatically. But now that we have static files set up, we can relax and let this happen automatically during deployment.

Adding a web process to Heroku and pushing our code there will get the dyno up and running.

```
git push heroku main
```

```
heroku ps:scale web=1
```

Your new app's URL can be found in the terminal output or by typing "heroku open."

PostgreSQL vs SQLite

We have been using Django's preconfigured SQLite database on our local machines and in production so far in this book. It's far simpler to set up and use than a server-based database. Although it's convenient, there is a price to pay for it. Because Heroku uses a transient file system, any modifications made to the cloud-based db.sqlite3 file are lost anytime a new deployment or server restart takes place. On the free tier that we are now using, the servers may be rebooted as frequently as once every 24 hours.

This ensures that any changes made to the database in a development environment may be replicated in a production environment with a simple push. However, new blog posts or changes you make to the live website won't last forever.

Thanks to some spare code, our Blog site now has sign up, log in, and log out capabilities.

Several potential security issues can arise when developing a custom user authentication method, but Django has already dealt with them. We deployed our site to Heroku with the static files set up correctly for production. Well done!

CONCLUSION

The completion of this fantastic Django course is a cause for celebration. We began with nothing and have already completed five separate web apps from scratch using all of the primary capabilities of Django, including templates, views, urls, users, models, security, testing, and deployment. You should now feel confident creating your own cutting-edge websites with Django.

Putting what you've learned into practice is essential if you want to become proficient at it. Our Blog and Newspaper sites share a feature known as CRUD (Create-Read-Update-Delete) with a wide variety of other web apps. Can you, for instance, develop a web-based to-do list? Will you create an app similar to Twitter or Facebook? You don't need anything else because you already have it all. The ideal way to learn the ropes is to construct many simple projects and gradually increase their complexity as you gain experience and knowledge.

Follow-Up Actions

There is a lot more to learn about Django than what we covered in this book. This is crucial if you plan on creating massive websites with hundreds of thousands or even millions of monthly visitors. There's no need to look further than Django itself for this. Django for Professionals is a follow-up book I wrote that covers topics like using Docker, installing a production database locally like PostgreSQL, registering advanced users, securing the site, optimizing performance, and much more.

When building mobile apps (iOS/Android) or websites with a dedicated JavaScript front-end framework like Vue, React, or Angular, Django is often utilized as the back-end API. Django REST Framework181, a third-party program that is tightly integrated with Django itself, makes it possible to convert any preexisting Django website into an API with no additional coding. If you're interested in reading more, I've devoted a complete book to the subject, entitled Django for APIs.

Third-party bundles

As we’ve seen in this book, 3rd party packages are an essential element of the Django ecosystem, especially regarding deployment or enhancements surrounding user registration. It’s not uncommon for a professional Django website to rely on dozens of such packages.

Caution: don't install 3rd party packages only to save a little time now. Any additional packages increase the chances that their maintainer won't fix all bugs or upgrade to the newest version of Django. Learn its uses.

Django Packages is a complete database of all available third-party apps if you're interested in viewing additional packages.